Generation of Concurrency Controls using Discrete-Event Systems

by

Christopher William Arthur Dragert

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
September 2008

Preface

The development of controls for the execution of concurrent code is non-trivial. This work shows how existing discrete-event system (DES) theory can be successfully applied to this problem. From code without concurrency controls and a specification of desired behaviours, a DES representation of the problem is obtained, and then used to generate concurrency control code. By applying rigorously proven DES theory, the resulting code comes with guarantees not present in similar works. All control schemes generated in DES are nonblocking, yielding code that is free of both livelock and deadlock. Additionally, the generated control scheme is minimally restrictive, meaning only problematic behaviours are prevented. If the specifications cannot be enforced as presented, the largest controllable subset is instead enforced. The result, which requires no further interaction to generate, is the best possible control scheme given the interaction between the specifications and the original code. Existing methods encounter difficulties when faced with multiple specifications that interact to form deadlocks. Modular DES theory is successfully applied, allowing resolution of these conflicts without requiring the user to introduce new specifications. Moreover, the approach is independent of specific programming or specification languages. A Java implementation is given, along with two problems showing the process in action.

Acknowledgments

I would like to thank Dr. Juergen Dingel and Dr. Karen Rudie for their support. Throughout this process, they have both shown the utmost commitment to academic freedom, and have given me a wide latitude in choosing and conducting my research. Their advice and support along the road has been invaluable, and much of the credit for this research belongs to their thoughtful supervision.

I would also like to thank my Mom and Dad. Without their loving support, none of this would have been possible. My gratitude knows no bounds.

Table of Contents

List of Figures

Chapter 1

Introduction

Concurrency is going mainstream. Leaders in the hardware and software industries along with academics agree that in just a few years even average programmers will have to be able to write concurrent code effectively and efficiently [33, 40]. Although concurrent programming has been studied for over four decades, current software development and programming language technology has not yet succeeded in making the design and implementation of correct concurrent code in everyday practice an easy undertaking. At present, our ability to develop concurrent code is still insufficient. As recently as 2006, Edward Lee wrote [27]:

> "I conjecture that most multi-threaded general-purpose applications are, in fact, so full of concurrency bugs that as multi-core architectures become commonplace, these bugs will begin to show up as system failures."

Lee reasoned that this was caused by the intrinsic difficulty of comprehending the non-determinism introduced by the threading model of concurrency. This is echoed by Herb Sutter in [40] as he states:

> "Probably the greatest cost of concurrency is that concurrency really is hard: The programming model, meaning the model in the programmer's head that he needs to reason reliably about his program, is much harder than it is for sequential control flow."

If a programmer cannot properly reason about his software, bugs becomes a distinct possibility. Richard Adhikari speaks directly to this issue in [1]:

> "Writing parallel computer programs is more difficult than writing standard, sequential ones because concurrency introduces entirely new potential software bugs. The most common of these are race conditions or race hazards, where there's a flaw in a system or process in which the output or the result, or both, are unexpectedly and critically dependent on the sequence or timing of other events."

While the difficulty of developing concurrent software is generally acknowledged, the prevalence of concurrent software continues to grow. This gap is dangerous for the software community, and is one that must be addressed. The notable David Patterson, former president of the ACM, states [35]:

> "...From my perspective, parallelism is the biggest challenge since high-level programming languages. It's the biggest thing in 50 years because industry is betting its future that parallel programming will be useful.
>
> Industry is building parallel hardware, assuming people can use it. And I think there's a chance they'll fail since the software is not necessarily in place. So this is a gigantic challenge facing the computer science community. If we miss this opportunity, it's going to be bad for the industry.

> Imagine if processors stop getting faster, which is not impossible. Parallel programming has proven to be a really hard concept. Just because you need a solution doesn't mean you're going to find it."

Long aware of the importance of this issue, computer scientists have tried, with some success, to address the problem. Existing work is largely divisible into two separate approaches:

New programming models and programming language abstractions: The goal is to conceive high-level concepts that allow the benefits of concurrency to be reaped while keeping its complexities in check. Seminal work by Dijkstra on semaphores [12], by Hoare on monitors [19], and by Brinch Hansen on languages [18] falls into this category. The intervening years have led to the creation of many more concurrency controls, often building on the fundamental controls introduced by the earlier papers. The concurrency library in Java [16] is fairly representative of the current state of the art.

These works have led to specific constructs that are used in code to enforce a concurrency control scheme. Controls of this nature provide convenient methods to *implement* concurrency control, but only *after* the the concurrent control scheme is determined. They do not inform the developer as to what concurrency controls should be included, where they should be placed, or how the controls will interact. A developer using them must still analyze the non-deterministic behaviour of the software before deciding how the controls should be applied to meet the objectives of the developer, a decidedly non-trivial task.

Automatic generation of concurrency control code from specifications: Rather than develop the concurrency control code manually, the programmer instead specifies the desired concurrent behaviour. Concurrent control code is then generated automatically. An example of early work on this idea is based on Habermann's path expressions [6]. Similar work exists with approaches differing in the kind of specification notations supported and the guarantees that the generated code provides. Chapter 6 provides a brief survey of these approaches.

A novel approach was taken by Emerson and Clarke, in [15]. An abstract model of the program is created, suppressing detail irrelevant to concurrency control. Specifications given in computational tree logic (CTL) are used to create and insert concurrency controls into the model. However, the specifications must be consistent, and both livelocks and deadlocks are possible, depending on how the specifications and the plant interact.

While both research directions intend to facilitate development by lifting the levels of abstraction, the second is more radical in that the concurrency control code is automatically generated. The realization of this vision could clearly benefit from the development of new programming models and concepts.

Recent approaches such as [13] use a model similar to Emerson and Clarke, but are still missing a key component. Specifications only state how a program should behave; they do not provide a complete control plan that will result in the program behaving as instructed by the specifications. Interactions between specifications, and interactions between specifications and code can result in unintended and undesirable behaviours, such as deadlock. In [4], there is a system to prevent deadlocks by finding 'last chance states', the last chance to prevent some action that would make deadlock

inevitable. However, it is unclear on the mathematical basis. Is the resulting system minimally restrictive? Does it prevent livelock? Is deadlock-freeness guaranteed under all conditions? Questions aside, this work addresses the interaction between specifications and the actual code.

In this research, we put forward a new process to automatically generate concurrency control that addresses many of these concerns. The central distinguishing feature of this work is the way in which the concurrency control code is computed. To this end, we leverage well-established work in the domain of control theory. More precisely, we employ "supervisory control synthesis" which was first proposed by Ramadge and Wonham [36] in order to facilitate the design of discrete-event systems (DES). In short, this process works as follows: From a system P generating events and a specification E describing allowed event sequences, a supervisor S is generated such that the composition of P and S exhibits a "minimally restrictive" subset of the allowed event sequences in E and is guaranteed nonblocking. We leverage this process by showing how the event-generating system P can be constructed from code that is devoid of concurrency controls, but contains user markup indicating events relevant to the specifications. The supervisor generated is then transformed into concurrency control code that enforces the specification. The central features of the resulting approach are as follows:

- **Precision**: Strong, precise, theoretically proven guarantees can be given about the generated code (adherence to specification, deadlock-freedom, and maximal permissiveness).

- **Generality**: Any notation allowing the specification of event orderings can be used; moreover, the approach is programming language independent in the sense

that control code for any language offering basic synchronization primitives can be generated.

- **Potential for extension**: Many extensions to the DES supervisory control problem have been developed. It is very likely that many of these will be applicable in future work.

While results from DES control theory have already been used for verification and analysis [49], to the best of the author's knowledge, this work is the first to suggest the use of supervisor synthesis for the generation of concurrency control code.

This approach targets the most troublesome aspect of developing concurrent code by taking into account all possible thread interleavings as part of developing a control plan. In addition, modular DES theory [46] provides a method to resolve conflicting specifications. If multiple specifications lead to a deadlock when combined, the process is able to identify and overcome the potential deadlock by enforcing the largest controllable sublanguage of the specification. Introduction of a new specification is unnecessary, as is the case with most existing procedures.

In this work, the process to apply DES to code is explored. Chapter 2 presents relevant background in DES and concurrency theory. Chapter 3 shows how concurrent code, lacking concurrency controls, along with a set of specifications, can be transformed into a DES model. Chapter 4 uses that model to generate concurrency controls with all the guarantees provided by DES. Chapter 5 presents an extended problem illustrating how the process overcomes conflicting specifications. Chapter 6 shows how this work relates to existing research, and Chapter 7 presents conclusions and explores future work. All automata are drawn using IDES [20], and DES operations are performed in both IDES and TCT [41].

Chapter 2

Background Theory

The research presented in this thesis addresses problems in concurrent programming using discrete-event systems theory. Typically, these two fields are disparate and will be presented as such. Special attention will be paid to those aspects that are used to connect the two research areas.

2.1 Discrete-Event Systems

Discrete-Event Systems (DES) is a formal branch of control theory, distinguished through the modeling of events as discrete, rather than continuous. As such, events are considered to be both time-independent and instantaneous. States of the system are also discrete, and change with the occurrence of events. For example, a messaging protocol could be modeled as a DES using events such as 'message sent', 'message received', 'message lost', and so on. At a high level, the objective in a DES supervisory control problem is to synthesize a supervisor such that the system exhibits the largest possible subset of desired behaviors. The Ramadge and Wonham formulation of DES

Theory [38], which we adopt, is built on language theory using deterministic finite-state automata. The following presentation largely mirrors that given by Cassandras and Lafortune in [7].

2.1.1 Automata and Languages

In language theory, an *alphabet* Σ contains symbols σ that are concatenated to form a *string* s. Strings may be concatenated with symbols ($s\sigma$) or other strings ($s_1 s_2$) to form larger strings. The set of all possible strings is given by Σ^*. A *language* L is a set of strings across an alphabet, given by $L \subseteq \Sigma^*$. The *prefix-closure* of L is defined as $\overline{L} = \{s \in \Sigma^* \mid \exists t \in \Sigma^*(st \in L)\}$.

Languages can be represented using deterministic finite-state automata (DFA). A DFA F is a 5-tuple, and is given by $F = (Q, \Sigma, \delta, q_0, Q_m)$ where

- Q is a finite set of states
- Σ is the alphabet of the FSA
- $\delta = Q \times \Sigma \rightarrow Q$ is the transition function. The function $\delta(q, \sigma) = q'$, where $q, q' \in Q$, q may equal q', and $\sigma \in \Sigma$, defines a transition from q to q' upon the occurrence of σ
- $q_0 \in Q$ is the initial state
- $Q_m \subseteq Q$ is the set of marked states (or final states, used interchangeably)

The automaton F, always starting from q_0, produces strings through a series of zero or more event occurrences as defined by δ. As a shorthand, the transition function is expanded recursively to take strings as input:

$\delta'(q, \epsilon) = q$, where ϵ is the null (or empty) string

$\delta'(q, s\sigma) = \delta(\delta'(q, s), \sigma)$ for $s \in \Sigma^*$ and $\sigma \in \Sigma$

Two languages are produced by a DFA: a *generated* and a *marked* language, denoted by $\mathcal{L}(F)$ and $\mathcal{L}_m(F)$, respectively, and defined as

$$\mathcal{L}(F) = \{s \in \Sigma^* \mid \delta'(q_0, s) \text{ exists}\}$$

$$\mathcal{L}_m(F) = \{s \in \mathcal{L}(F) \mid \delta'(q_0, s) \in Q_m\}.$$

Thus, the generated language is the set of all possible paths through the DFA, while the marked language is the subset of the generated language with paths that end on a marked state. The language marked by a DFA is a *regular language*. Additionally, if a DFA marks a language, then it is said to be a *recognizer* for that language.

A DFA is said to be *blocking* if $\overline{\mathcal{L}_m(F)} \subset \mathcal{L}(F)$, and *nonblocking* if $\overline{\mathcal{L}_m(F)} = \mathcal{L}(F)$. In other words, in any state $q \in Q$, where Q is a nonblocking DFA, there exists a string s such that $\delta'(q, s) \in Q_m$. In a blocking DFA, there is at least one q where there does not exist such an s. This notion of path completion to a marked state will be used in the discussion of concurrency.

Two operations, *parallel composition* and *product*, are used to combine DFAs. Both take two DFAs as input and produce a new DFA as output, and both are regularly used in DES theory. The parallel composition, or synchronous product of DFAs F_1 and F_2 is given by $F_1 \| F_2 = (Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta, (q_{01}, q_{02}), Q_{m1}, Q_{m2})$, where

$$\delta(\sigma,(q_1,q_2)) = \begin{cases} (\delta_1(\sigma,q_1),q_2) & \text{if } \sigma \in \Sigma_1 \backslash \Sigma_2 \text{ and } \delta_1(\sigma,q_1) \text{ is defined} \\ (q_1,\delta_2(\sigma,q_2)) & \text{if } \sigma \in \Sigma_2 \backslash \Sigma_1 \text{ and } \delta_2(\sigma,q_2) \text{ is defined} \\ (\delta_1(\sigma,q_1),\delta_2(\sigma,q_2)) & \text{if } \sigma \in \Sigma_1 \cap \Sigma_2 \text{ and } \delta_1(\sigma,q_1) \text{ and} \\ & \quad \delta_2(\sigma,q_2) \text{ are defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

This operation allows unshared events to interleave freely, while shared events may only occur when allowed by both automata. In the special case that there are no shared events, the two automata simply interleave and the operation is instead called the *shuffle*. The product, or intersection, of F_1 and F_2 is given by $F_1 \cap F_2 = (Q_1 \times Q_2, \Sigma_1 \cap \Sigma_2, \delta, (x_{01}, x_{02}), Q_{m1} \times Q_{m2})$, where

$$\delta(\sigma,(q_1,q_2)) = \begin{cases} (\delta_1(\sigma,q_1),\delta_2(\sigma,q_2)) & \text{if } \delta_1(\sigma,q_1) \text{ and } \delta_2(\sigma,q_2) \text{ are defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Only those events contained in both DFAs can be executed in the resulting DFA, and then only in unison.

2.1.2 Supervisory Control

With the necessary automata theory in place, relevant areas of DES theory can now be presented. The system model, represented as a DFA, is referred to as the *plant*, given by G. The plant generates events, with $\mathcal{L}(G)$ describing all possible event sequences. This is sometimes referred to as the plant behaviour. The marked states represent the desired end-points of the system, and imply that the system can terminate gracefully once those states are reached. The marked language $\mathcal{L}_m(G)$ shows the possible paths to these final states. For example, a manufacturing line could have several events

corresponding to the stages in manufacturing, while the completion of a part could be an event transitioning to a marked state.

Events are either *controllable* or *uncontrollable*. A controllable event can be *disabled*, meaning its occurrence can be prevented, and *enabled*, meaning its occurrence is allowed. Enabling an event does not force its occurrence. An uncontrollable event cannot be disabled, and is considered to be 'always enabled'. In a manufacturing example, placing a part in a buffer is usually controllable, while having a machine break down would be uncontrollable. The event set Σ is broken into a set for controllable events, Σ_c and a set for uncontrollable events Σ_u, where $\Sigma = \Sigma_c \cup \Sigma_u$ and $\Sigma_c \cap \Sigma_u = \emptyset$.

Supervisory control introduces a *supervisor* S, which is in the form of a DFA and uses the same event set as G. A supervisor controls a plant by enabling and disabling events in the plant. When the supervisor is coupled with the plant in this manner, the resulting system is called the *closed-loop system* and is given by S/G. A *control action*, given by $S(x)$, is the set of all enabled events at supervisor state x. The set of all control actions is the *control policy*. Figure 2.1 shows the feedback relation between the supervisor and plant. The plant generates events that are observed by the supervisor, and in turn the supervisor responds with the appropriate control action as dictated by the control policy.

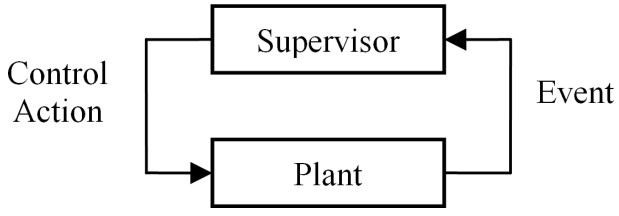


Figure 2.1: Basic supervisory control loop.

A specification E, usually in the form of a DFA, gives the set of desired behaviours

for the closed-loop system and is known as the *legal language*. The typical goal in a DES problem is to synthesize a supervisor S such that only those behaviours contained in E are generated in S/G. When this is the case, S is said to *enforce* E. Supervisory control is limited to enabling and disabling events in G, so specifications outside of G are irrelevant. For this reason, specifications are usually constructed such that $E \subseteq \mathcal{L}_m(G)$. If this condition is not met, then $E' = E \cap \mathcal{L}_m(G)$ can be used without loss of generality. Thus, a specification can be said to give the set of desirable event orderings from all the orderings possible in the plant.

A specification is *controllable* with respect to G if, for all $s \in \mathcal{L}(E)$, and for all $\sigma \in \Sigma_u$, $s\sigma \in \mathcal{L}(G)$ implies $s\sigma \in \mathcal{L}(E)$. In other words, if an event cannot be prevented, it must be allowed for the system to be controllable. It has been proven that a supervisor can always be generated to enforce a controllable language [38].

If a specification does not meet the controllability condition, then it is called *uncontrollable* and no supervisor can enforce it. This problem is addressed by Ramadge and Wonham in [45], where they develop the idea of the supremal controllable sublanguage. Given by $\sup \underline{C}(E)$, it is the largest sublanguage of E that is controllable. It is possible that $\sup \underline{C}(E) = \emptyset$, and thus it is usually required that some minimally acceptable language A be contained in $\sup \underline{C}(E)$, giving the relation $A \subseteq \sup \underline{C}(E) \subseteq E$. If E is controllable, then $\sup \underline{C}(E) = E$. For this reason, $\sup \underline{C}(E)$ is usually enforced by a supervisor rather than E, since $\sup \underline{C}(E)$ is always controllable and is equal to E if E is controllable. An S constructed to enforce $\sup \underline{C}(E)$ would do so by ensuring that $\mathcal{L}_m(S/G) = \mathcal{L}_m(\sup \underline{C}(E))$.

All supervisors given in this text are *implicit*, meaning that the transition function of S is undefined on any event that is disabled by the supervisor. When a supervisor

is implicit, the closed-loop system is simply $S/G = S \cap G$. Given this construction, a supervisor S that recognizes $\sup \underline{C}(E)$ will necessarily satisfy $\mathcal{L}(S/G) = \sup \underline{C}(E)$. A supervisor is thus most easily generated by building a recognizer for $\sup \underline{C}(E)$. This reduces the problem to finding $\sup \underline{C}(E)$. An algorithm accomplishing this is given in [7]. The stated complexity is $O(n^2m^2|\Sigma|)$, where n is the number of states in E, and m is the number of states in G.

This method of supervisor construction guarantees nonblocking for free. First, note that

$$\mathcal{L}_m(S/G) = \sup \underline{C}(E) \tag{2.1}$$

and furthermore, it can be found that

$$\mathcal{L}(S/G) = \overline{\sup \underline{C}(E)} \; . \tag{2.2}$$

By taking the prefix-closure of (2.1), an expression for $\overline{\sup \underline{C}(E)}$ is obtained, which can then be subsituted into (2.2). The result is $\overline{\mathcal{L}_m(S/G)} = \mathcal{L}(S/G)$, which is the nonblocking condition. Thus, S is nonblocking.

In the DES software packages TCT [41] and IDES [20], commands exist to automatically generate a nonblocking S that enforces the largest controllable sublanguage of a specification E on a plant G. The process is straightforward, and only G and E need to be supplied as input to create such a supervisor.

In modular DES theory [46], multiple specifications can be synthesized into multiple supervisors. However, when multiple nonblocking supervisors are applied to a plant, there is no assurance that the resulting system is nonblocking. Given two supervisors S_1 and S_2 enforcing specifications E_1 and E_2, respectively, on a plant G, the closed-loop system is nonblocking if and only if

$$\overline{\mathcal{L}_m(S_1/G) \cap \mathcal{L}_m(S_2/G)} = \overline{\mathcal{L}_m(S_1/G)} \cap \overline{\mathcal{L}_m(S_2/G)}.$$

Languages satisfying this property are said to be *non-conflicting*. If supervisors are not non-conflicting, then the *monolithic supervisor* S_M is instead used. To compute S_M, the *monolithic specification* E_M must first be constructed. This is straightforward, with E_M being the product of all specifications, and is given by $E_M = E_1 \cap E_2$. Then, S_M is synthesized using standard supervisor synthesis with E_M as the specification to be enforced. Both the monolithic construction and the non-conflicting test extend to larger numbers of supervisors by adding identical terms for S_3, S_4,...,S_n.

2.2 Concurrent Programming

Concurrent programming describes the practice of developing software with components designed to run in parallel. A single program is broken up into multiple threads of execution, or *threads* for short. As an example, a program designed to communicate across a network could have a thread to send data, another thread to receive data, and a third thread to manage the overall system behaviour. Each thread executes its own statements sequentially.

Single processor multi-threading is common place in current operating systems (OS), whereby each thread is given a small block of processor time to execute its programming, and when the time is up, a *context switch* occurs and the next scheduled thread begins execution. The duration of each block of processing time and the ordering of process execution are both determined by the scheduler (a component of the OS) and can rapidly change based upon the current computing environment. With respect to the concurrent program attempting to run, the scheduler operates in a non-deterministic fashion. Simply put, the exact manner in which the various statements of different threads will interleave at run-time is unknowable *a priori*.

Correct concurrent software must take every possible interleaving into account.

Different threads must share information if they are to work together. This is often done through *shared memory*, where data is explicitly shared by multiple threads[1]. A common scenario is easily imagined wherein a writer thread records data into shared memory, followed by a reader thread subsequently reading that data. This scenario also introduces a common concurrency bug. If the interleaving is non-deterministic, how can we know that the reader thread will only read after the writer has completed? What if the thread attempts to read *while* the other thread is writing?

2.2.1 Concurrency Controls

Concurrent programming is essentially a thread synchronization program. Concurrency controls cause the programs to only interleave in permitted sequences. Concurrent programs usually contain some blocks of statements that must be controlled, and others that may interleave freely. These are known as *critical sections* and *non-critical sections*, respectively. For example, a series of statements that perform a write operation to shared memory would comprise a critical section.

The most common concurrency control is, by far, the semaphore. Originally described by Dijkstra in [12], the semaphore has become the standard synchronization primitive. A semaphore is a data structure with a counter and a corresponding queue in the scheduler. The most basic implementation provides a `Wait` and a `Signal` method as defined in Figs. 2.2 and 2.3, respectively.

[1]Message passing is also used, but will not be discussed here.

```
Wait {
  counter = counter - 1;
  if (counter < 0) wait_on_queue;
}
```

Figure 2.2: The `Wait` method for a semaphore.

```
Signal {
  counter = counter + 1;
  if (counter <= 0) wake_up_one_process_on_queue;
}
```

Figure 2.3: The `Signal` method for a semaphore.

If a thread waits on a semaphore, it goes onto a scheduling queue where it lies dormant until another thread signals it via `Signal`. A thread forced to wait is said to be *blocked* (while the term 'blocking' has a different meaning in DES theory, the intended usage should be clear through context). Through a pattern of waiting and signalling, a correct ordering of threads can be established. When a semaphore is initialized, it is given a number of permits. If that number is 0 or less, then the first thread to `Wait` on that semaphore will block while it waits on the queue. A number greater than 0 will allow that number of threads to pass without blocking. A semaphore used so that the number of permits is always 0 or 1 is called a *binary semaphore*.

Most operating systems now provide semaphore functionality, ensuring that semaphore operations modifing a counter are uniterruptable. This allows thread coordination to occur without worrying about the concurrent interactions of the concurrency controls themselves. A variety of concurrency controls, such as monitors,

locks, and thread pools, build off this basic structure. While these other controls are not used in this work, the interested reader is referred to an introductory text, such as [3].

2.2.2 Safety and Liveness Properties

Specifications give us system properties that must be met for the program to run as intended. Indirectly, they give the desired interleavings of the threads in the system. A specification is usually given as a formal logic formula. There exists a wide variety of applicable logic systems with different descriptive powers, including automata as described above, temporal and modal logic [14], and μ-calculus [25]. We will be using automata, so that an easy bridge can be formed to DES theory.

Regardless of the formalization employed, all specifications can descibed as members of one or both of two categories. *Safety properties* describe undesirable system states that must never occur. For example, a specification such as "The buffer must never overflow" is a safety property. *Liveness properties* declare that something (desirable) must eventually happen. Examples include "The program must eventually terminate" and "the receiver thread must eventually enter its critical section if it ever tries to enter its critical section". A property such as "The buffer must never overflow and the program must eventually terminate" would be both a safety and a liveness property. Lamport provides an accessible introduction to these concepts in [26].

Deterministic Finite-Automata are limited to describing safety properties only as they can only express regular languages. Modal and temporal logics, such as LTL, CTL, and CTL*, allow the specification of safety and liveness properties with varying degrees of expressiveness [14]. μ-calculus can also specify safety and liveness

properties and is the most inclusive out of the formalizations noted here.

2.2.3 Model Checking

Testing concurrent software presents a special difficulty. Since context switches can potentially occur after every line of code, the number of possible interleavings grows at an exponential rate. The state space quickly reaches sizes that are unmanageable through manual testing. *Model checking* performs an exhaustive search of the state space, checking every possible interleaving. Common checks include deadlock-freedom, assertion checks (in supporting languages), and verification of temporal logic specifications. Emerson et al. provide a summary of the topic in [9].

Depending on the model checker employed, specifications can be given in a variety of different logic formalizations, with modal and temporal logics being the most common. Typically, a developer will take a specification, create concurrency controls to satisfy that specification, and then run a model checker with the specification as input. If the check passes, then the specification has been satisfied. If it fails, then the process repeats, with help from the check results. Oftentimes this help comes in the form of an error trace showing the exact interleaving that led to the violation. However, it is also possible that the problem is untractable due to excessive program size and thus an unmanageable state space. In that case no answer is returned, and the user must adjust the model in hopes of making the problem tractable.

Chapter 3

Process Description

Given source code without any concurrency control, and given a set of informal specifications, the goal of this process is to generate concurrency control code and insert it back into the original code such that the resulting program satisfies the specifications. To generate a control policy, we apply supervisory control theories put forth in DES literature [36, 37, 38]. This chapter covers the process through to the creation of the DES model.

The fundamental connection between abstract DES theory and actual code is made through the identification of relevant events in the code. First, the code is instrumented to provide an event set. This will be used both to model the behaviour of the code in an FSA, and to formalize specifications also as an FSA. Standard DES operations are then performed to generate a supervisor guaranteed to satisfy the specifications. The control policy from the supervisor, guaranteed correct, nonblocking, and minimally restrictive, acts as input for an algorithm that automatically generates and inserts concurrency control code. The process is diagrammed in Fig. 3.1.

Part of the intent of this approach is to maximize automation, or at least provide

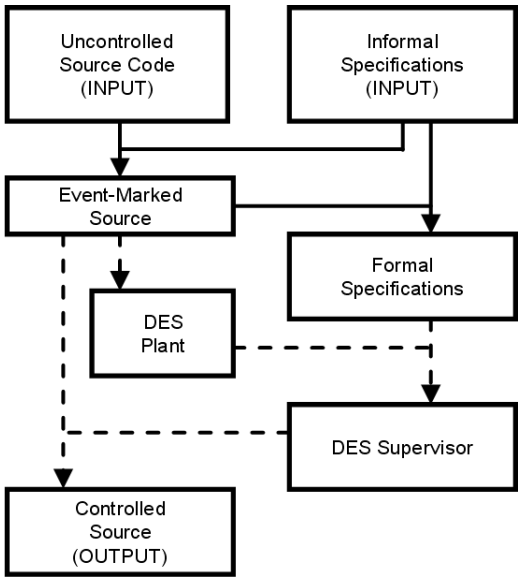


Figure 3.1: Process to create and inject concurrency controls into source code. Automatable steps are noted with a dotted line.

a process that is automatable. Marking relevant events and formalizing specifications must be done manually, but all other steps could be automated using a variety of methods.

The steps from Fig. 3.1 are described in more detail in the following list.

Input: Threaded source code without concurrency controls, and specifications at any level of formality giving the desired concurrent behavior of the program.

1. Build the set of relevant events. This set is used as part of the alphabet to describe the concurrent behavior of the software.

2. For each thread in the software, build a DFA that contains transitions for each relevant event in that thread, and introduces necessary structure-preserving irrelevant events. The language generated by the DFA must match the actual behaviour of the software.

3. Build the specifications. Specifications must also be represented as DFAs and should use the same alphabet as the DFAs from step 2.

4. Construct DES supervisors to enforce the specifications using the plant.

5. Build concurrency control code by realizing the supervisor(s). Weave the code into the marked source code to generate the actual closed-loop system.

Output: Source code with concurrency controls that enforce the given specifications.

The following sections of this chapter bring us from the start of the process to having a DES problem in hand, covering steps 1–4 listed above. Chapter 4 closes the loop by generating and inserting code to act as the supervisor. The presented process is programming language independent, and all algorithms are presented in pseudocode. It is expected that any implementation of this process would be specific to a programming language supporting concurrency (e.g., Java, C++, etc.).

3.1 Introduction to the Running Example

Throughout this chapter and the next, each step of the process will be described in detail. A running example is provided both to illustrate concepts and to provide continuity. However, demonstration of the process requires that a concrete programming language be used. Out of many possible choices, the Java programming language has been selected. Like many other suitable languages, it provides multi-threading and necessary concurrency controls.

A precedence problem cast in Java is addressed using the process described herein. The relative simplicity of the problem allows readers to focus on the process being

introduced rather than dwelling on the intricacies of the example. The problem is comprised of five threads governed by four specifications. While all threads start together, `thread-2`, `thread-3,` and `thread-4` (`T2`, `T3`, and `T4`, respectively) must wait for `thread-1` (`T1`) to finish before executing their code. Additionally, `T4` must wait for `thread-5` (`T5`) to finish. This is illustrated in the precedence graph in Fig. 3.2.

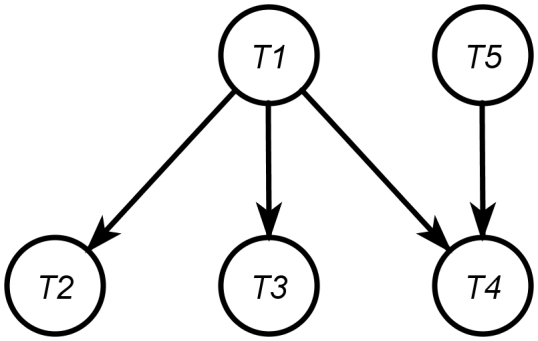


Figure 3.2: Precedence graph for the running example.

Code for the running example is equally straightforward, and is given in Fig. 3.3 on the next page. All five threads are identical, save for the identifier. Each thread is essentially a shell, with behaviour limited to starting, waiting to simulate working, and then terminating.

3.2 Creating the Event Set

To model code using DES, we must isolate events in the code that are both discrete and instantaneous. For this, we look to statements. In programming languages, statements are the building blocks that make up the program. Examples include assignments, conditionals (e.g. `if` statements), and `while` and `for` loops. The precise definition of a statement is highly language-dependent and typically defined in the grammar for the language in question.

```
package precedenceExample;

public class PrecedenceThread3 extends Thread {

  private int id;

  public PrecedenceThread3(int id) {
    this.id = id;
  }

  public void doWork() {
    int sleepTime = (int)(Math.random() * 1000);
    try {
      Thread.sleep(sleepTime);
    }
    catch(Exception e){
      //do nothing
    }
  }

  public void run() {
    System.out.println(id);
    doWork();
  }
}
```

Figure 3.3: Source code for `T3` in the precedence example.

We define a *software event* as the instant between the completion of execution of one program statement and the start of execution of the next. This is instantaneous (by definition), and is also discrete, since the completion of execution of a statement acts as a clear demarcation point. Using this definition, let the set of all software events for a given piece of code be defined as Σ_S. This definition is powerful, in that any transition between two statements is captured as an event.

Some software events are relevant to concurrency control while most others are not. An event that immediately preceeds a critical section, for example, would be relevant, as entry into a critical section affects concurrency control. Leaving a critical section would also be relevant for the same reason. However, most events, such as those between local variable assignments, branching statements, method definitions, math functions, and so on, are all irrelevant, insofar as no specification addresses those events. We define a set of *relevant events*, Σ_R, as the set of events necessary to enforce desired concurrent behaviour. The set of *irrelevant events*, Σ_I, contains all software events that are not relevant and is given by $\Sigma_I = \Sigma_S - \Sigma_R$.

Relevant events are noted in the code as comments in the form `//event marking: eventName`. These markings act as flags for subsequent operations to transform code into a DES model. As well, these events are used in the formalization of specifications. In the final stage of the process, event markings act as targets for the insertion of generated concurrency control code. While the idea of using comments as markup in code is a common one, this is the first time that markup has been used in this manner. This is a direct consequence of the novel definition of software events.

To build the set of relevant events, the informal specifications must be examined. The specifications describe the desired concurrent behaviour, and as such, embody a

description of what events are to be controlled, and how they are to be controlled. By definition, a relevant event is one of these events to be controlled. Developers of concurrent code would typically discern these events and then insert their own control code. Instead, this process merely asks the developer to note these locations in the code using relevant event markings. Given that, it is a reasonable expectation that an average developer will be able to mark relevant events from specifications. Additionally, the subsequent process of specification formalization gives the developer a second chance in case a relevant event was missed.

Precedence Example: The precedence problem presents a set of specifications from which the set of relevant events must be extracted. Threads `T2`, `T3`, and `T4` all have conditions placed upon starting. Thus, control needs to be placed at the very beginning of each of their run methods to enforce these conditions. A relevant event is marked at the first line of code for each of those threads. This is shown for `T3` in Fig. 3.4 on the following page. Threads `T2` and `T4` are identical in structure and result. The same specifications tell us that the end of `T1` is relevant and should have an event marking after the last line of code in the run method. Similarly, the end of `T5` is also marked as a relevant event. In total, there are five relevant events: `T1-finish`, `T2-start`, `T3-start`, `T4-start`, and `T5-finish`. The specification stating that `T4` must not start until `T5` finishes introduces the events `T4-start` and `T5-finish`, which are already covered. All other software events are irrelevant events and are left unnamed at this point.

```
public void run() {
  //relevant event: T3-start
  System.out.println(id);
  doWork();
}
```

Figure 3.4: Java code for `thread-3` showing the relevant event `T3-start`.

3.2.1 Controllable and Uncontrollable Events

In a DES plant, each event must be modelled as either controllable, meaning its occurence can be prevented, or uncontrollable, meaning its occurence cannot be prevented. In software, however, control will be enacted through the addition of code. At every point in the software, it is possible to insert control code, thus making the concept of uncontrollability unclear. An event that can be indefinitely prevented is not truly uncontrollable.

Why then, should the concept of uncontrollable events be maintained instead of eliminated? The primary reason is that both the control policy and generated code will be more concise. Fewer controllable events mean fewer possible control actions and thus less code. In addition, marking an event as uncontrollable is a shorthand method to specify that that event must never be prevented. In practice, events that are uncontrollable are denoted as such by appending '`-u`' to the event label.

The following guidelines describe situations when it may be helpful to specify an event as uncontrollabile:

- It is an irrelevant event.
- The event occurence is predicated by the actions of an external entity.
- It should never be prevented from occuring.

Irrelevant events can be classified as uncontrollable because, by definition, they do not appear in any specification. There is nothing to be gained, with respect to enforcing a specification, by delaying or disabling irrelevant events. As such, it should always be safe to classify an irrelevant event as uncontrollable.

An event occurence predicated by the actions of an external entity can be classified as uncontrollable because it is not possible for an external entity to be controlled. As an example, think of a network communication problem, wherein a separate thread is created to act as a network listener, and received messages are put into shared memory to be processed by other threads. The transition between listening for a message and receiving a message can be modelled as uncontrollable, to represent the fact that the actions of the remote program are outside the control of the current program.

Most uncontrollable events will be uncontrollable because they should not be prevented. This is largely at the discretion of the developer. Entry into a critical section is usually controllable, while exiting a critical section could be modelled as uncontrollable. This would ensure that a thread is never prevented from exiting a critical section, and helps to maintain availability of resources accessed in the critical section. To be clear, this would serve the same purpose as specifying that "It is always the case that critical sections can be exited without delay". However, simplying tagging the event as uncontrollable is a more elegant solution, as a specification would have to be formalized, while uncontrollable events require no further action. Additionally, code to be inserted at the end of the overall process is smaller and more efficient for uncontrollable events. Similarly, any event that is controllable but never becomes disabled can be marked as uncontrollable. Unfortunately, this information is

not known at this stage of the process and so events are rarely marked uncontrollable for this reason.

Precedence Example: In this example, it is clear that `T2-start`, `T3-start`, and `T4-start` must be controllable. `T1-start` and `T5-start` could be classified as uncontrollable, as intuition suggests that these events never need to be disabled. In this solution, they were left as controllable events.

3.3 Building the DES Model

The plant, as a model of the real unconstrained system, must generate exactly the events that are produced by the system, and produce these events in an ordering identical to that found in the real system. However, we are only concerned with the aspects of the code that are relevant to concurrency control. The behaviour of the code need not be duplicated in the abstract model. Thus, the model must merely generate the same relevant events as the code, and in the same ordering. Since we are using the Ramadge and Wonham framework, the plant will be represented as a DFA. For this process to be meaningful, the code must already be broken up into threads and ready to run as a concurrent program, minus the appropriate concurrency control code.

We accomplish this through the use of control-flow graphs (CFG). A CFG includes a node for each program statement, and transitions between nodes as dictated by the branching structure of the code. A generated CFG will thus provide a model that gives software events in the ordering dictated by the real system. Additionally, CFGs can be represented as DFAs, since there is a finite number of program locations, and thus a finite number of possible transitions (from each node to each other node, worst

```
x = 5;
//Event Marking E
foo(x);
```

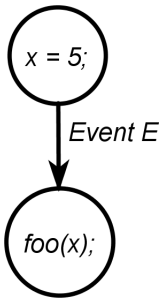


Figure 3.5: Relation between an event in the code and CFG.

case). Construction of CFGs can be automated using tools, such as Σοφια [39] for the Java programming language.

Relevant events in the CFG are identified using a simple mapping from software events to transitions in a control-flow graph. A software event refers to a transition between two statements, and thus there is a single originating statement and a single statement that is the target. In the CFG, each statement is represented by a node, with the edges in the CFG representing the possible transitions between statements. In fact, the edges in the graph are direct representations of software events. To find the associated software event/CFG pairing, simply locate the two statements that precede and follow the software event, then find that same transition in the CFG. Figure 3.5 shows a code snippet with a relevant event marking alongside the CFG for that snippet.

Unfortunately, there is the potential for the existence of infeasible paths in a CFG. Looking ahead, if a feasible path is disabled in response to the existence of an infeasible path in a CFG, it suggests a flaw in the process generating a model of the code. The claim of minimial restrictiveness must be tempered by only claiming minimal restrictiveness with respect to the model that is being constructed, and not

the actual code. As a way around this, a static analysis could be performed to determine path feasibility and improve the model. Regardless, by creating a control plan based upon the fullest possible branching structure, it is assured that the control plan will correctly enforce specifications for any behaviour of the plant.

Since we are dealing with concurrency, the behaviour of the program is determined by the behaviour of each thread, as they act independently of one another (before concurrency controls are added). Thus, each thread should be modelled as its own entity. In Java, the behaviour of a thread is determined by the contents of its `run` method. To capture the behaviour, the CFG for the `run` method of each thread is constructed.

Each CFG must then be transformed into a DFA. The entry node of the CFG is converted into the initial state of the DFA. All exit nodes become marked states. All edges in the CFG that correspond to relevant events are labelled with the relevant event name. All irrelevant events are given a label from the sequence `i1`, `i2`, `i3`,.... The labels should continue incrementing across each thread, that is, labels must not be reused across threads as this will interfere with plant generation. The irrelevant event labels applied to a CFG plus any relevant events appearing in that CFG form the event set for the DFA.

The above ideas lead to Algorithm 1, which is presented in Fig. 3.6. Algorithm 1 takes code annotated with relevant events and returns a reduced DFA version of the event-marked source code. The last step in Algorithm 1 is to reduce the resulting DFAs, which is described in the immediately following subsection.

Algorithm 1 transforms relevant event labeled source code into a DFA that can be used for DES operations.

Input: Event-marked code
Output: DFA representations of the code

For each thread, including the main thread:
1. Build the control-flow graph for the code executed by that thread.
2. Set the entry node as the initial state, and all exit nodes as marked states.
3. Label any edges that are also relevant events with the relevant event name.
4. Discard any CFGs that contain no relevant events.
5. Label all remaining unlabeled edges using i1, i2, i3, ... in increasing order. Do not repeat labels across threads - instead, continue incrementing.
6. Apply reductions.

Figure 3.6: Algorithm 1 transforms source code with event markings into a DFA.

3.3.1 Reducing a DFA

The number of states and transitions in a CFG is directly related to the program size and branching structure. However, much of the information captured in the CFG is unrelated to concurrency control. Each resulting CFG should be reduced through a structure-preserving transformation. For this process to be correct, the reduction must maintain all existing relevant events and preserve their ordering.

First, any CFG that contains no relevant events can be discarded outright. Such a CFG has no effect on the concurrent behaviours to be enforced, and can safely be ignored. Similarly, any method or function calls in a CFG can be left as unexpanded nodes in situations where no path through the call contains a relevant event. This includes paths that follow through nested method calls. If, however, a method call contains a relevant event, then the node containing that call must be expanded to

include the method. A node that starts a thread can always be left unexpanded, as the CFG for the new thread will be used to track the behaviour of that thread.

The main reduction proceeds by collapsing a branchless chain of (irrelevant event)-(node)-(irrelevant event) into a single irrelevant event, using the same label as the first irrelevant event. A branchless chain of (relevant event)-(node)-(irrelevant event) or a branchless chain of (irrelevant event)-(node)-(relevant event) both reduce to the single relevant event. If no path through a branch contains a relevant event, the branch and all paths through it may be reduced to a single node. Node labels are not used in the process and do not need to be maintained. At the end of this process, the event set for the DFA is updated to only include those events still remaining in the DFA.

Ideally, a DFA would be reduced such that no irrelevant events remain. However, branching behaviour of the code may leave some irrelevant events that cannot be removed. In Fig. 3.7, we see a code snippet on the left, the generated DFA model of the CFG in the middle, and the reduced DFA on the right. Only upon following the if-branch does the relevant event occur. When reducing this CFG, we must maintain that branching structure to ensure that event ordering is not affected. Event `i1` must remain a part of the CFG.

3.3.2 Marking Irrelevant Events

Much later in the process, code will need to be inserted at the location of every event, including any remaining irrelevant events. Event markings will act as the target for code insertion. Since irrelevant events only arise in the CFG, there are no event markings in the code for irrelevant events. Event markings must be made for every

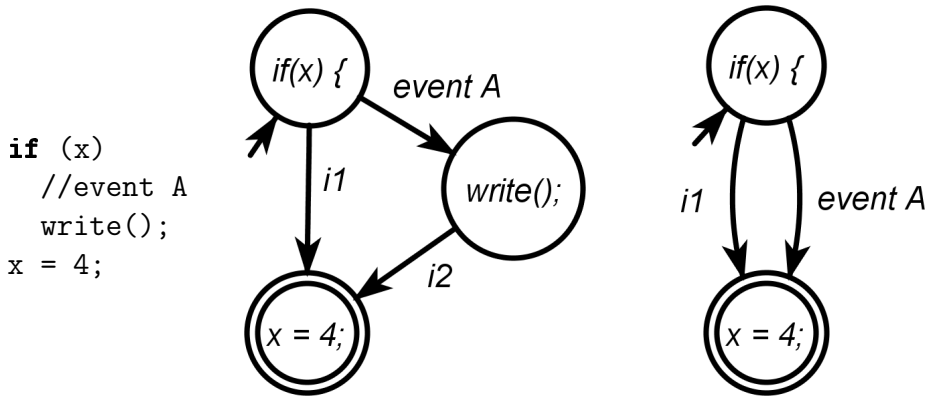
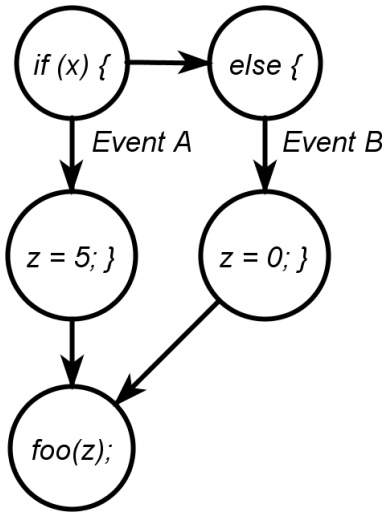


Figure 3.7: (From l. to r.) A code snippet, a DFA-converted CFG for that snippet, and the reduced DFA.

irrelevant event, using the original CFGs as a map to connect an irrelevant event to a location in code. It is most efficient to mark irrelevant events at this time, since the CFGs are still available. As noted earlier, all irrelevant events should be treated as uncontrollable, and should be marked with a '`-u`'.

The DFA reduction explicitly removes irrelevant events not in a branch. In other words, irrelevant events only remain in the reduced DFA when they are on branches. Also, since the reduction removes branching structures where no branches have a relevant event, it can be inferred that where there is a remaining irrelevant event, it must be in a branching structure where there is at least one branch with a relevant event and at least one branch with an irrelevant event. Additionally, any remaining relevant event is immediately preceeded by a branching statement because the reduction collapes irrelevant events upwards in the CFG. Under execution, an irrelevant event occurs if and only if a branch containing a relevant event is not taken. This mutually exclusive occurence must be maintained in the actual code when the irrelevant event is marked. This means that placing an irrelevant event must take more into account than a simple location; paths must also be considered.

Some irrelevant events lie on *explicit paths*, meaning there is at least one location in the actual code that is reached only when the branch containing that irrelevant event is followed. Figure 3.8 shows two events each on an explicit path. Event A only occurs when the if-branch is followed, and event B only occurs when the else branch is followed. The original (unreduced) CFG tells us that Event A should be marked between the `if-else` statement and the following assignment on the `if` branch, while event B should be marked between the `if-else` statement and the following assignment on the `else` branch. Each of these locations exist in the code, and lie on mutually exclusive paths.

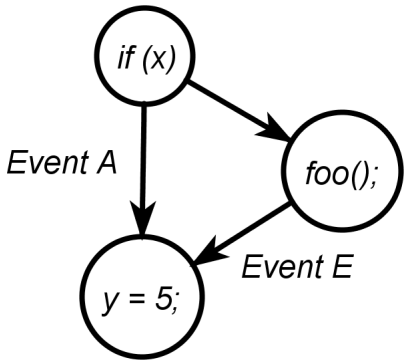



```
if (x) {
  //event A
  z = 5;
}
else {
  //event B
  z = 0;
}
foo(z);
```

Figure 3.8: Two explicit paths in the CFG and the resulting event markings.

Paths through the software are not always explicit. An *implicit path* is a path through a branching structure that has no location in the code unique only to that branch. In Fig. 3.9, event A occurs when the if-branch is not taken. There is no place in the code, however, that corresponds to the if-branch not being followed. If event A is located immediately after the `if` statement, or immediately before the assignment statement for y, then event A will occur when the path containing event E is followed.

This violates our mutual exclusion requirement for paths.




```
if (x)
  foo();
  //Event E
y = 5;
```

Figure 3.9: Event A lies on an implicit path and cannot be marked.

The task now is to place code at an event location that does not exist—an impossible task. The solution is avoiding the problem by modifying the source code to transform problematic implicit paths into explicit paths. At first glance, this may seem to be a deficiency in the process. The source code is being altered, so it could be suggested that the process only works on source code that does not contain implicit paths. However, no new paths through the software are being introduced, and no new functionalities are being created. The only difference is that some preexisting implicit paths now have explicit locations. In essence, this transformation ensures that inserted concurrency control code will be on the correct path, and thus code introduced as part of this transformation is best thought of as part of the inserted concurrency control code.

Every programming language has a variety of branching structures. Some structures have the potential to introduce implicit paths while others do not. The same branching structure may have different properties across different languages. A full exploration of the implicit path problem is thus infeasible due to the variety of programming languages and branching structures. However, one of two basic approaches

can be applied to render any implicit path explicit.

The first approach works on those branching structures that have an optional catch-all branch. For example, many languages allow an `if` statement to exist on its own or as part of an `if-else` statement. We can infer that if there is an implicit path, the optional catch-all `else` portion of the statement must have been omitted, since the `else` branch would have caught the irrelevant event if it existed. This strongly suggests the solution, which is to introduce the optional catch-all and mark the irrelevant event in the catch-all branch. This does not introduce a new path; it only makes explicit the path followed when no branch in the existing branching structure is followed. Some candidates for this approach include `if` statements without an explicit `else`, and `case-switch` statements without a default `switch`.

The second approach is to introduce a boolean variable used to track the path followed. This may be applied to branching structures without a catch-all branch. For example, a `while` loop has an entry condition that must be met to enter the loop; otherwise, the loop is skipped. The path followed when the loop is skipped is an implicit path. Before the `while` statement, a tracking boolean is set to true, and toggled false if the `while` loop is entered. At the join point for the implicit path around the loop and the explicit path through the loop, the tracker is checked using an introduced `if` statement. If the tracker is still true, the implicit path was followed and the inside of the `if` statement is reached. This location is thus the explicit location for the implicit path, and the irrelevant event should be marked here. Structures that can be addressed with this approach include `while` loops and `for` loops.

3.3.3 Constructing the Plant

The plant is given by some combination of all the resulting reduced DFAs. A typical DES approach would simply take the synchronous product of all DFAs. That would force shared events to happen in unison, and would interleave any unshared events. However, this standard approach presents a problem. In our software model, shared events can occur when multiple threads call a shared method or function containing a relevant event. Multiple threads would then cause an occurence of the same software event. Modelling this by using the synchronous product does not capture this behaviour, since the event would only occur once, and in unison, for all threads.

To accurately capture the real behaviour, we propose a method to 'unshare' all shared events. Each occurence of a shared event is renamed by adding an identifier for the calling thread. For example, an event `read` that is shared by `thread-1` and `thread-2` could be renamed `read-t1` in the DFA for `thread-1` and `read-t2` for `thread-2`. At the end of this renaming, there are no longer any shared events. It is now safe to use the synchronous product operation to combine the DFAs. With no shared events, the combination is simply the shuffle, and thus models the true interleaving behaviour of the actual system.

Combining the threads using the shuffle operation presents a second problem. Shuffle combines the initial states of the threads into one initial state. This is only correct when it is assumed that all threads are in their initial state before any thread acts. If a thread is dynamically created by another thread only after a relevant event occurrence, this assumption would be violated, since that shared initial state would not exist in the real system. Research on dynamic DES theory [17] proposes a DES model wherein event-generating modules can appear and disappear during

execution. This is uninvestigated for our work, but seems to offer an alternative to the assumption that all threads begin at the same time. In the meantime, this work proceeds assuming that all programs under consideration start such that all threads are in their initial state at the same time.

Precedence Example: Each thread was processed using Algorithm 1. The threads in the running example generated simple straight-line stuctures due to the lack of branching behaviour. The lack of branches makes it possible for the reduction step to remove all irrelevant events. Each thread was reduced to only two nodes connected by a single relevant event. Fig. 3.10 shows the results on `T3`. Note that the DFA for T3 shows irrelevant events `i5` and `i6`. This is due to `i1` and `i2` being used in `T1`, and `i3` and `i4` being used in `T2`. The other threads gave identical results

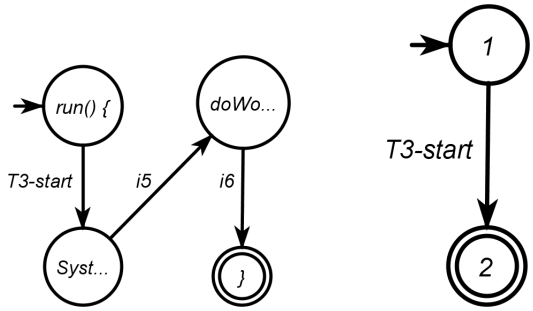


Figure 3.10: DFA-converted CFG and reduced DFA for `T3`.

Next, the plant is generated. Dynamic thread creation does not occur, so the assumption regarding the initial state is not violated. There are no shared events, so renaming events is unneccessary, and the threads can be combined using shuffle. The resulting plant for the precedence example contains 32 states and 80 transitions, representing all possible interleavings of the five relevant events across the five threads. The number of states in the plant shows exponential growth resulting from the shuffle operation. The plant is not shown as a figure due to the difficulty of visually

interpreting large DFAs.

3.4 Formalizing Specifications

The DES plant made in the previous section generates all possible event sequences of the system. A specification describes some subset of those event sequences that are to be allowed. Specifications must be each given as a DFA since we are using DFA-based DES theory. This limits our process to enforcing safety properties only, and precludes liveness properties. In the previous step, relevant events were extracted from each specification. These are now used to formalize specifications. For each specification, take the associated relevant events, and then use them as the event set for a DFA that generates those events in the ordering demanded by the specification.

As noted in Section 2.1.2, modular DES theory allows for the usage of multiple specifications. Only behaviours permitted by all specifications will be allowed in the controlled plant. It is vital that each specification does not inadvertently restrict other behaviours. All events possible in the plant should be permitted in every specification except where an event must be explicitly restricted by the specification. In a DFA, this is accomplished through the addition of a self-loop at every state for each event in the plant that is not a relevant event arising from that specification. This includes any irreducible irrelevant events. At the end of this process, the event set for each specification will be the same as the event set for the plant.

Precedence Example: Our running example has two safety properties to enforce. Threads that must wait on `T1` to finish are restricted by the specification in Fig. 3.11, and the thread waiting on `T5` is restricted by the specification in Fig. 3.12. Note the self-loops for all events not arising in the specification. This results in

specifications that permit all possible event occurences except those restricted by the specification in question.

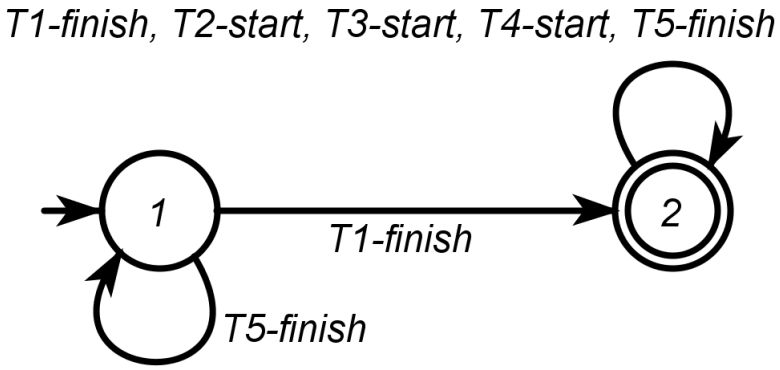


Figure 3.11: Specifying that `T2`, `T3`, and `T4` must wait for `T1` to finish.

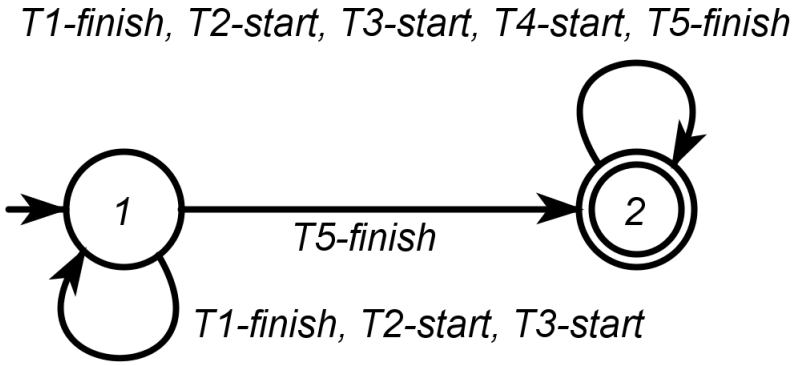


Figure 3.12: Specifying `T4` must wait for `T5` to finish.

3.5 Constructing the Supervisor

With the plant and a set of modular specifications we can proceed to synthesize a supervisor. Since we are using modular specifications, the possibility exists that these specifications are not non-conflicting and therefore the closed-loop system may deadlock. This can always be avoided by constructing the monolithic specification and using it to synthesize the monolithic supervisor. Unfortunately, this approach comes

at the cost of an exponential increase in the number of supervisor states. Since a test for non-conflicting can be easily performed using DES tools, a typical approach would be to test for conflicts, then employ the modular or monolithic supervisor accordingly. In our implementation, we choose to always build the monolithic specification and proceed with the monolithic supervisor. Though not always the most efficient, this approach guarantees a correct result. The supervisor construction algorithm in [38] performs this task in polynomial time for a single specification.

As noted in the introduction, two software packages were used to perform DFA operations: IDES [20] developed at Queen's University, and TCT [41] developed at the University of Toronto.

Precedence Example: The monolithic specification was produced in IDES by finding the product of the specifications in Figs.3.11 and 3.12, and is shown in Fig. 3.13. It should be expected that initially only `T1` and `T5` can act, and it is clear that this is the case. Only after both threads have acted is `T4` finally allowed to act. The monolithic specification, along with the plant from Section 3.4, was exported to TCT, where the supervisor construction operation (`SUPCON`)was performed. The resulting supervisor has 14 states and 23 transitions, and is shown in Fig. 3.14 on page 43.

In general, there is no requirement that a specification contain only paths that are possible in the plant. This differs from a supervisor as they must only contain paths that are possible in the plant. Since, in this example, the specification is controllable with respect to the plant, the largest controllable sublanguage is just the monolithic specification. Following from this, the supervisor is just the intersection of the specification and the plant. If the specification had been uncontrollable, the supervisor

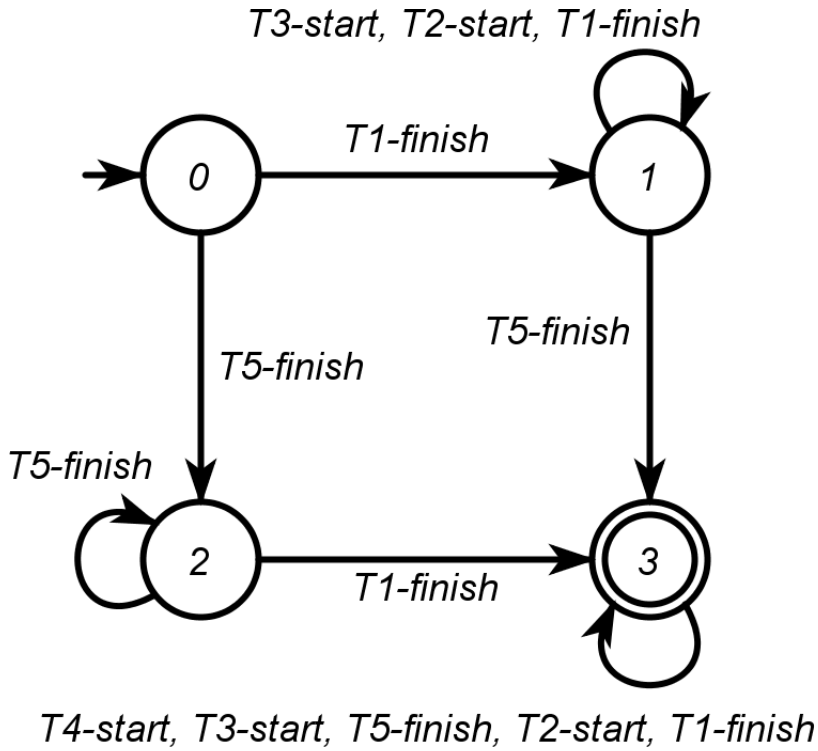


Figure 3.13: The monolithic specification for the precedence example.

would have only permitted the largest controllable subset of the specification.

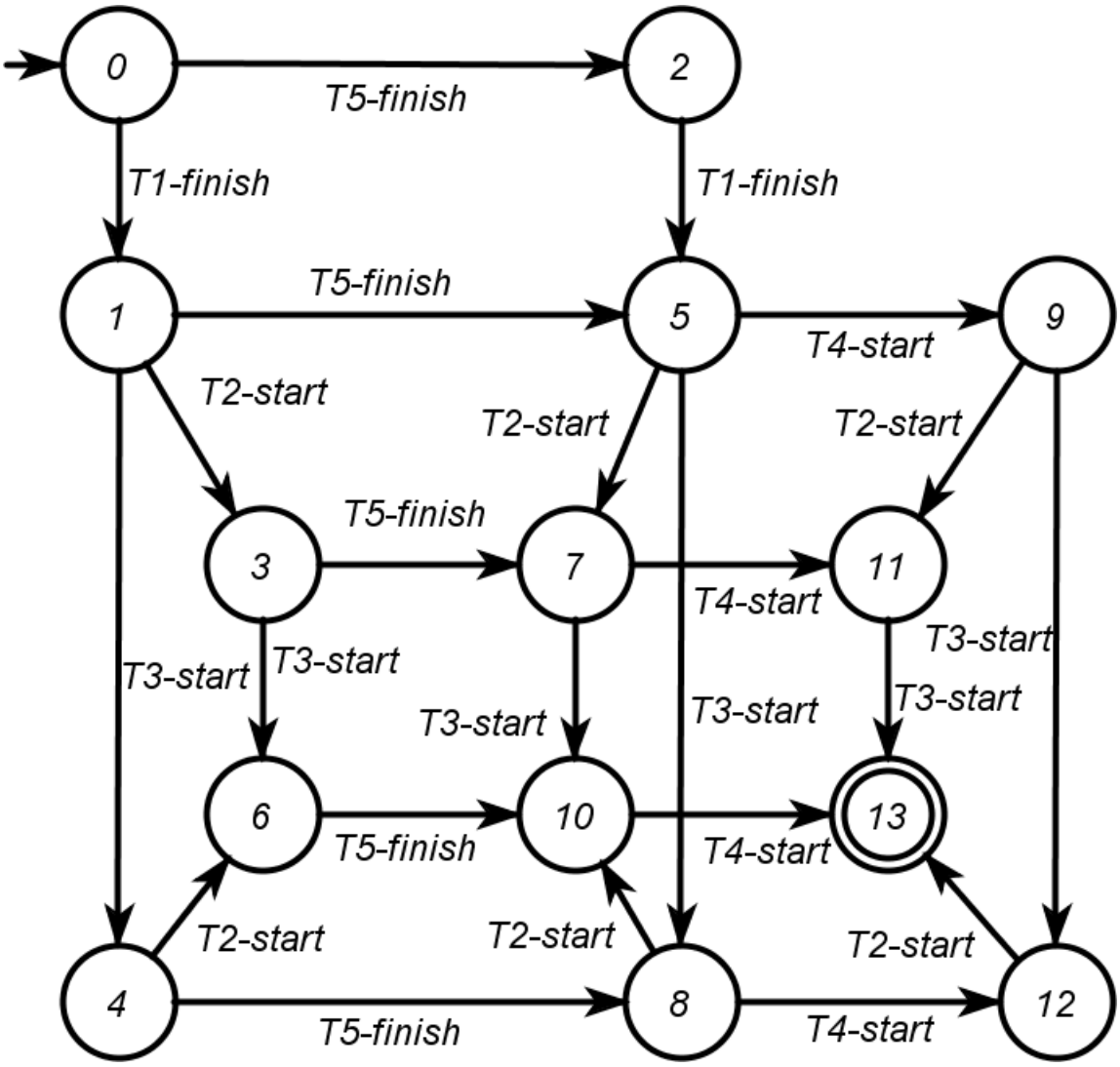


Figure 3.14: The monolithic supervisor for the precedence example.

Chapter 4

Supervisor Realization

At the end of the previous chapter, the described process had created a supervisor, guaranteed when coupled with the plant to result in a closed-loop system that is non-blocking while enforcing the largest controllable sublanguage of the specifications. The focus now shifts. This supervisor only exists as an abstract mathematical construct; the supervisor must now be realized. Practically, this means that the abstract supervisor must be transformed into code—code that can be woven into the original source, and thus couple a realized supervisor with the actual plant. In other words, the generated concurrency control code will be the supervisor realized into code. The work presented in this chapter details how this is accomplished.

The problem is addressed by working from our model of supervisory control shown in Fig. 2.1 on page 11. Several aspects are explored, revealing guiding principles that culminate in code blocks that, when taken together, fully implement supervisory control. Next, an algorithm is presented that takes an existing supervisor and event-marked code as input and automatically generates supervisory control code.

Like the work in Chapter 3, these principles are applicable across programming languages, and code is presented as pseudocode. Since realizing a supervisor as code necessarily involves a programming language, an implementation is necessary to gain a full understanding. The algorithm is implemented and described in detail for the Java programming language, showing clearly that the process can be successfully implemented. The precedence example, introduced in Section 3.1, will be further addressed by generating code to realize the already generated supervisor. Resulting code is shown to be correct through model checking, a process that will be explained at the end of the chapter.

All code (and its associated behaviour) in this chapter is novel, and is developed based upon the reasoning presented throughout this chapter. To the best of the knowledge of the author, supervisory control of this sort has never been implemented with the purpose of controlling a concurrent software program.

4.1 Implementing Supervisory Control

Each step of the supervisory control process is examined in turn, and code is developed that implements each function. The basic supervisory control model involves only two steps. First, the plant generates an event that must be observed by the supervisor[1]. In response, the supervisor issues control actions by enabling and disabling events as per its control policy. The process repeats for each event generated by the plant. A realized supervisor must contain code that performs these functionalities. Once the

[1]This requirement was relaxed in [29], and subsequent research such as [8]. Called "partial observation", the theory fits in as an expansion to the core Ramadge and Wonham framework, and is not directly addressed in this work. Instead, it is noted in future work as a possible avenue for future exploration.

code is generated, it is combined with the original source code to create the controlled system.

4.1.1 Controlling Events

During code execution, a statement is executed as one or more statements at the assembly level, with the program counter incrementing between each assembly statement. Each statement at the programming level is represented by a block of statements at the assembly level. Upon each increment that moves the program counter from a block implementing one higher-level statement to another block, the program generates the intervening abstract software event. Since this process continues unhindered without concurrency controls, we conclude that the default condition of a software event is enabled. As described in Chapter 3, only a subset of these events are relevant to concurrency control, and these are the events that appear in the supervisor. The set of software events to manage is further limited to only those that are modelled as controllable. Out of all software events in the program, only those that appear in the supervisor as controllable events require concurrency control code. A mechanism must be generated to provide the ability to enable or disable a controllable event.

A semaphore provides the capabilities necessary to disable an event, and is nearly universal in modern concurrent programming languages. Forcing a thread to wait on a semaphore halts the actions of that thread, indefinitely preventing the event occurrence, thereby disabling the impending event. Signalling that semaphore at a later time will wake the thread, enabling the event. Each controllable event will be assigned an exclusive semaphore, the behaviour of which will be managed by the

supervisor. If an event is disabled, the semaphore should have no permits, and if the event is enabled, it should possess exactly one permit. Thus, the semaphore is a binary semaphore. Changing the number of permits in a semaphore is the responsibility of the supervisor, as it determines if the event is disabled or enable. However, threads must check the semaphore using the `Wait` command to determine the current control state, and this removes the permit from the semaphore if it has one. To negate this, a permit should be immediately returned to the semaphore with a `Signal` instruction. This restores the permit that was taken by the `Wait`, thus returning the semaphore to its correct state. Failure to replace this permit would mean that the current control action has changed without a supervisory control action being issued. This would violate the model of supervisory control and lead to unpredictable consequences, most likely taking the form of deadlocks.

Every controllable event is assigned a dedicated binary semaphore; uncontrollable events do not require association with a semaphore as they never become disabled. If a thread is waiting on an event semaphore, and then awakes, there is still no guarantee that the event can actually occur. A context switch could occur causing that event to be disabled before the thread acts. Passing the semaphore is no guarantee that the thread is enabled. Thus, when the thread does eventually act, it is necessary that it again checks the associated semaphore. This implies that a looping mechanism will need to be created to recheck the semaphore. This loop is only passable when the event is first checked and found to be enabled.

When each semaphore is created, it should be initialized so that it matches the control state of the associated event in the initial state of the supervisor. If the event is enabled, the semaphore should be initialized with one permit; if disabled, it should

have no permits. By this strategy, the semaphores initally enact the control action in the initial state of the supervisor.

4.1.2 Observing Events

All events in the supervisor must be observed when they occur in the plant—both controllable and uncontrollable. When the plant generates an event, the supervisor transitions through its own internal structure based on that same event. Control actions are then issued based upon the new state. If the supervisor does not observe an event and fails to make the matching transition, synchronization with the plant would be lost. The control policy could no longer be enforced, and the system would become unpredictable. Synchronization must be maintained through observation.

Observation can be accomplished by inserting code at each event to notify the supervisor of the event occurrence. Additionally, the supervisor must respond to the change in state by updating its own state, and issuing a control action if necessary. However, a context switch at this point would be problematic. Consider the following chain of events. A thread `thread-A` executes an event `event-A`. The supervisor observes the action and in response disables the currently enabled event (`event-B`) in a second thread (`thread-B`). However, before the supervisor can disable `event-B`, a context switch occurs and `thread-B` resumes execution. The actions of `thread-B` result in an occurrence of `event-B` because the supervisor had not yet disabled the event. This chain of events violates the control scheme in the supervisor, and shows how a context switch between observation and supervisor reaction could cause a loss of synchronization. Thus, it is necessary to reserve time for the supervisor to update and react. During this time, event occurrences must be queued until the supervisor

has updated its own state and effected a control action.

4.1.3 Event Code

The aforementioned observations lead to the first description of actual generated code. It is clear that actions must be taken at an event location, but what exact location is implied? Recall that software events are abstract concepts, defined to occur as execution passes from one statement to the next. The event location is *between* the statement immediately preceding the software event and the statement immediately following it. Thus, saying that code should be placed at an event actually means that code should be placed between the statements couching the event, as shown in Fig. 4.1. Note that this approach only works when events lie on explicit paths, explaining the necessity of the transformation in Section 3.3.2 that makes implicit paths explicit.

```
Preceding statement                    Preceding statement
//event marking: event1      becomes   //event marking: event1
Subsequent statement                   inserted concurrency control code
                                       //event1 occurs
                                       Subsequent statement
```

Figure 4.1: The effect on event location when code is inserted.

The concepts above lead to code that implements the supervisor functions necessary at the event location. At every controllable event, code implementing Fig. 4.2 should be inserted. The code containing the semaphore check is trapped in a `while` loop. If allowed to proceed, the thread breaks out of the loop and is allowed to proceed, thus executing the event. If the event is disabled, then the thread must wait on the associated semaphore until signalled. Immediately the semaphore permit is

replaced, thus maintaining the enforced control action. Upon awakening, the `while` loop returns execution to the start of this block, and the supervisor is once again notified that the thread is trying to execute an event. At an uncontrollable event, the event is always enabled, so the code from Fig. 4.2 reduces to mere supervisor notification. This is given in Fig. 4.3. The method `observeAndReact`, explained in the following section, notifies the supervisor of the event occurrence and gives reserved time for the supervisor to act.

```
//event marking: eventName
while (true) {
  if(observeAndReact("eventName",eventNameSemaphore))
    //the event is enabled and may occur, exit the while loop
    break from while;
  //event is disabled, force thread to wait
  Wait on eventNameSemaphore;
  //semaphore has been signalled - replace the semaphore permit
  Signal eventNameSemaphore;
}
//event occurs
```

Figure 4.2: Code to be placed at each controllable event.

```
//event marking: eventName-u
observeAndReact("eventName-u",null);
//no semaphore - the event is always enabled
//event occurs
```

Figure 4.3: Code to be placed at each uncontrollable event.

4.2 Supervisor Observation and Reaction

Upon observing an event, the supervisor reacts by issuing a control action if necessary. This process must not be interrupted by other event occurrences, in the interest of avoiding race conditions. Similarly, event occurrence and event observation must also occur as an atomic group, once again to avoid race conditions. In fact, all three operations must occur as an atomic group. In other words, as an event occurs, other events must be disabled while the supervisor observes and reacts. First, the event semaphore must be checked to ensure that the event is enabled. Secondly, if the event occurs, the supervisor must be notified. Third, the supervisor must update its state and issue a new control action if necessary. These three actions must occur as an atomic group for each event, and that group can only execute in mutual exclusion with other groups. In this manner, no race conditions are introduced.

Unfortunately, the fact that the control state of an event (i.e. if it is enabled or disabled) is managed through a semaphore means that a thread would be forced to wait if the semaphore is checked using `Wait` and the event is disabled. Since the semaphore check is enclosed in a mutually exclusive block, no other thread would be able to execute an event if a thread became stuck in the mutually exclusive block due to a disabled event. Deadlock would ensue. In an implementation, this must be avoided, possibly through a flag maintained by the supervisor indicating the control state of an event. Some programming languages provide advanced semaphore methods that allow a semaphore to be checked without the possibility of a wait[2]. Regardless, if the event is disabled, the thread must eventually be forced to wait. This command takes place outside of the mutually exclusive group in the event code

[2]Java is one of these languages. Later in this chapter, this method of determining control state will be implemented.

given in Fig. 4.2 on page 50. If the event is enabled, then the supervisor should check its transition table, update its event, and issue a control action if necessary.

This leads to the code presented in Fig. 4.4. The `observeAndReact` method is called at every event occurrence (as seen in Figs. 4.2 and 4.3), and as input takes the name of the occurring event and a pointer to the semaphore associated with that event (or `null`, if the event is uncontrollable). This input gives the supervisor the information needed to update the supervisor state and issue a control action if necessary. If the event is controllable, the supervisor determines the control state of the event. If the event is disabled, the method returns `false`. This triggers the event code to wait on the semaphore. If the event is enabled, the supervisor calls `updateSupervisorState()` to update the supervisory state and issue a new control action. Upon seeing a `null` semaphore, the event is uncontrollable and the supervisor proceeds to the `updateSupervisorState()` method.

This code must only be accessed in mutual exclusion with itself. Only one thread can execute an event at any time, and all other threads must wait for the `observeAndReact` code to be completed before being able to access it.

4.2.1 Determining Control Actions

While the control policy dictates the events that are to be enabled and disabled in each supervisory state, it does not include information on transitions. Thus, at every supervisor transition, there needs to be a full review of the control state for each event, and then changes would need to be made for each event that became enabled or disabled. Since the control policy is unchanging, greater efficiency can be achieved by determining the changes in control action across each transition prior to run-time,

```
observeAndReact: Notifies the supervisor that an event
is about to occur.  Determines if the event can occur, and if so,
updates supervisor state and control action. Must be accessed in
mutual exclusion.

Input: name of event that is about to occur, semaphore for that
       event (or null if uncontrollable)
Output: A boolean variable that is true if the event can occur,
       and false if the event is disabled.

observeAndReact(String eventName,Semaphore sem) {
  if event is controllable {
    if event is disabled {
      //event cannot occur
      return false }
  }
  //event can occur
  updateSupervisorState(eventName);
  return true;
}
```

Figure 4.4: Code for `observeAndReact`.

and then hard-coding this in the Supervisor class as a table look-up.

We define a new construct, which we will call the *change map*, to track changes in the control action based upon transitions. The change map is constructed as follows. Recall that transitions in the supervisor are defined by the transition function δ, and each transition is given by $\delta(q, \sigma) = q'$, where q is the source state, q' is the target state, and σ is the event on the transition. For each transition in the supervisor, we compare the set of disabled events at the source q and target q'. We define two similar functions, the *disabling function*, Δ_D, and the *enabling function*, Δ_E as

$$\Delta_D(q, \sigma') = \{\sigma \in \Sigma \mid \sigma \text{ is enabled in } q \text{ and is disabled in } \delta(q, \sigma')\}$$

$$\Delta_E(q, \sigma') = \{\sigma \in \Sigma \mid \sigma \text{ is disabled in } q \text{ and is enabled in } \delta(q, \sigma')\}$$

Together, these two functions form the change map. For any transition $\delta(q, \sigma)$, the change map can be consulted to determine exactly which events must be enabled and which must be disabled. In practice, the source state for the transition is always the current supervisory state, and thus only the event σ would need to be observed by the supervisor before consulting the change map.

In Fig. 4.5, the given sample code provides a model to implement this supervisor functionality. Essentially, the code searches through the list of events to find the information for the event that is occurring. Then, it searches through the list of possible transitions for that event to find the transition for the current supervisor state. In the presented pseudocode, it is shown as a brute force search by going through all options using `for` loops. An implementation could use any one of a variety of search methods and branching structures. Once the transition is isolated, the supervisor state is updated. Finally, a series of statements enable and disable events as instructed by the change map by adding or removing the permit from the associated semaphores. These statements are responsible for actually issuing the control action.

4.2.2 Comments on Correctness

The code introduced in this chapter combines to form an implementation of supervisory control. Consider the initial state s of the program. Semaphores are initialized such that the control action at this state matches the control action from the supervisor's initial state. A subsequent transition to a child program state s' causes an update to the control action as informed by the change map, which is built directly from the

```
updateSupervisorState: Updates the current state of the
supervisor and issues control actions.

Input: Name of pending enabled event.
Output: None, updates state and issues control actions inline.

updateSupervisorState(event e){
  enable all events in Δ_E(supervisorState, e):
    add a permit to the semaphores for these events
  disable all events in Δ_D(supervisorState, e):
    remove the permit from the semaphores for these events)
  supervisorState = δ(supervisorState, e)
}
```

Figure 4.5: Pseudocode showing the updateSupervisorState method.

control policy in the correct-by-design supervisor. Any accesses to the semaphores explicitly ensure that the semaphore is left unchanged, meaning that the supervisor and only the supervisor changes the current control action. By this reasoning, it is claimed that s' also enforces the correct control action. This same logic can be applied from s' to another state s'', and so on, eventually reaching all states since the supervisor is reachable. This shows that the correct control action is applied at every state. Thus, the supervisor code starts in the correct state, safely transitions to new states, and safely issues control actions as informed by the change map. This strongly suggests that the code above embodies a viable method to implement a supervisor.

Like all abstract algorithms, this approach is only effective when implemented correctly. If concurrency bugs, such as a deadlock or a race condition, are introduced, then the implementation is clearly lacking. Claims of correctness are accordingly withheld until the algorithm is implemented and can be analyzed. The implementation must read the current state, determine if an event is allowed, and then update

the control action without introducing the possibility for deadlock or race conditions due to the supervisory control process. While attention has been paid to preventing the introduction of concurrency bugs during supervisor realization, the difficulty in spotting these bugs is one of the reasons why this entire process has been developed. The lack of concurrency bugs must be confirmed on an actual implementation, and is best done through model-checking.

4.3 Generating Code

The supervisor implementation developed above is at the core of the code generation process. However, the presented pseudocode cannot be completed without additional input, and thus must be implemented as skeletons. The goal of the algorithm is to use information from the input to flesh out the skeletons, and thereby generate concurrency control code in the form of a realized supervisor. For input, the algorithm will take the results of Chapter 3: event-marked code and a DES supervisor. Output will be the event-marked code with added concurrency controls that implement the supervisor.

The algorithm to generate concurrency control code is presented in Fig. 4.6 on page 58. The algorithm can be implemented in any language that supports file manipulation. However, an implementation must be made to work for the programming language used in the event-marked source, since it will generate new code to be inserted back into that event-marked source. The pseudocode samples developed above must be implemented in the same programming language as the source. For the purposes of this algorithm, it is assumed that that step has been completed, and that skeletons of those algorithms are available to be used as part of algorithm 2.

Step 3 deliberately lacks clarity, as each programming language has different methods of sharing data between threads. In Java and C++, the supervisor could be created as a static class with the data (methods/functions and variables) marked as public so that other threads can access them. Other programming languages share data using global variables or other methods. So long as all threads can access the data in the supervisor, the requirement in step 3 is satisfied.

4.4 Java Implementation

Implementing Algorithm 2 in the Java programming language first involves the implementation of all pseudocode skeletons in Java. Using those, a program implementing Algorithm 2 was written. This work was done in Java, though this is by no means necessary. Algorithm 2 was implemented in Java, with the exception of the final step of the algorithm, which was performed manually. Note that there are no development issues preventing the implementation of step 8. Regardless, since step 8 is not being performed, it is not necessary to include the event-marked code as input. A proper implementation would require this. This section details how the algorithm was implemented. The implementation details are largely irrelevant to understanding the overall project, and as such all details are remanded to Appendix A. Interested readers are referred there if they wish to examine the implementation.

The implementation can read a supervisor from an IDES model (`.xmd`) [20], compute the change map, and output code that realizes the supervisor. While the implemented portion does not automatically insert event code into the original source, there are no hurdles preventing this. The supervisor is implemented as a public static `Supervisor` class, and any variables or methods in the supervisor are accessed directly

Algorithm 2 generates concurrency control code by realizing a DES supervisor, then inserts it into the source code.

Input: Event-marked code, supervisor.
Output: Code with concurrency controls

1. Analyze the DES supervisor. Determine the event set including event controllability, and note all transitions. Determine the events enabled and disabled at each state.
2. Build the change map by cycling through all transitions and comparing the events enabled and disabled at the source and the target.
3. Create a supervisor in code such that data in the supervisor can be accessed by all threads.
4. For each controllable event, create a semaphore with a unique name in the code supervisor. Initialize each semaphore to 0 if the associated event is disabled in the initial state of the DES supervisor, or 1 if it is enabled.
5. Add code implementing observeAndReact to the supervisor.
6. Add code implementing updateSupervisorState to the supervisor, completing the code by using the information from the event names, transition structure, and change map.
7. For each event, complete the skeleton implementing the event code by inserting the event name and the associated semaphore name, using the controllable or uncontrollable event code as necessary.
8. Search through the source to find event markings. Insert the appropriate event code at the event markings.

Figure 4.6: Algorithm 2 creates concurrency control code through supervisor realization.

using `Supervisor.variable` or `Supervisor.methodName()`.

4.4.1 Supervisor Implementation

Each of the pseudocode skeletons were transformed into Java skeletons, ready for use by the implementation of Algorithm 2. Each of these skeletons is incomplete without information from the supervisor; in these cases, data to be supplied later is indicated using double angle brackets (e.g. `<<eventName>>`). Additionally, some of the code presented here has been developed with information from model-checking earlier versions of the resulting code, in which case it is noted.

First, the event code is examined. For controllable events, the Java code skeleton is given in Fig. 4.7, and for uncontrollable events, the Java code skeleton is given in Fig. 4.8. Both of these methods make calls to `observeAndReact` in the `Supervisor` class. In Fig. 4.9, a sequence diagram shows how these skeletons behave for an enabled event. In Fig. 4.10, the sequence diagram shows the behaviour for a disabled event. Basically, the event must wait until another thread executes an event that causes the supervisor to issue a control action enabling the disabled event.

```
while (true) {
  if (Supervisor.observeAndReact(<<"eventName">>,
        Supervisor.<<eventNameSem>>))
    break;
  Supervisor.<<eventNameSem>>.acquireUninterruptibly();
  Supervisor.<<eventNameSem>>.release();
}
```

Figure 4.7: Java code skeleton to be completed and inserted at controllable events.

In Java, the semaphore `Wait` command is implemented as the semaphore method

```
Supervisor.observeAndReact(<<"eventName-u">>, null);
```

Figure 4.8: Java code skeleton to be completed and inserted at uncontrollable events.

`acquire()`, and `Signal` becomes `release()`. However, the semaphore method `acquireUninterruptibly()` was used rather than `acquire()`. When a thread waits using the `acquire()` method, it is possible to force an early wake before being signalled via the `release()`. This would throw an exception, the possibility of which caused errors to be discovered in model checking. These errors are avoided by using `acquireUninterruptibly()` as it prevents interrupts.

The Java implementation for the `observeAndReact` method is given in Fig. 4.11. This is, in fact, a proper Java implementation and not a skeleton—the method does not require any information from the supervisor to be completed. To ensure mutually exclusive access, it is defined as `synchronized`, a Java keyword that is the equivalent of wrapping all calls to that method in a Mutex semaphore. The specialty Java method `Semaphore.tryAcquire()` is used to determine the control state of the event; if a permit is not available in the semaphore, execution simply proceeds, never blocking. This allows the thread to move out of the synchronized method, eventually blocking on the semaphore in the inserted event control code. If a permit is present, the `tryAcquire()` method will remove a permit from the semaphore and execution proceeds. As in the event code, this permit is immediately replaced, to ensure that the current control action is not altered.

Finally, the Java implementation of `updateSupervisorState` is given in Fig. 4.12. Unfortunately, this method is almost entirely based on the supervisor itself, so the Java skeleton is still largely pseudocode. To search through the supervisor for the

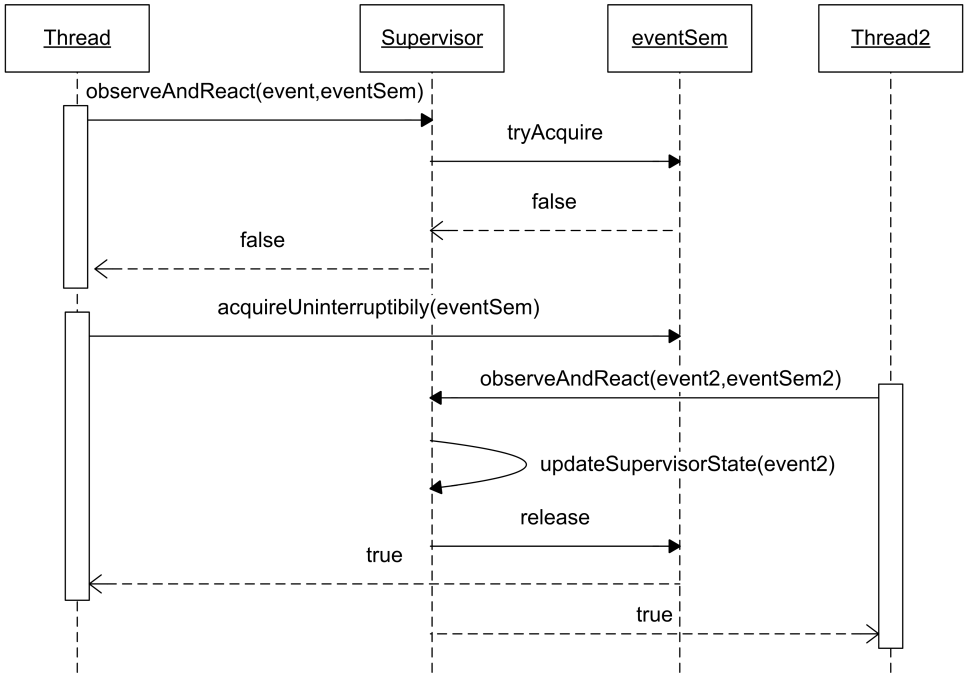


Figure 4.9: A sequence diagram of the interaction between thread and supervisor when an event is disabled.

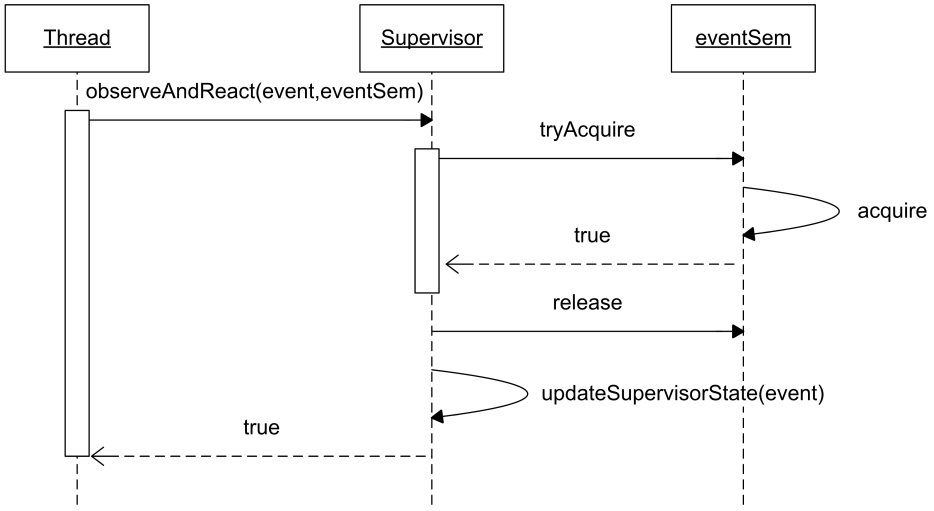


Figure 4.10: A sequence diagram of the interaction between thread and supervisor when an event is enabled.

```
public static synchronized boolean observeAndReact
                     (String event, Semaphore eventBlocker) {
  if (!(eventBlocker == null)) {
    if (!eventBlocker.tryAcquire())
      return false;
    eventBlocker.release();
  }
  updateSupervisorState(event);
  return true;
}
```

Figure 4.11: The Java implementation of `observeAndReact`.

transition that is occurring, there is a large `if-else` statement that contains a branch for each event. Using the current supervisor state, a `switch-case` statement finds the instructions for updating the supervisor state and issuing control actions. This manner of searching is by no means a unique method to isolate the occurring event, and no claim is made as to its optimality or efficiency, other than to say that this method is essentially brute-force and is far from optimal.

The `updateSupervisorState` method issues control actions by adding and removing permits from event semaphores. It is not possible in Java to directly add or remove a permit from a semaphore. Instead, the supervisor can only remove a permit by acquiring the semaphore using the `acquireUninterruptably()` method . By design, this operation will never block on the semaphore, since the change map only instructs the supervisor to disable events when they are currently enabled, and thus the related semaphore always has a permit when the supervisor attempts to acquire the semaphore. To enable an event, a permit must be added, which is accomplished by releasing the associated semaphore using `release()`.

A concrete example can be seen in the following section, where the precedence

```
private static void updateSupervisorState(String event) {
  if (event.equals(<<"eventName">>)) {
    //for the first event in the supervisor, insert:
    switch(Supervisor.supervisorState) {
      //for each state with this event as a transition out, insert:
      case(<<sourceState>>):
        //for each event to be enabled, insert:
        Supervisor.<<eventSemaphore>>.release();
        //for each event to be disabled, insert:
        Supervisor.<<eventSemaphore>>.acquireUninterruptibly();
        Supervisor.supervisorState = <<targetState>>;
        break;
    }
  }
    //for each additional event in the supervisor, insert:
    else if (event.equals("T3start")) {
      switch(Supervisor.supervisorState) {
      //for each state with this event as a transition out, insert:
      case(<<sourceState>>):
        //for each event to be enabled, insert:
        Supervisor.<<eventSemaphore>>.release();
        //for each event to be disabled, insert:
        Supervisor.<<eventSemaphore>>.acquireUninterruptibly();
        Supervisor.supervisorState = <<targetState>>;
        break;
    }
  }
}
```

Figure 4.12: The Java skeleton for `updateSupervisorState`.

example is completed and the complete supervisor is listed. It is assumed that the current supervisor state is tracked in a variable called `Supervisor.supervisorState`.

4.4.2 Results on Precedence Example

Code to provide concurrency control was generated for the precedence example using the Java implementation of Algorithm 2. The IDES supervisor generated at the end of Chapter 3 was examined, determining event information and the transition structure. This was used to build the change map. Next, the `Supervisor` class was created. This class is given in full in Figure 4.13, which spans several pages. The supervisor contains a semaphore named after each controllable event, initialized to match the initial supervisory state. The `observeAndReact` method was inserted, and the `updateSupervisor` state method was completed and inserted.

This implementation adds an initialization method `init()` that sets all supervisor variables to their initial values. When the code was model checked, issues were found with the initialization of the variables in `Supervisor`. Curiously, these errors were not reproducible via manual testing, and do not seem consistent with the operation of the Java programming language. It is unknown if these errors are due to the model checker, the code, or Java itself. However, adding an initialization method, called in the `main` method of the `Main` class before any threads were started, eliminated these errors. Being tangential to the research, the matter was not further pursued.

All events in the precedence example are controllable, so at each event code from Fig. 4.7 was inserted, differing only by the event name. The supervisor was created as a static class as described above. Though it appeared to make no difference in manual testing, calling the initialization method before creating the threads removed

possible deadlocks detected by model checking.

4.5 Code Verification

Given that DES theory is rigorously proven to generate correct, nonblocking supervisors, it is implied that the produced control policy satisfies the specifications and is nonblocking. However, the DES work is only as reliable as the input, and the concurrency control code is only reliable to the extent that it correctly implements the generated control policy. It would be reasonable to show that Algorithms 1 and 2 used to create the DES model and synthesize code, respectively, are correct through a formal correctness proof. However, this only goes so far in deciding the correctness of implemented code. Model checking can provide assurance that the code, as implemented, is correct. Model checkers are limited to work with a specific language, and we restrict further discussion of model checking to Java Pathfinder [22] (JPF), a model checker that works on Java source code.

By default, JPF will search for deadlocks only. However, this does not give the full assurance of correctness that is desired. The implemented code should be deadlock free, but should also satisfy all specifications. The easiest way to search for violated specifications is to instrument the code with assertions that would fail if and only if the specification were violated. A violated assertion causes an exception, which JPF can search for by adding `gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty` to the search properties.

Model checking even small examples can be expensive with regards to memory and time. When running JPF, the Java virtual machine was given extra memory by using the switch `-Xmx1536m`, where 1536 is the amount of memory, in megabytes,

```
package precedenceExample;

import java.util.concurrent.*;

public class Supervisor {

  static Semaphore T1finish;
  static Semaphore T3start;
  static Semaphore T4start;
  static Semaphore T5finish;
  static Semaphore T2start;
  static int supervisorState;

  public static volatile boolean t1finished;
  public static volatile boolean t5finished;

  public static void init() {
    Supervisor.T1finish = new Semaphore(1);
    Supervisor.T3start = new Semaphore(0);
    Supervisor.T4start = new Semaphore(0);
    Supervisor.T5finish = new Semaphore(1);
    Supervisor.T2start = new Semaphore(0);
    Supervisor.supervisorState = 0;
    Supervisor.t1finished = false;
    Supervisor.t5finished = false;
  }

  public static synchronized boolean observeAndReact(String
                                    event, Semaphore eventBlocker) {
    if (!(eventBlocker == null)) {
      if (!eventBlocker.tryAcquire()) {
        return false;
      }
      eventBlocker.release();
    }
    updateSupervisorState(event);
    return true;
```

Figure 4.13: Realized supervisor for precedence example.

```
}

private static void updateSupervisorState(String event) {

  if (event.equals("T1finish")) {
    switch(Supervisor.supervisorState) {
      case(0):
        Supervisor.T3start.release();
        Supervisor.T2start.release();
        Supervisor.T1finish.acquireUninterruptibly();
        Supervisor.supervisorState = 1;
        break;
      case(2):
        Supervisor.T3start.release();
        Supervisor.T4start.release();
        Supervisor.T2start.release();
        Supervisor.T1finish.acquireUninterruptibly();
        Supervisor.supervisorState = 5;
        break;
    }
  }
  else if (event.equals("T3start")) {
    switch(Supervisor.supervisorState) {
      case(1):
        Supervisor.T3start.acquireUninterruptibly();
        Supervisor.supervisorState = 4;
        break;
      case(3):
        Supervisor.T3start.acquireUninterruptibly();
        Supervisor.supervisorState = 6;
        break;
      case(5):
        Supervisor.T3start.acquireUninterruptibly();
        Supervisor.supervisorState = 8;
        break;
```

Figure 4.13 continued.

```
      case(7):
        Supervisor.T3start.acquireUninterruptibly();
        Supervisor.supervisorState = 10;
        break;
      case(9):
        Supervisor.T3start.acquireUninterruptibly();
        Supervisor.supervisorState = 12;
        break;
      case(11):
        Supervisor.supervisorState = 13;
        break;
    }
  }
  else if (event.equals("T4start")) {
    switch(Supervisor.supervisorState) {
      case(5):
        Supervisor.T4start.acquireUninterruptibly();
        Supervisor.supervisorState = 9;
        break;
      case(7):
        Supervisor.T4start.acquireUninterruptibly();
        Supervisor.supervisorState = 11;
        break;
      case(8):
        Supervisor.T4start.acquireUninterruptibly();
        Supervisor.supervisorState = 12;
        break;
      case(10):
        Supervisor.supervisorState = 13;
        break;
    }
  }
  else if (event.equals("T5finish")) {
    switch(Supervisor.supervisorState) {
      case(0):
        Supervisor.T5finish.acquireUninterruptibly();
```

Figure 4.13 continued.

```
        Supervisor.supervisorState = 2;
        break;
      case(1):
        Supervisor.T4start.release();
        Supervisor.T5finish.acquireUninterruptibly();
        Supervisor.supervisorState = 5;
        break;
      case(3):
        Supervisor.T4start.release();
        Supervisor.T5finish.acquireUninterruptibly();
        Supervisor.supervisorState = 7;
        break;
      case(4):
        Supervisor.T4start.release();
        Supervisor.T5finish.acquireUninterruptibly();
        Supervisor.supervisorState = 8;
        break;
      case(6):
        Supervisor.T4start.release();
        Supervisor.T5finish.acquireUninterruptibly();
        Supervisor.supervisorState = 10;
        break;
    }
  }
  else if (event.equals("T2start")) {
    switch(Supervisor.supervisorState) {
      case(1):
        Supervisor.T2start.acquireUninterruptibly();
        Supervisor.supervisorState = 3;
        break;
      case(4):
        Supervisor.T2start.acquireUninterruptibly();
        Supervisor.supervisorState = 6;
        break;
      case(5):
        Supervisor.T2start.acquireUninterruptibly();
```

Figure 4.13 continued.

```
          Supervisor.supervisorState = 7;
          break;
        case(8):
          Supervisor.T2start.acquireUninterruptibly();
          Supervisor.supervisorState = 10;
          break;
        case(9):
          Supervisor.T2start.acquireUninterruptibly();
          Supervisor.supervisorState = 11;
          break;
        case(12):
          Supervisor.supervisorState = 13;
          break;
      }
    }
  }
}
```

Figure 4.13 continued.

made available. Additionally, JPF was set to ignore repeated states at lower search depths by setting `search.match_depth` to `true`.

Precedence Example: The complete code, including generated concurrency control, was model checked using JPF. First, the code was instrumented using flags and assertions so that it could be verified that the specifications were properly enforced. The first specification, that `T1` must finish before `T2`, `T3`, and `T4` could start, was instrumented using a boolean variable `Supervisor.t1finished`. Initialized to false in the supervisor, this variable was not set true until `T1` completed, as seen in Fig. 4.14. It is important to note that this was done before the event `T1-finish` was observed by the supervisor. If this ordering were reversed, the supervisor could enable events in the other threads before the tracking variable correctly noted that `T1` had, in fact, completed. At the beginning of each of `T2`, `T3`, and `T4`, the assertion `assert (Supervisor.t1finished)` was inserted, as shown in Fig. 4.15 for `T3`. This assertion must be inserted immediately after the event, to ensure that when `T3-start` occurs, `t1finished` is already set to true. If this assertion ever evaluated to false, then the violating path was one that also violated the specification. Similar instrumentation was added for the specification relating to `T5` and `T4`.

After instrumentation, JPF was run on the precedence example. JPF found the code to be free of deadlocks and free from assertion violations. The model check completed within seconds on an AMD Athlon 64 3800+ desktop machine with 2GB of RAM running Windows XP x64. These results show that the generated concurrency control code for the precedence example is both nonblocking and within specification. The supervisor realization process presented in this chapter has successfully solved a single concurrency control problem. Further conclusions are withheld until

an additional example is explored.

```
public void run() {
  doWork();
System.out.println(id);
  //tracker noting that T1 has finished
  Supervisor.t1finished = true;
  //event marking: T1finish
  while (true) {
    ...
```

Figure 4.14: Instrumentation tracking the finish of **T1** for model checking.

```
public void run() {
  //event marking: T3start
  while (true) {
    if (Supervisor.observeAndReact("T3start", Supervisor.T3start))
      break;
    Supervisor.T3start.acquireUninterruptibly();
    Supervisor.T3start.release();
  }
  //model checking assertion
  assert(Supervisor.t1finished);
  System.out.println(id);
  doWork();
}
```

Figure 4.15: In `T3`, asserting that `T1` has finished.

Chapter 5

Transfer Line Example

The transfer-line problem is a well-studied problem in DES literature, and is described by Wonham in [44]. The problem is interesting in that it is modular, with the specifications presenting a conflict that must be addressed. The specifications cannot be directly enforced, otherwise a deadlock will ensue. Thus, the solution involves finding the largest controllable subset of the specifications through supervisor construction.

The problem is made relevant to this work by recasting it as a concurrency control problem in Java. The goal of this example is to show that the process, as presented in Chapters 3 and 4, can solve a concurrency control problem that is not addressed by existing processes. By automatically working around the conflicting specifications, this problem showcases the generative aspect of the process that is lacking in other work.

In the interest of readability, not all work relating to this problem is given in this chapter. Only work that is unique or necessary in obtaining a solution will be presented. That being said, all omitted work is included in Appendix B, and will be referenced accordingly.

5.1 Introduction

The transfer-line problem is centered around creating and processing parts that get moved between machines and buffers. Figure 5.1 gives a block diagram of the system, with the arrows showing the flow of parts. Additionally, each arrow refers to an event causing the corresponding part movement. Machine 1 (`M1`) creates parts and places them in Buffer 1 (`B1`) via event 1. Machine 2 (`M2`) removes parts from `B1` with event 2, works on them, and then on event 3 places them in Buffer 2 (`B2`). The test unit (`TU`) removes parts from `B2` via event 4 and then tests them. If the part passes the test, event 5 removes it from the system. If the part fails, event 6 uncontrollably returns the part to `B1`. The two buffers are both passive entities, while `M1`, `M2`, and `TU` are active components and cause the events. Each active component can only work on one part at a time. Specifications are quite simple. Buffer 1 has a capacity of three parts, `B2` has a capacity of one part, and buffers must never overflow or underflow.

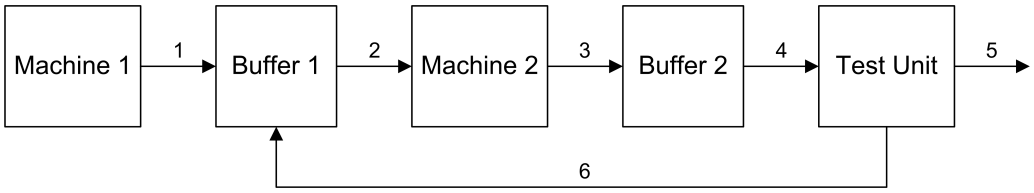


Figure 5.1: A block diagram showing the transfer-line system.

If the specifications are enforced as given, the system can deadlock. Path $p = 11123121$ is a event ordering that leads to problems. Both buffers are full at the end of p, and `M2` is busy working on a part. If `M1` acts by causing an occurence of event 1, `B1` will overflow. If `M2` acts by causing event 3, `B2` will overflow. If the test unit takes a part from `B2`, it could reject the part and then overflow `B1` via event 6. Thus,

no active component can act and the system is deadlocked, despite all specifications being met. DES theory allows us to overcome this issue by generating a nonblocking supervisor that enforces the largest subset of controllable behaviours.

5.1.1 Java Implementation

The Java implementation creates a thread for each of the three active components `M1`, `M2`, and `TU`. A `Part` class was defined to allow for the creation of objects to be passed through the system. Buffers are objects that are instantiated by the `Main` class and are used as shared resources by the active components. The `Main` class instantiates the buffers and active components, starts the active components, then terminates. Listings of actual code are given in Appendix B, Section B.1.

To ensure data integrity of the shared buffers, we add mutual exclusion specifications. Simply put, `B1` must be accessed in mutual exclusion, and `B2` must also be accessed in mutual exclusion. For user output, each buffer access is coupled with console output describing the result of the action. To ensure output matches the program state, the output statement is considered part of the buffer access and is thus included in the mutual exclusion specification.

5.1.2 Building the Event Set

Specifications describe the intended behaviour of the program. Each specification will describe the behaviour of some portion of the code. By identifying these sections, relevant events can then be determined. Each relevant event refers to a specific location in the code—a location that must be marked. To build the event set, every specification must be examined in turn, and then the location of relevant events

marked in code.

- **`B1` may not overflow or underflow**: This specification refers to parts being placed in or removed from `B1`. Thus, any software event that is immediately followed by code that places a part into `B1` is a relevant event. This happens in `M1`, and also in `TU` if a part is rejected (this event is also uncontrollable, as indicated in the problem description). Similarly, `M2` also has a relevant event arising from the removal of parts from `B1`. The added relevant events are `M1addB1-start`, `TUaddB1-start-u`, and `M2getB1-start`.

- **`B2` may not overflow or underflow**: Similar to the specification above, any software event followed by code that places a part in `B2` becomes a relevant event, along with any software event followed by code that removes a part from `B2`. Placing a part in `B2` only occurs in `M2`, and removing a part occurs in `TU`. The added relevant events are `M2addB2-start` and `TUgetB2-start`.

- **`B1` and `B2` must each be accessed in mutual exclusion**: Buffer access start events are already defined as relevant events. New relevant events are needed to note completion of buffer access. Any software event that immediately follows a buffer access (add or remove) becomes relevant. Also, we assume that when an active component finishes with a buffer, that finish is uncontrollable. The added relevant events are `M1addB1-finish-u`, `TUaddB1-finish-u`, `M2getB1-finish-u`, `M2addB2-finish-u`, and `TUgetB2-finish-u`.

Now that the events have been determined, they are marked in code. Each event falls in the `run` method of one of `M1`, `M2`, or `TU`. The event-marked `run` method for `M1` is given in Fig. 5.2 on the next page, the event-marked `run` method for `M2` is given in

Fig. 5.3 on the following page, and the event-marked `run` method for `TU` is given in Fig. 5.4 on page 79.

There is a legitimate question as to where these events should be marked. Would it be correct to instead locate the events in the `Buffer` class? In this implementation, active components output console messages on buffer accesses. The messages should be synchronized with the actual buffer access, ensuring that the output reflects the actual program state. For this reason, the relevant events must be marked in the active components. Were this not the case, either location would lead to a correct solution. However, locating events in the buffer class would lead to a shared event, resulting in a more complicated solution.

```
public void run() {
  while (true) {
    //wait for some random period
    doWork();
    //create a new Part
    Part newPart = new Part();
    //event marking: M1addB1-start
    //put it in the target buffer
    System.out.println("Machine 1 tries to put a part in " + target);
    target.addPart(newPart);
    //event marking: M1add-B1-finish-u
  } // loop forever
}
```

Figure 5.2: The event marked `run` method for `M1`.

```
public void run() {
  while (true) {
    //get part from source Buffer
    //event marking: M2getB1-start
    System.out.println("Machine 2 tries to get a part from "
                       + source);
    Part currentPart = source.removePart();
    //event marking: M2getB1-finish-u
    //we have a part - do some work on it
    doWork();
    //put the part in the next buffer
    //event marking: M2addB2-start
    System.out.println("Machine 2 tries to put a part in " + target);
    target.addPart(currentPart);
    //event marking: M2addB2-finish-u
    //reset machine 2
    doWork();
  } //loop forever
}
```

Figure 5.3: The event marked `run` method for `M2`.

```
public void run() {
  while (true) {
    //get a part from the source
    //event marking: TUgetB2-start
    System.out.println("Test Unit tries to get a part from "
                       + source);
    Part currentPart = source.removePart();
    //event marking: TUgetB2-finish-u

    //test that part for some period of time
    doWork();
    if (Math.random() > rejectionChance) {
      //no good!  send back to rejectBin
      //event marking: TUaddB1-start-u
      System.out.println("Test Unit tries to put a part in "
                         + rejectBuffer);
      rejectBuffer.addPart(currentPart);
      //event marking: TUaddB1-finish-u
    }
  } //loop forever
}
```

Figure 5.4: The event marked `run` method for `TU`.

5.2 Building the Plant

As per Algorithm 1 presented in Fig. 3.6 on page 31, control-flow graphs were produced en route to obtaining the DFA representation of each thread. The `Main` class, `Part` class, and `Buffer` class were immediately discarded, as none of these classes contained either a relevant event or a method call that led to a relevant event (other than the `thread.run()` calls in the `Main` class, which are ignored).

The CFGs for `M1`, `M2`, and `TU` were produced as they contained relevant events. Nodes representing calls to the buffer were left unexpanded as the calls contain no

relevant events. The CFGs were transformed into DFAs and then reduced as described in Algorithm 1. In Fig. 5.5, both the original CFG and the reduced DFA are given for `M1`. In Fig. 5.6, the CFG-DFA pair is given for `M2`, and Fig. 5.7 on the next page presents the CFG-DFA pair for `TU`.

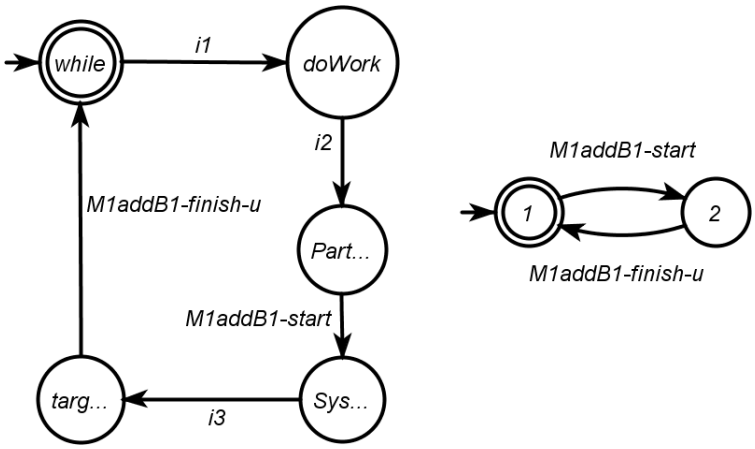


Figure 5.5: Transfer-Line: The CFG and the reduced DFA for `M1`.

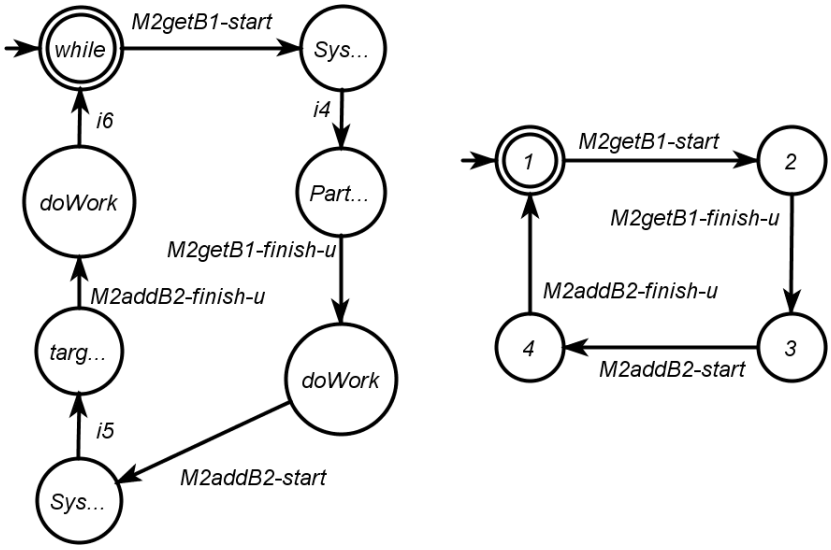


Figure 5.6: Transfer-Line: The CFG and the reduced DFA for `M2`.

The reduced DFA for `TU` presents a remaining irrelevant event, `i9`. It lies on the

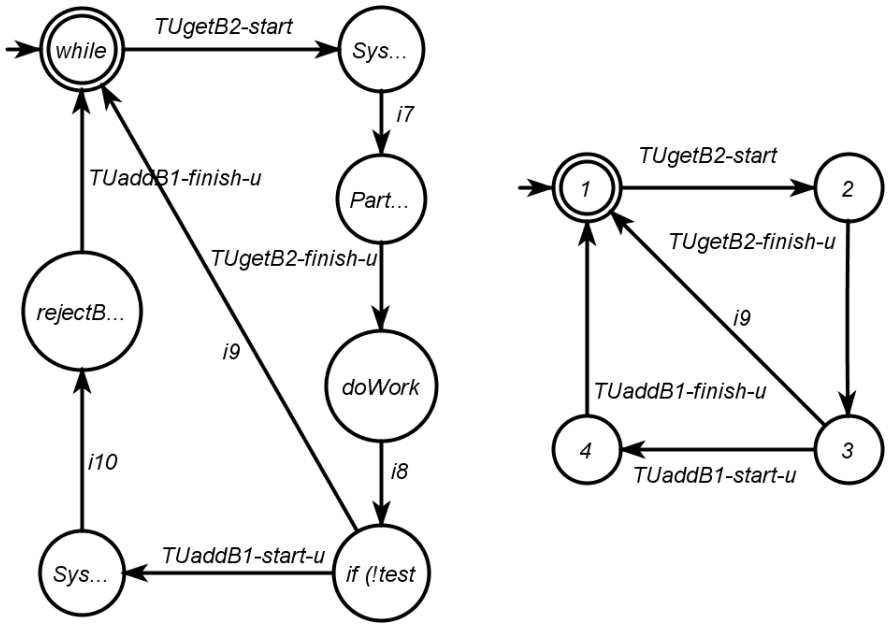


Figure 5.7: Transfer-Line: The CFG and the reduced DFA for `TU`.

transition between the `if` statement that tests a part and the `while` statement at the beginning of the code. This is an implicit path, so the source must be transformed to make that path explicit. Since this is on an `if` statement, and there is no existing `else` component, the `if` statement is transformed into an `if-else` statement to make the path explicit. The `run` method in `TU` is thus modified and the irrelevant event `i9` is marked inside. This is shown in Fig. 5.8 on the following page.

As there are no shared events to be dealt with, and as all threads start at the same time (thus meeting the no-dynamic threads assumption), it is safe to build the plant. The plant is given by the synchronous product of each of the reduced DFAs. The resulting plant has 32 states and 104 transitions, representing all possible interleavings of the three threads being considered. A visual representation is not helpful due to the large state-space. The full transition listing is given in Appendix B, Fig. B.8 on page 136.

```
public void run() {
  while (true) {
    //get a part from the source
    //event marking: TUgetB2-start
    System.out.println("TestUnit tries to get a part from "
                       + source);
    Part currentPart = source.removePart();
    //event marking: TUgetB2-finish-u

    //test that part for some period of time
    doWork();
    if (Math.random() > rejectionChance) {
      //no good!  send back to rejectBin
      //event marking: TUaddB1-start-u
      System.out.println("TestUnit tries to put a part in "
                         + rejectBuffer);
      rejectBuffer.addPart(currentPart);
      //event marking: TUaddB1-finish-u
    }

    //code to make i9 explicit
    else  {
      //event marking: i9
    }
    //end of code making i9 explicit
  } //loop forever
}
```

Figure 5.8: Transformed `run` method for `TU` placing `i9` on a now explicit path.

The event set for the plant is exactly {`M1addB1-start`, `M1addB1-finish-u`, `M2getB1-start`, `M2getB1-finish-u`, `M2addB2-start`, `M2addB2-finish-u`, `TUaddB1-start-u`, `TUaddB1-finish-u`, `TUgetB2-start`, `TUgetB2-finish-u`, `i9`}. This is clearly noted as all specifications will use this same event set. Additionally, because of the presence of an irrelevant event in the event set, this event set was not available before this stage.

5.3 Specifications

Each of the specifications is now ready to be presented as a separate DFA. All use the same event set as the plant. The overflow and underflow specifications are similar to those found in [44]. It is important to remember that only those behaviours covered by a specification should be affected. All events that do not appear in the constructed DFA are added in self-loop to all states. A self-loop is a transition that begins and ends at the same state; events appearing in self-loop occur freely without having an effect on the system. Note: In the following figures showing the formal specifications, events in self-loop are *not* shown.

The first specification addressed is that **B1** may not overflow or underflow. Recall that `B1` has a capacity of three, that two events add parts (`M1addB1-start` and `TuaddB1-start-u`), and only one event removes parts (`M2getB1-start`). Prevention of overflow is ensured by disallowing both adding events when three buffer-add actions have occurred without any counter-balancing remove events. To prevent underflow, a remove event is only permitted when an unmatched add event has occurred. The formal specification is given in Fig. 5.9 on the following page. The set of events in self-loop is {`M1addB1-finish-u`, `M2getB1-finish-u`, `M2addB2-start`,

M2addB2-finish-u,

TUaddB1-finish-u, TUgetB2-start, TUgetB2-finish-u, i9}.

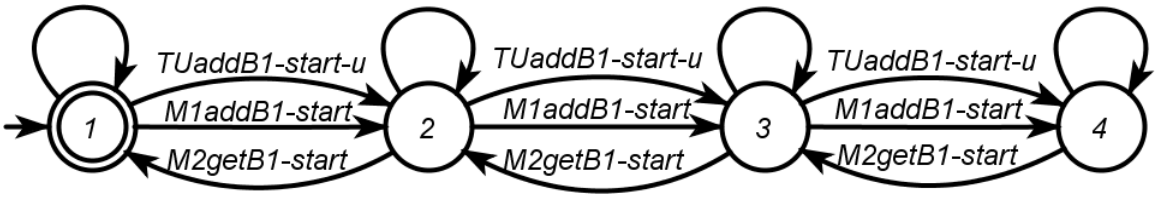


Figure 5.9: Specification preventing B1 overflow and underflow. Events in self-loop are not labelled.

The specification preventing overflow and underflow for B2 is treated similarly to the specification for B1. Here, only M2addB2-start adds parts to B2, and only TUgetB2-start removes them. The capacity of B2 is only one part, so each add must be followed by a remove. A remove can never precede an add. The formal specification is given in Fig. 5.10. The set of events in self-loop is {M1addB1-start, M1addB1-finish-u, M2getB1-start, M2getB1-finish-u, M2addB2-finish-u, TUaddB1-start-u, TUaddB1-finish-u, TUgetB2-finish-u, i9}.

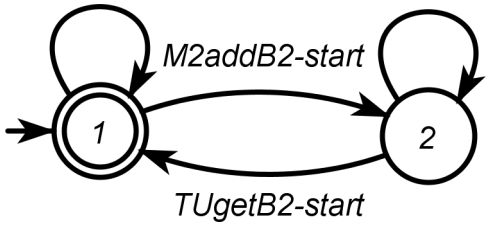


Figure 5.10: Specification preventing B2 overflow and underflow. Events in self-loop are not labelled.

Next, the mutual exclusion specification for B1 is formalized. There are six events that access B1: TUaddB1-start, TUaddB1-finish, M1addB1-start, M1addB1-finish, M2getB1-start, and M2getB1-finish. Whenever a start event begins an access to the buffer, the associated uncontrollable finish event must occur before any new start

event can occur. The DFA is given in Fig. 5.11. The set of events in self-loop is {`M2addB2-start`, `M2addB2-finish-u`, `TUgetB2-start`, `TUgetB2-finish-u`, `i9`}.

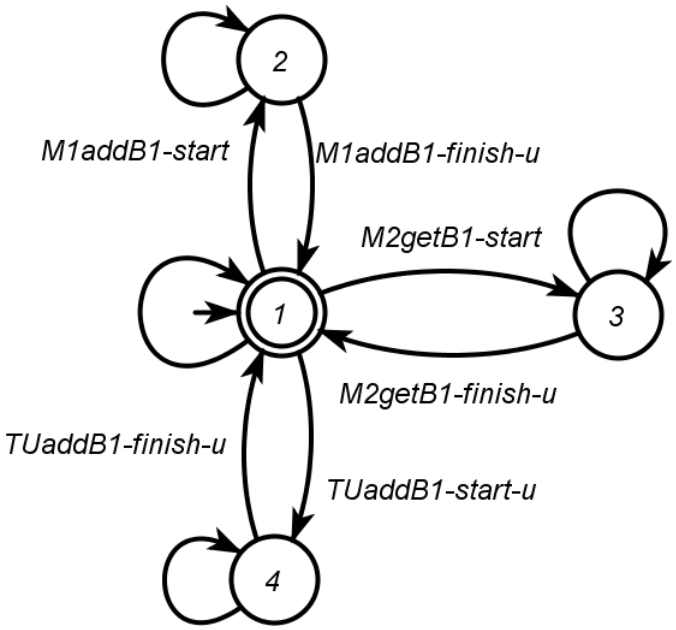


Figure 5.11: Specification enforcing mutually exclusive access to `B1`. Events in self-loop are not labelled.

Last is the mutual exclusion specification for `B2`. This buffer is accessed by `M2addB2-start`, `M2addB2-finish-u`, `TUgetB2-start`, and `TUgetB2-finish-u`. The formalized DFA is given in Fig. 5.12 on the next page. The set of self-loop events is {`M1addB1-start`, `M1addB1-finish-u`, `M2getB1-start`, `M2getB1-finish-u`, `TUaddB1-start-u`, `TUaddB1-finish-u`, `i9`}.

The four specifications taken together form a set of modular specifications. At this stage, it is not typically known if the specifications lead to conflicting or non-conflicting specifications. Usually the next step would be to synthesize multiple supervisors and then test for conflicts. However, prior knowledge of the problem tells us that the specifications will in fact lead to conflicting supervisors. To avoid this,

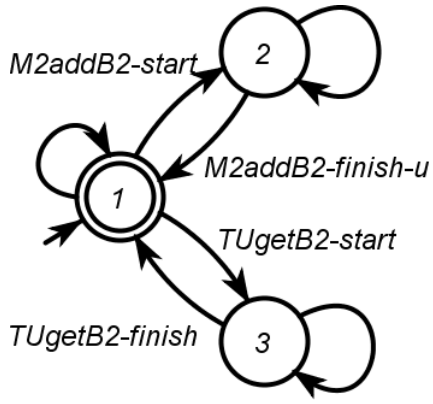


Figure 5.12: Specification enforcing mutually exclusive access to `B2`. Events in self-loop are not labelled.

the monolithic supervisor is generated by finding the intersection of all specifications. Only those behaviours allowed in all specifications are allowed in the monolithic specification. It is for this reason that it is imperative to properly assign events to self-loops when creating specifications.

The monolithic specification was computed using IDES. As it contains 52 states and 176 transitions, it is too large to display here. A full transition listing is provided in Appendix B as Fig. B.9 on page 139.

5.4 Supervisor Synthesis

Both the plant and monolithic supervisor were exported to TCT, wherein the `SUPCON` operation was employed to generate a supervisor. The supervisor contains 69 states and 114 transitions. As with other large DFAs, it is not represented here, though interested readers can find a transition listing in Appendix B as Fig. B.10 on page 144. Instead of producing the exact legal language, the supervisor produces the largest controllable nonblocking subset of the legal language. This resolves the deadlock that

arises due to the conflicting specifications.

For sake of comparison, the conflicting modular supervisors were generated and coupled with the plant. The closed-loop system contains 71 states and 116 transitions, exactly two more of each than the monolithic supervisor. Recall that in Section 5.1 on page 74, a path was presented that leads to system deadlock. The path results in both `B1` and `B2` being at capacity while `M2` is working on a part. When the modular closed-loop system was examined, this path was found to have been preserved. When `B2` is full, `M2` is working on a part, and `B1` contains two parts, `M1` was allowed to place a third part in `B1`, thus filling it. After this, the system was in a deadlock state. In the Java implementation, the final `M1-addB1-start` is followed by an `M1-addB1-finish-u`; these two transitions lead to the two additional states and events that are found in the modular supervisor but not the monolithic one. In the monolithic supervisor, the event where `M1` places that final part in `B1` is disabled, despite the fact that the event is allowed in all specifications. By doing this, the ensuing deadlock is avoided without introducing a new specification or modifying one that already exists.

5.5 Code Generation and Verification

Using the event-marked code and the generated supervisor, Algorithm 2 was applied to generate code. Generated code is largely boilerplate in nature, and follows exactly the results of Chapter 4. No significant errors or problems were encountered during the process; everything ran as expected. Readers interested in the actual generated code are referred to Appendix B, Section B.6.

Like the precedence example, the transfer-line solution was subjected to model checking with the JPF tool. The code was instrumented with assertions to check

for specification violation. Since all specifications relate to buffer access or buffer overflow/underflow, the added assertions were all in the `Buffer` class. Figure 5.13 gives an excerpt of the `Buffer` class containing all added assertions.

The instrumentation to verify mutual exclusions works by tracking buffer access. Each buffer is given a flag, defaulting to false, that indicates if the buffer is in use. When any thread accesses a buffer by either adding or removing a part, the first action is to assert that the flag is false, which is to say, the buffer is not in use. Immediately after the assertion, the flag is set to true indicating the buffer is now being accessed. Once the access is complete, the flag is reset to false. If a thread were to access a buffer while it was already in use, the flag would be true and the assertion would fail. JPF would identify this as an exception and report the error.

Instrumention for overflow and underflow was straightforward. The `buffer` class already checked for overflows and underflows with corresponding output to the console. After the output statements, an `assert(false)` statement was inserted. If these statement were ever reached, then overflow or underflow had occured, the failed assertion would cause an exception, and JPF would identify the error.

Model checking of this problem completed in under an hour on the same AMD Athlon 64 3800+ desktop machine with 2GB of RAM running Windows XP x64 used for the precedence example. Settings were the same as those used for the precedence example. Once again, the model check was successful, finding no deadlock and no assertion violation. This leads us to conclude that the generated code is deadlock-free and within specification.

```
//new boolean tracker for mutual exclusion
boolean active;

public void addPart(Part p) {
  //assertion to catch Mutex violations
  assert(!active);
  active = true;
  if (holding < size) {
    holding++;
    //System.out.println (name + " is holding " + holding
                          + " parts.");
    partBuffer.add(p);
  }
  else {
    System.out.println(name + " overflow!!!");
    //following assertion catches overflows
    assert false;
  }
  active = false;
}
public Part removePart() {
  //assertion to catch Mutex violations
  assert(!active);
  active = true;
  if (holding > 0) {
    holding--;
    //System.out.println (name + " is holding " + holding
                          + " parts.");
    active = false;
    return partBuffer.remove();
  }
  else {
    System.out.println(name + " underflow!!!");
    //following assertion catches underflows
    assert false;
    return null;
  }
}
```

Figure 5.13: Transfer-Line: `Buffer` class with assertions for model checking.

Chapter 6

Literature Review

Many approaches have been undertaken to automatically generate concurrency control code. In the software engineering community, it is common to create formal specifications, and then apply them in the synthesis of concurrent control code. These approaches guarantee that the specifications are faithfully implemented, but usually do not guarantee that the resulting code will be correct and deadlock-free. Control theorists use DES to automatically generate correct control schemes, but are not concerned with code. By combining results from these previously disparate fields, we use DES to generate correct control schemes that are then translated into code. Relevant results from each field are discussed in the first two sections. In addition, there are alternate approaches to concurrent code generation that do not fit easily into these categories. These are discussed as other approaches at the end of this chapter.

6.1 Specifications and Concurrent Code

Serious efforts have long been made to automatically synthesize concurrency controls from specifications. Early work is based on Campbell and Habermann's Path Expressions [6], which allow the specification of the allowed sequences of operations on an object of an abstract data type. In [2], Andler extends this idea by proposing Predicate Path Expressions (PPEs) and also describes an implementation scheme which is based on the translation of PPEs to finite automata. Every invocation of an operation of the data type is bracketed by a prologue and an epilogue which consults the automaton and ensures only conforming invocations can proceed. The potential for deadlock in the generated code is not discussed, but the use of formal methods is suggested. These methods focus on translating a specification into code—they do not consider how these specifications might interact with each other or with the program.

Emerson and Clarke, in [15], use branching time temporal logic to produce 'synchronization skeletons', which are program abstractions suppressing detail irrelevant to synchronization. Specifications are given in Computational Tree Logic (CTL), but can only be used to create a synchronization skeleton if the CTL specifications are satisfiable. The program is modelled as a 'finite model' (a simpler version of a deterministic finite-state automaton) through the definition of non-critical sections, critical sections, and try sections found at the boundary when moving from non-critical to critical sections. Using CTL formulae, a scheme is developed that details when each thread is allowed to enter a critical section without violating the specifications. A similar method was proposed by Manna and Wolper in [31] using Linear Temporal Logic (LTL) rather than CTL. In a broad manner, our work mirrors this approach. However, the substitution of DES solves several major issues, such as unsatisfiable or

conflicting specifications, and can provide guarantees on the quality of the solution.

The general approach from [15] has been adopted several times, differing by the usage of an alternate specification formalization or a different modelling technique. In [5], specifications are given using process algebra, whereas [47] employs Bultan's Action Language. Tools allow the formal analysis of the specifications. An approach based on global invariants is discussed in [11]. The invariants specify allowed behaviour and use specific counter variables that keep track of how many processes are currently executing user-specified regions. Patterns of some common specifications are given. Support for model checking the generated code is provided to detect synthesis errors. These approaches improve the range of implementable specification, but still run into the same issues as [15].

6.2 Applying DES to Concurrency

A theory of controlling discrete-event systems was intiated by Ramadge and Wonham in [36, 37, 45, 38]. The authors indicate that DES theory is applicable to computing, but the idea is left unexplored. To the best of our knowledge, the application of DES to software development is left unaddressed in the DES community.

Modular DES, developed in [46], extends the standard DES framework. The work allows for multiple supervisors to act on a system, and thus introduces an easy method to include multiple specifications. In addition, it provides a simple mathematical test to ensure that the modular supervisors, when coupled with the plant, will be nonblocking. This test is vitally important to our work, as it will be used to identify undesired interactions between inconsistent specifications.

There has been work in the opposite direction, where results from model-checking

have been applied to DES results. The supervisor control problem was revisited by Ziller and Schneider in [48, 49], where the authors generalize the supervisor synthesis algorithm to allow for specifications given in the μ-calculus. This allows the consideration of fairness properties in addition to safety and liveness properties.

Some limited attempts have been made by control theorists to address the concurrency control problem. Thistle, in [42], adapted the verification framework from Manna and Pnueli [30] to solve the control problem for a small set of abstract concurrent processes. It used linear temporal logic and modeled both the system and specifications as a set of logic statements. This approach allowed safety properties and some liveness properties, but was manual in nature. In a certain context, one could argue that any DES problem with more than one module being used to form the plant does in fact deal with concurrency. This is partly why DES seemed to be an interesting approach to the problem. However, no known approach deals explicitly with the development of concurrency controls in software.

6.3 Other Approaches

Alt, Sander and Wilhelm present an approach for the generation of synchronization code for parallel compilers [3]. The compiler is modularized into engines. A global dependence graph is computed from specifications of the input-output behaviour of these engines. From this graph, code controlling the invocation of engines and access to shared data structures is generated. Deadlock avoidance is guaranteed, but the access policy has no guarantee on its restrictiveness.

Matos et al.[32] describe a technique for the automatic generation of synchronization conditions based on finite-state machine descriptions of both the components

and the specifications [20]. A synchronous communication model such as Esterel, Lustre, or SMV is assumed. The tool implementing the approach supports this by outputting an SMV representation of the system. The resulting code needs to be checked for deadlocks, because "circular dependence between synchronized components" will cause deadlocks. This contrasts with controllability results in DES, where the largest controllable subset is used to resolve circular dependencies.

In [4], Autili et al. describe a tool called SYNTHESIS, which produces correct and deadlock free distributed component systems. Several examples are cited where the tool is successfully applied. They introduce a concept of 'last chance' states, which are the last chance to prevent a deadlock by preventing some transition. This echoes early DES work by Ramadge and Wonham, wherein last chance states correspond to the last states where control could be applied. The mirroring here highlights the relevance of introducing DES theory to concurrency control problems, as it is made clear by this thesis that success can be found using DES-like techniques.

Aspect-oriented programming, introduced by Kiczales et al. in [24], is based on identification of cross-cutting concerns, coding each concern individually, and then weaving it back into the main code base. Recent related work [13] deals explicitly with concurrent aspects. Here, each thread is modeled as an aspect with the weaving advice acting as the concurrent control scheme. Advice can be represented as a finite-state graph, and aspects are modeled to act in parallel. Unlike DES, there is no mathematical guarantee on the correctness of the result, since there is no given method to confirm the viability of the control scheme *a priori*. As the authors note, "In addition, because of the inherent difficulty of developing correct concurrent programs, [...] a model for concurrent aspects should support the use of automatic

verification techniques, such as model checking...".

Chapter 7

Conclusions

This work presented a process to use discrete-event systems to generate concurrency controls. The primary result has been to show that DES can be successfully applied to the problem of concurrency control coding. This has been facilitated through the design of two algorithms—one to transform code into a DES plant, and one to transform a DES supervisor into concurrency control code. The efficacy of the process has been demonstrated by solving two example problems. It has been shown that the process presented, and more generally DES, is a viable approach in the automatic generation of concurrency control code.

The basic link between DES and software was formed through the novel concept of software events, and applied by the process of event marking. The process to make implicit paths explicit, and the unsharing of shared events provides the means to disambiguate any event and makes the link solid. This also naturally leads to the introduction of uncontrollable events to the concurrency control discussion.

Constructing the CFG allows for the behaviour of the program to be captured in an automated fashion. Specifications describe the desired paths through the program,

and provide the desired concurrent behaviour. After the application of DES theory, a supervisor is synthesized. One of the strengths of this process lies in the generative nature of DES. Unlike other concurrency control generation methods found in the literature, DES actually generates a control policy. Using a model of the code, and a set of specifications, a supervisor is built to enforce the largest controllable subset of the specifications. Furthermore, the control policy embodied by the supervisor is guaranteed to create a nonblocking closed-loop system, free of both deadlock and livelock. All of these claims about the supervisor and DES have been mathematically proven in DES literature.

Once the abstract supervisor has been synthesized, a novel approach is taken to realize the supervisor as code. Special allowances had to be made to account for the instantaneous nature of DES theory, and the fact that supervisor actions take time when actually implemented. This led to the careful synchronization of the observation and control action steps of the supervisory control loop, found in the looping nature of the semaphore checks at the event and the ordering of steps in the `observeAndReact` and `updateSupervisorState`. Additionally, using the event markings, it should be possible to encapsulate the entire code generation process as part of a tool. In fact, this was partially done in this initial research, showing the viability of automation.

A problem common in concurrency control code generation is solved by the introduction of DES. While specifications may be consistent with each other, it is possible that when applied to the code, deadlocks may result. This problem has been called 'circular dependency', and is known as conflicting specifications in DES literature. This process automatically resolves these problems by finding the 'best-possible' solution, that is, the supervisor that enforces the largest controllable subset of the

specified behaviour while guaranteeing a nonblocking system. By creating a nonblocking supervisor using the monolithic specification, the problem is avoided without any user intervention. No other known approach to generate concurrency control code can guarantee this. These claims are directly supported by the solution to the transfer-line problem, which had exactly this type of problem. By correctly generating a working solution, the process was able to overcome conflicting specifications.

Few references and results are recent. Though this is somethime worrisome, it is the author's opinion that this reflects positively on our work. Primarily, this work is the union of two generally disparate fields—control theory from electrical engineering and software development from computer science. The groundwork has existed for this connection to be made since the early 1980's, but to our knowledge, this research is the first to make it. The absence of directly related work only reinforces the conclusion that this work is novel.

7.1 Future Work

As a first foray into the strategy of using discrete-event systems to automatically generate concurrency controls, this work is largely a proof-of-concept. There are many areas that readily present themselves for future work. Most are centered around expanding the capabilities of the process described, while a few focus on improving it.

It should be noted that, while these research directions do not seem to be mutually exclusive, they do represent different forks in DES research. Working in multiple theories may require significant theoretical investments in the DES research before a single coherent DES framework can be constructed. For instance, a theory of

DES based around μ-calculus exists, and a theory of dynamic DES exists, but the combination has not been researched—there is not yet a μ-calculus based dynamic DES theory.

7.1.1 Liveness Specifications

The Ramadge and Wonham DES framework was originally built using DFAs as the modelling structure. Since specifications are given as DFAs, only specifications expressible as a regular language can be used in the current process. This excludes any liveness specifications, and is a significant limitation on this work. All specifications are either safety, liveness, or both, so incorporating liveness specifications will yield a much more versatile solution.

First and foremost among these is the μ-calculus given by [25], which is able to express both safety and liveness conditions. Ziller and Schneider rework the supervisor synthesis problem [48, 49] to start from a plant and specifications both given as sets of μ-calculus equations. The generated supervisor is also a set of μ-calculus equations.

Work on this would be divided into two tasks. First, the plant is currently built from CFGs that are transformed into DFAs. Since μ-calculus subsumes regular languages, the DFAs can be converted into μ-calculus equation sets. A method for this would need to be implemented. Secondly, the resulting supervisor gives a control policy. This must be extracted from the μ-calculus equations and used to synthesize code. This is complicated by the complex path information that gives the μ-calculus its expressiveness. How can this path information be translated into code?

There are other formulations of DES that can be applied, though none are as expressive as the μ-calculus formulation. These include a CTL* recasting of the

supervisory control problem by Jiang and Kumar [21], work on the infinite behaviour of finite automata by Thistle and Wonham [43], and a wide range of material based on Petri-nets (see [7] for an introduction to the field). However, due to the expressiveness, work using the μ-calculus seems to offer the greatest potential.

7.1.2 Thread Creation and Termination

This work assumes that threads all come into existence at the same time, then run their course. However, this is not a safe assumption in general. Dynamic thread creation in languages such as Java and C++ allow for threads to be created and started at almost any point in the code. The method used here, synchronous product, to combine thread DFAs first assumes that all threads start at the same time.

A change in this assumption would have to cover the possibility of threads appearing and disappearing over the course of the execution. Either the plant would have to be mapped out in full, for all possible appearance/disappearance sequences, or a system would have to be created that assumes nothing and instead reacts to changes in the plant. Dynamic DES theory, proposed by Grigorov and Rudie in [17], offers a DES framework that accounts for changes in the plant over time, and could offer a way to work around this assumption.

7.1.3 Multiple Thread Instances

It is possible in an object oriented language to create a thread object, then instantiate it repeatedly. As an example, consider a database application with many identical reader threads, and many identical writer threads. A typical specification would be that when no writers are writing, multiple readers can access a shared resource

simultaneously. A specification of this type is difficult to formalize when each reader and each writer is treated as a separate entity. This becomes even harder when the number of reader and writer threads is not known at run time. What is needed is a method to abstract individual and identical threads into a group, such as a group of readers or a group of writers.

Parameterized DES, as proposed by de Oliveira et al. in [34], seems to offer a solution to the problem. States in the FSA model are given parameters that track some system value. In the case of a reader-writer problem, parameters could store the number of readers currently reading, the number of writers waiting to write, or other important values. A similar theory called "counting abstraction", given in [10], is used by Yavuz-Kahveci and Bultan in [47] to model an arbitrary number of processes. These results may be applicable in expanding the work at hand.

7.1.4 Other Areas

A wide range of other research areas exist. First, efficiency of the process has not been addressed. This is especially true in regards to the generated code. Quite simply, efficiency was never a consideration in this research. It is felt that significant gains could be made in this area by a person with a good background in concurrency and efficiency.

Aspect-oriented programming has the straightforward functionality to weave code together. This could form the basis of an elegant solution to weave generated supervisor code back into the marked source code. In Java specifically, tools such as AspectJ [23] may be effective in facilitating this.

Real-time systems are not addressed by the standard Ramadge-Wonham framework. There is, however, an expansion to the framework that provides this capability. In [28], the concept of time is built into the model using clock-tick events that occur in regular intervals.

Distributed systems provide a different set of challenges for a concurrent system. Additionally, there is the potential for remote events to be unobservable by a local supervisor. Theories of DES based around partial observation [8] allow for the construction of supervisors with unobservable events. Additionally, the generated code would need to be different, as distributed programming differs from concurrent programming.

To make a stronger claim of usefulness, further problems should be addressed, especially problems with a non-obvious solution. The presented problems are simple and possess obvious solutions. Solving additional problems would both demonstrate the value of the solution, and likely give insight as to ways in which the process can be refined and improved.

Finally, it is possible for infeasible paths to be preserved in the DFA models, as the content of the code is never examined. This could result in control decisions being made to avoid paths that are not actually viable. This is a special concern in situations where branching decisions are not made until run time, such as the completion of a method call on an instance of an object in a language that supports polymorphism. Some type of static analysis could be worked into the process to help remove infeasible paths.

Bibliography

[1] R. Adhikari. Intel, Microsoft: The future of computing is parallel. *TechNewsWorld*, Mar. 2008. `http://www.technewsworld.com/story/62199.html?wlc=1221878232`.

[2] S. Andler. Predicate path expressions. In *Proceedings of the 6th ACM Symposium on Principles of programming languages*, pages 216–236, Jan. 1979.

[3] G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addision Wesley, 2000.

[4] M. Autili, P. Inverardi, A. Navarra, and M. Tivoli. Synthesis: A tool for automatically assembling correct and distributed component-based systems. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 784–787, 2007.

[5] E. Bontà, M. Bernardo, J. Magee, and J. Kramer. Synthesizing concurrency control components from process algebraic specifications. In *Proceedings of the 8th International Conference on Coordination Models and Languages*, pages 221–362, Jun. 2006.

[6] R. H. Campbell and A. N. Habermann. The specification of process synchronization by path expressions. *Lecture Notes on Computer Science*, 16:89–102, 1974.

[7] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems.* Springer Science+Business Media, Inc., 1999.

[8] R. Cieslak, C. Desclaux, A. S. Fawaz, and P. Varaiya. Supervisory control of discrete-event processes with partial observations. *IEEE Transactions on Automatic Control*, 33(3):249–260, Mar. 1988.

[9] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.

[10] G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *Proceedings of the 12th International Conference on Computer Aided Verification*, pages 53–68, London, UK, 2000. Springer-Verlag.

[11] X. Deng, M. B. Dwyer, J. Hatcliff, and M. Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *Proceedings of the 24th International Conference on Software Engineering*, pages 442–452, 2002.

[12] E. W. Dijkstra. *Cooperating Sequential Processes.* Academic Press, 1965.

[13] R. Douence, D. Le Botlan, J. Noyé, and M. Südholt. Concurrent aspects. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, pages 79–88, 2006.

[14] E. A. Emerson. *Handbook of Theoretical Computer Science*, chapter Temporal and Modal Logic. North-Holland Publishing Company, 1995.

[15] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 3(2):241–266, 1982.

[16] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[17] L. Grigorov and K. Rudie. Near-optimal online control of dynamic discrete-event systems. *Discrete Event Dynamic Systems*, 16(4):419–449, 2006.

[18] P. Brinch Hansen. The programming language concurrent Pascal. *IEEE Transactions on Software Engineering*, 1(2):199–207, Jun. 1975.

[19] C. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[20] IDES: The integrated discrete-event systems tool. Discrete-Event Control Systems Lab, Queen's University, available at `http://www.ece.queensu.ca/hpages/labs/discrete/software.html`, Mar. 2008.

[21] S. Jiang and R. Kumar. Supervisory control of discrete event systems with CTL* temporal logic specifications. *Proceedings of the 40th IEEE Conference on Decision and Control*, 5:4122–4127, 2001.

[22] Java Pathfinder. Robust Software Engineering Group, NASA Ames Research Center, Sourceforge project page available at `http://javapathfinder.sourceforge.net`, Mar. 2008.

[23] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European*

Conference on Object-Oriented Programming, pages 327–353, London, UK, 2001. Springer-Verlag.

[24] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, 1997.

[25] D. Kozen. Results on the propositional μ-calculus. In *Proceedings of the 9th Colloquium on Automata, Languages and Programming*, pages 348–359, 1982.

[26] Leslie Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, 1989.

[27] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[28] Y. Li and W. M. Wonham. Supervisory control of real-time discrete-event systems. *Information Sciences*, 46:159–183, 1988.

[29] F. Lin and W. M. Wonham. On observability of discrete-event systems. *Information Science: An International Journal*, 44(3):173–198, 1988.

[30] Z. Manna and A. Pnueli. Verification of concurrent programs: A temporal proof system. Technical Report CS-TR-83-967, Stanford University, Dept. of Computer Science, 1983.

[31] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, Jan. 1984.

[32] G. Matos, J. Purtilo, and E. White. Automated computation of decomposable synchronization conditions. *Proceedings of the 2nd IEEE High-Assurance Systems Engineering Symposium (HASE 97)*, pages 72–77, Aug. 1997.

[33] R. Merritt. Wintel will fund parallel software lab at Berkeley. *EE Times*, Feb. 2008. `http://www.eetimes.com/showArticle.jhtml?articleID=206503988`.

[34] C. Oliveira, J. Cury, and C. Kaestner. Synthesis of supervisors for parameterized and non-regular discrete event systems. To appear in *IEEE Transactions on Automatic Control*.

[35] D. Patterson. A conversation with John Hennessy and David Patterson. *ACM Queue*, 4(10), Dec. 2006.

[36] P. J. Ramadge and W. M. Wonham. Supervision of discrete event processes. In *Proceedings of the 21st IEEE Conference on Decision and Control*, volume 3, pages 1228–1229, Dec. 1982.

[37] 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. Also appears as Systems Control Group Report #8311, Department of Electrical Engineering, University of Toronto, 1983.

[38] P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1), Jan. 1989.

[39] Σοφια: A Java bytecode analysis tool. University of Nebraska-Lincoln, available at `http://sofya.unl.edu/`, Mar. 2008.

[40] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs Journal*, 30(3), Mar. 2005.

[41] TCT. Systems and Control Group, Dept. of Electrical and Computer Engineering, University of Toronto, available at `http://www.control.toronto.edu/DES`, Mar. 2008.

[42] J. Thistle and W. M. Wonham. Control problems in a temporal logic framework. *International Journal of Control*, 44(4):943–976, 1986.

[43] J. G. Thistle and W. M. Wonham. Control of infinite behavior of finite automata. *SIAM Journal on Control and Optimization*, 32(4):1075–1097, 1994.

[44] W. M. Wonham. Supervisory control of discrete-event systems. Systems Control Group, University of Toronto, 2006.

[45] 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. Also appears as Systems Control Group Report #8312, Department of Electrical Engineering, University of Toronto, 1983.

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

[47] T. Yavuz-Kahveci and T. Bultan. Specification, verification, and synthesis of concurrency control components. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 169–179, May 2002.

[48] R. Ziller and K. Schneider. A μ-calculus approach to supervisor synthesis. *Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, pages 132–143, 2003.

[49] R. Ziller and K. Schneider. Combining supervisor synthesis and model checking. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(2):221–362, May 2005.

Appendix A

Java Implementation of Algorithm 2

As in all Java programs, the implementation begins with a `Main` class, as given in Fig. A.1 on the next page. The program takes the IDES supervisor as input using a command line argument. The event-marked code is not needed, since the program doesn't perform the insertion of generation into existing files. After checking for existence of the file, the program creates an `IDESImportManager` that has the responsibility of parsing the IDES file.

The `IDESImportManager` class is given in Fig. A.2 on page 112. An IDES file stores its information in an XML format, so an XML parser is created and used to read information. Only event and transition information is important for the purposes of this algorithm; much of the file is layout information for the IDES interface and is not relevant here. A `Transition` object is created for each transition, and an `Event` object is created for each event. Those classes are listed in Fig. A.3 on page 116 for the `Transition` class, and Fig. A.4 on page 117 for the `Event` class.

```
package controlCodeGenerator;

import java.io.*;

public class Main {

  public static File controlMap;
  public static File supervisor;

  public static void main(String[] args) {
    //args[0] is the IDES file for the supervisor in .xmd format
    //create file object for the IDES supervisor
    try {
      supervisor = new File(args[0]);
      if (!supervisor.exists()) {
        throw new Exception();
      }
    }
    catch (Exception e) {
      String output = "Error opening file!\r\n"}
        + "\r\nTerminating program...");
      System.out.println(output);
      System.exit(-1);
    }

    //read information from IDES file
    IDESImportManager idesFile = new IDESImportManager(args[0]);
    idesFile.analyze();

    //now build the ControlMap from the implicit supervisor
    idesFile.buildControlMap();

    //generate the concurrent code
    CodeGenerator generator = new CodeGenerator(idesFile);
    generator.buildCode();
  }
}
```

Figure A.1: The `Main` class in the Java implementation of Algorithm 2.

```
package controlCodeGenerator;

import java.util.Vector;
import java.util.HashSet;
import java.util.Iterator;
import java.io.*;
import javax.xml.parsers.*;
import org.xml.sax.*;

public class IDESImportManager {

  HashSet<Event> events;
  HashSet<Event> controllableEvents;
  Vector<Transition> transitions;
  ControlMap controlMap;
  String fileName;

  //creates the FileReader
  public IDESImportManager(String fileName) {
    this.fileName = fileName;
    events = new HashSet<Event>();
    controllableEvents = new HashSet<Event>();
    transitions = new Vector<Transition>();
  }

  protected void addEvent(Event e) {
    events.add(e);
    if (e.getControllable())
      controllableEvents.add(e);
    System.out.println(e);
  }

  protected void addTransition(Transition t) {
    transitions.add(t);
    System.out.println(t);
  }
```

Figure A.2: The `IDESImportManager` class parses an inputted IDES supervisor.

```
//get Methods
public HashSet<Event> getControllableEvents() {
  return controllableEvents;
}
public ControlMap getControlMap() {
  return controlMap;
}
public HashSet<Event> getEvents() {
  return events;
}
public Vector<Transition> getTransitions() {
  return transitions;
}

public void buildControlMap() {
  controlMap = new ControlMap(controllableEvents);
  for (Iterator<Transition> it = transitions.iterator();
       it.hasNext(); ) {
    controlMap.updateControlMap(it.next());
  }
  controlMap.display();
}

public Event getEventById(int id) {
  Iterator<Event> it = events.iterator();
  try {
    while(it.hasNext()) {
      Event e = it.next();
      if (e.id == id) {
        return e;
      }
    }
    throw new Exception();
  }
  catch (Exception e) {
    System.out.println("Error - transition ID inconsistent!");
```

Figure A.2 continued.

```
    System.exit(1);
    return null;
  }
}

public void analyze() {

  //set up an xml parser for the IDES file
  SAXParserFactory parserFactory = SAXParserFactory.newInstance();
  parserFactory.setValidating(false);
  try {
    SAXParser parser = parserFactory.newSAXParser();
    parser.parse(new File(fileName), new IDESHandler(this));
  }
  catch (Exception e) {
    System.out.println(e.getMessage());
    System.exit(1);
  }
}

private class IDESHandler extends
             org.xml.sax.helpers.DefaultHandler {

  StringBuffer accumulator = new StringBuffer();
  IDESImportManager parent;
  String type = "none";
  //data accumlators
  String name;
  int id;
  int source;
  int target;
  boolean controllable;

  public IDESHandler(IDESImportManager reader){
    parent = reader;
  }
```

Figure A.2 continued.

```
    public void startElement(String uri, String localName,
                             String qname, Attributes attr) {
      if (qname.equals("event")){
        type = "event";
        id = Integer.parseInt(attr.getValue(0));
      }
      if (qname.equals("controllable") && type.equals("event")){
        controllable = true;
      }
      if (qname.equals("name")&& type.equals("event")){
        type = "eventname";
      }
      if (qname.equals("transition") && !type.equals("ignore")){
        source = Integer.parseInt(attr.getValue(1));
        target = Integer.parseInt(attr.getValue(2));
        id = Integer.parseInt(attr.getValue(3));
        Event eventMatch = parent.getEventById(id);
        parent.addTransition(new Transition(eventMatch,
                             source, target));
      }
    }
    public void endElement(String uri, String localName,
                           String qname) {
      if (qname.equals("data")) {
        type = "ignore";
      }
    }
    public void characters(char[] ch, int start, int length) {
      if (type.equals("eventname")){
        name = new String(ch, start, length);
        parent.addEvent(new Event(name, id, controllable));
        type = "none";
        controllable = false;
      }
    }
  }
}
```

Figure A.2 continued.

```
package controlCodeGenerator;

//a simple class to store transition information
public class Transition {
  //tracks the highest numbered state
  static int maxStates = 0;
  Event event;
  int source;
  int target;
  //constructor
  public Transition(Event event, int source, int target) {
    this.event = event;
    this.source = source;
    this.target = target;
    if (source > maxStates)
      maxStates = source;
    if (target > maxStates)
      maxStates = target;
  }
  //get methods
  public Event getEvent() {
    return event;
  }
  public int getSource() {
    return source;
  }
  public int getTarget() {
    return target;
  }
  public static int getNumStates() {
    return maxStates;
  }
  public String toString() {
    return ("Transition - source: " + this.source + ", target: "
        + this.target + ", event name: " + this.event.getName());
  }
}
```

Figure A.3: The `Transition` class, used to store transition information.

```
package controlCodeGenerator;

public class Event {

  String name;
  int id;
  boolean controllable;

  public Event(String newName, int id, boolean controllable) {
    //must clean up name so that it only contains letters and numbers
    char[] nameCharArray = newName.toCharArray();
    String cleanName = "";
    for (int i = 0; i < nameCharArray.length; i++) {
      char current = nameCharArray[i];
      if ((current >= 48 && current <= 57) ||
        (current >= 65 && current <= 90) ||
        (current >= 97 && current <= 122)) {
        cleanName+=current;
      }
    }

    this.name = cleanName;
    this.id = id;
    this.controllable = controllable;
  }

  //get methods
  public String getName() {
    return name;
  }
  public int getId() {
    return id;
  }
  public boolean getControllable() {
    return controllable;
  }
```

Figure A.4: The `Event` class, used to store event information.

```
  public String toString() {
    return ("Event - name: " + this.name + ", id: "
        + this.id + ", controllable: " + this.controllable);
  }

  public boolean equals(Event e){
    return e.getName().equals(this.name);
  }
}
```

Figure A.4 continued.

Once the `IDESImportManager` has run its course, the main method proceeds to build the change map through the use of the `control map`. The control map is a DES formalization that lists the events disabled by the supervisor at each state. The supervisor is assumed to be implicit, so any controllable event not appearing as a transition from a state is considered to be disabled. The `ControlMap` class, shown in A.5 provides the capabilities to build a control map, and also acts as an object that contains a control map. The change map is never formally constructed. Instead, when the change map is needed while constructing the `UpdateSupervisorState` method, the control map is referenced to build just the section of the change map used at that point. During the execution of the program the full set of operations needed to construct the complete change map are performed, albeit in a piecemeal fashion. It would work equally well to construct the change map in full at this point, but instead the operations are broken up and performed later. Regardless, the change map is used exactly as described in Algorithm 2.

```
package controlCodeGenerator;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.Iterator;

public class ControlMap {

  ArrayList<StateControlMap> stateControlMaps;
  HashSet<Event> controllableEvents;

  public ControlMap(HashSet<Event> controllableEvents){
    stateControlMaps = new ArrayList<StateControlMap>();
    this.controllableEvents = controllableEvents;
  }

  public void updateControlMap(Transition trans) {
    int id = trans.getSource();
    StateControlMap currentState;
    try {
      currentState = stateControlMaps.get(id);
    }
    catch (java.lang.IndexOutOfBoundsException e) {
      stateControlMaps.add(id, new StateControlMap(id,
                           controllableEvents));
      currentState = stateControlMaps.get(id);
    }
    currentState.removeEvent(trans.getEvent());
  }

  public HashSet<Event> getDisabledEventByID(int stateID) {
    for (Iterator<StateControlMap> it = stateControlMaps.iterator();
         it.hasNext();){
      StateControlMap map = it.next();
      if (map.getID()==stateID)
        return map.getDisabledEvents();
```

Figure A.5: The `ControlMap` class builds the implicit control map for a DES supervisor.

```
    }
    return null;
  }

  public void display() {
    for (Iterator<StateControlMap> it = stateControlMaps.iterator();
         it.hasNext();) {
      StateControlMap currentState = it.next();
      System.out.print(currentState);
    }
  }

  private class StateControlMap {

    private int stateID;
    private HashSet<Event> disabledEvents;

    public StateControlMap(int id, HashSet<Event> events) {
      stateID = id;
      disabledEvents = new HashSet<Event>(events);
    }

    public int getID() {
      return stateID;
    }

    public HashSet<Event> getDisabledEvents() {
      return disabledEvents;
    }

    public void removeEvent(Event e) {
      disabledEvents.remove(e);
    }

    public String toString() {
      String output = ("ID: " + stateID + ", disabled events: ");
```

Figure A.5 continued.

```
      for (Iterator<Event> iths = disabledEvents.iterator();
           iths.hasNext(); ) {
        output += iths.next().getName()+",";
      }
      return output + "\r\n";
    }
  }
}
```

Figure A.5 continued.

Now that the supervisor has been analyzed, and the control map is prepared, code may be generated. The `CodeGenerator` class, listed in Fig. A.6 on the next page, handles the creation of code. It follows Algorithm 2 in function. In this incarnation, it simply outputs the constructed `Supervisor` class to the console.

```
package controlCodeGenerator;

import java.util.HashSet;
import java.util.Vector;
import java.util.Iterator;

public class CodeGenerator {

  IDESImportManager idesInfo;
  public CodeGenerator(IDESImportManager idesFile) {
    idesInfo = idesFile;
  }
    public void buildCode() {
  //the proper package name is unknown as no code is read in.
    System.out.println("package controlCodeGenerator;\r\n");
    System.out.println("import java.util.concurrent.*;\r\n");
    System.out.println("public class Supervisor {\r\n");

    //define a semaphore for each controllable event
    HashSet<Event> controllableEvents =
                   idesInfo.getControllableEvents();
    for (Iterator<Event> it = controllableEvents.iterator();
         it.hasNext(); ) {
      Event e = it.next();
      System.out.println("  static Semaphore " + e.name +";");
    }
    System.out.println("  static int supervisorState;");
    System.out.println("");

    //now create the initialization method
    System.out.println("  public static void init() {\r\n");
    //need to set up the initial state
    HashSet<Event> initialDisEvents = idesInfo.getControlMap().
                                      getDisabledEventByID(0);
    for (Iterator<Event> it = controllableEvents.iterator();
          it.hasNext(); ) {
      Event e = it.next();
```

Figure A.6: The code listings for the `CodeGenerator` class.

```
  System.out.print("    Supervisor." + e.name
                   + " = new Semaphore(");
  if (initialDisEvents.contains(e)) {
    System.out.println("0);");
  }
  else {
    System.out.println("1);");
  }
}

//initialize state tracker
System.out.println("    Supervisor.supervisorState = 0;");
System.out.println("  }\r\n");

//insert the synchronized state change method
System.out.println("  public static synchronized boolean
       observeAndReact(String event, Semaphore eventBlocker) {");
System.out.println("    if (!(eventBlocker == null)) {");
System.out.println("      if (!eventBlocker.tryAcquire()) {");
System.out.println("        return false;");
System.out.println("      }");
System.out.println("      eventBlocker.release();");
System.out.println("    }");
System.out.println("    updateSupervisorState(event);");
System.out.println("    return true;");
System.out.println("  }\r\n");

//insert the updateSupervisorState method
System.out.println("  private static void updateSupervisorState
                   (String event) {\r\n");

HashSet<Event> events = idesInfo.getEvents();

boolean doneFirst = false;
for (Iterator<Event> it = events.iterator(); it.hasNext(); ) {
  Event e = it.next();
  if (doneFirst) {
```

Figure A.6 continued.

```
      System.out.println("    else if (event.equals("
                         + e.getName() + ")) {");
    }
    else {
      System.out.println("    if (event.equals("
                         + e.getName() + ")) {");
      doneFirst = true;
    }
  //now we have to change the state for each transition
  //  using the event as the base
    System.out.println(stateChangesByEvent(e) + "\r\n    }");
  }
  System.out.println("  }");
  System.out.println("}");
}

private String stateChangesByEvent(Event e) {
  String output = "      switch(Supervisor.supervisorState) {\r\n";
  Vector<Transition> transitions = idesInfo.getTransitions();
  for (Iterator<Transition> it = transitions.iterator();
       it.hasNext(); ) {
    Transition t = it.next();
    if (t.getEvent() == e) {
      //this event qualifies, time to create code for it
      output+= "        case(" + t.getSource() + "):\r\n";
      HashSet<Event> enabled = new HashSet<Event>();
      HashSet<Event> disabled = new HashSet<Event>();
      HashSet<Event> sourceEvents = idesInfo.getControlMap().
                          getDisabledEventByID(t.getSource());
      HashSet<Event> targetEvents = idesInfo.getControlMap().
                          getDisabledEventByID(t.getTarget());

      for (Iterator<Event> seit = sourceEvents.
                    iterator(); seit.hasNext(); ){
        Event sourceDisabledEvent = seit.next();
        if (null!=targetEvents) {
          if (!targetEvents.contains(sourceDisabledEvent)) {
```

Figure A.6 continued.

```
            enabled.add(sourceDisabledEvent);
          }
        }
      }
      if (null!=targetEvents) {
        for (Iterator<Event> teit = targetEvents.iterator();
             teit.hasNext(); ){
          Event targetDisabledEvent = teit.next();
          if (!sourceEvents.contains(targetDisabledEvent))
            disabled.add(targetDisabledEvent);
        }
      }
      if (!enabled.isEmpty()) {
        for (Iterator<Event> eit = enabled.iterator();
             eit.hasNext(); ){
          Event eventToEnable = eit.next();
          output+="          Supervisor." + eventToEnable.getName()
            + "." + "release();\r\n";
        }
      }
      if (!disabled.isEmpty()) {
        for (Iterator<Event> dit = disabled.iterator();
             dit.hasNext(); ){
          Event eventToDisable = dit.next();
          output+="          Supervisor." + eventToDisable.getName()
            + "." + "acquireUninterruptibly();\r\n";
        }
      }
      output+= "          Supervisor.supervisorState = "
        + t.getTarget() + ";\r\n";
      output+= "          break;\r\n";
    }
  }
  output+= "      }";
  return output;
 }
}
```

Figure A.6 continued.

Appendix B

Transfer Line in Java

Complete work for the transfer-line problem is given here. This includes Java code listings, CFGs and FSAs, as well as generated code in the concurrency-controlled output. The problem is covered in full in Chapter 5.

B.1 Code Listings

The first set of figures provides code listings for each of the classes used in the transfer-line example. Rather than give the original code that acted as the input for the process, the listings given include event markings. First is the `Main` class in Fig. B.1. This class instantiates the buffers and threads, starts the threads, then terminates. The `Part` class in Fig. B.2 is instantiated for each part to be transferred through the system. The `Buffer` class is given in B.6, while the three active components `M1`, `M2`, and `TU` are given in Fig. B.3 on page 128, Fig. B.4 on page 129 and Fig. B.5 on page 130, respectively.

```
package transferLine;
public class Main {
  final static int BUFFER_1_SIZE = 3;
  final static int BUFFER_2_SIZE = 1;
  //arbitrary reject chance of 1/4
  final static double TEST_REJECT_CHANCE = 0.25;
  public static void main(String[] args) {
    // build the system
    Buffer b1 = new Buffer(BUFFER_1_SIZE, "Buffer1");
    Buffer b2 = new Buffer(BUFFER_2_SIZE, "Buffer2");
    Thread m1 = new Machine1(b1);
    Thread m2 = new Machine2(b1,b2);
    Thread tu = new TestUnit(b2,b1,TEST_REJECT_CHANCE);
    //start the threads
    m1.start();
    m2.start();
    tu.start();
  }
}
```

Figure B.1: Transfer-Line: `Main` class.

```
package transferLine;

//each part is an object that is instantiated and passed
//between threads
public class Part {

  public Part() {
    //no instance variables
  }
}
```

Figure B.2: Transfer-Line: `Part` class.

```
package transferLine;

//creates new parts and puts them into the first buffer
public class Machine1 extends Thread {

  //the target for created  parts
  Buffer target;

  public Machine1(Buffer target) {
    new Thread();
    this.target = target;
    this.setName("Machine1");
  }

  public void doWork() {
    try {
      Thread.sleep((int)(Math.random()*1000));
    }
    catch (Exception e) {
      //do nothing
    }
  }

  public void run() {
    while (true) {
      //wait for some random period
      doWork();
      //create a new Part
      Part newPart = new Part();
      //event marking: M1addB1-start
      //put it in the target buffer
      System.out.println("Machine1 tries to put a part in " + target);
      target.addPart(newPart);
      //event marking: M1add-B1-finish-u
    } // loop forever
  }
}
```

Figure B.3: Transfer-Line: `Machine1` class.

```
package transferLine;

public class Machine2 extends Thread {

  //the source and target for parts used by this machine
  Buffer source;
  Buffer target;

  public Machine2(Buffer source, Buffer target) {
    new Thread();
    this.source = source;
    this.target = target;
    this.setName("Machine2");
  }

  public void doWork() {
    try {
      Thread.sleep((int)(Math.random()*10));
    }
    catch (Exception e) {
      //do nothing
    }
  }

  public void run() {
    while (true) {
      //get part from source Buffer
      //event marking: M2getB1-start
      System.out.println("Machine2 tries to get a part from "
                         + source);
      Part currentPart = source.removePart();
      //event marking: M2getB1-finish-u
      //we have a part - do some work on it
      doWork();
      //put the part in the next buffer
```

Figure B.4: Transfer-Line: `Machine2` class.

```
      //event marking: M2addB2-start
      System.out.println("Machine2 tries to put a part in "
                         + target);
      target.addPart(currentPart);
      //event marking: M2addB2-finish-u
      //reset machine 2
      doWork();
    } //loop forever
  }
}
```

Figure B.4 continued.

```
package transferLine;

public class TestUnit extends Thread{

  Buffer source;
  Buffer rejectBuffer;
  double rejectionChance;

  public TestUnit(Buffer source, Buffer rejectBuffer,
                           double rejectionChance) {
    new Thread();
    this.source = source;
    this.rejectBuffer = rejectBuffer;
    this.rejectionChance = rejectionChance;
    this.setName("TestUnit");
  }
  public void doWork() {
    try {
      Thread.sleep((int)(Math.random()*10));
    }
    catch (Exception e) {
      //do nothing
    }
  }
```

Figure B.5: Transfer-Line: `TestUnit` class.

```
  public void run() {
    while (true) {
      //get a part from the source
      //event marking: TUgetB2-start
      System.out.println("TestUnit tries to get a part from "
                         + source);
      Part currentPart = source.removePart();
      //event marking: TUgetB2-finish-u

      //test that part for some period of time
      doWork();
      if (Math.random() > rejectionChance) {
        //no good!  send back to rejectBin
        //event marking: TUaddB1-start-u
        System.out.println("TestUnit tries to put a part in "
                           + rejectBuffer);
        rejectBuffer.addPart(currentPart);
        //event marking: TUaddB1-finish-u
      }
    } //loop forever
  }
}
```

Figure B.5 continued.

```
package transferLine;
import java.util.LinkedList;

public class Buffer {

  String name;
  //the maximum number of parts in the buffer
  int size;
  //the current number of parts in the buffer
  int holding;
  LinkedList<Part> partBuffer;

  public Buffer(int size, String name) {
    this.name = name;
    this.size = size;
    holding = 0;
    partBuffer = new LinkedList<Part>();
  }

  public void addPart(Part p) {
    if (holding < size) {
      holding++;
      System.out.println (name + " is holding "
                          + holding + " parts.");
      partBuffer.add(p);
    }
    else {
      System.out.println(name + " overflow!!!");
    }
  }

  public Part removePart() {
    if (holding > 0) {
      holding--;
      System.out.println (name + " is holding "
                          + holding + " parts.");
      return partBuffer.remove();
```

Figure B.6: Transfer-Line: `Buffer` class.

```
    }
    else {
      System.out.println(name + " underflow!!!");
      return null;
    }
  }

  public String toString() {
    return name;
  }
}
```

Figure B.6 continued.

B.2 Marking Implicit Events

The reduced DFA in Fig. 5.7 has an irrelevant event that lies on an implicit path. Event `i9` arises from an `if` statement. To make this explicit, a source transform is used to give the `if` statement an `else` branch. The transformed source for `TU` is given in Fig. B.7.

B.3 Plant

The reduced DFAs for **M1**, **M2**, and **TU** were combined using the shuffle operation. The resulting plant has 32 states and 104 transitions. A transition listing is provided in Fig. B.8 on page 136. State 0 is the initial state, and is the only marked state.

```
package transferLine;

public class TestUnit extends Thread{

  Buffer source;
  Buffer rejectBuffer;
  double rejectionChance;

  public TestUnit(Buffer source, Buffer rejectBuffer,
                          double rejectionChance) {
    new Thread();
    this.source = source;
    this.rejectBuffer = rejectBuffer;
    this.rejectionChance = rejectionChance;
    this.setName("TestUnit");
  }

  public void doWork() {
    try {
      Thread.sleep((int)(Math.random()*10));
    }
    catch (Exception e) {
      //do nothing
    }
  }

  public void run() {
    while (true) {
      //get a part from the source
      //event marking: TUgetB2-start
      System.out.println("TestUnit tries to get a part from "
                         + source);
      Part currentPart = source.removePart();
      //event marking: TUgetB2-finish-u
```

Figure B.7: Transfer-Line: Excerpt of `TestUnit` class showing event `i9` made explicit.

```
      //test that part for some period of time
      doWork();
      if (Math.random() > rejectionChance) {
        //no good!  send back to rejectBin
        //event marking: TUaddB1-start-u
        System.out.println("TestUnit tries to put a part in "
                           + rejectBuffer);
        rejectBuffer.addPart(currentPart);
        //event marking: TUaddB1-finish-u
      }

      //code to make i9 explicit
      else  {
        //event marking: i9
      }
      //end of code making i9 explicit

    } //loop forever
  }
}
```

Figure B.7 continued.

```
Transition: Source="0" Target="1" Event="M2getB1-start"
Transition: Source="0" Target="2" Event="M1addB1-start"
Transition: Source="0" Target="3" Event="TUgetB2-start"
Transition: Source="1" Target="4" Event="M2getB1-finish"
Transition: Source="1" Target="5" Event="M1addB1-start"
Transition: Source="1" Target="6" Event="TUgetB2-start"
Transition: Source="2" Target="5" Event="M2getB1-start"
Transition: Source="2" Target="0" Event="M1addB1-finish"
Transition: Source="2" Target="7" Event="TUgetB2-start"
Transition: Source="3" Target="6" Event="M2getB1-start"
Transition: Source="3" Target="7" Event="M1addB1-start"
Transition: Source="3" Target="8" Event="TUgetB2-finish"
Transition: Source="4" Target="9" Event="M2addB2-start"
Transition: Source="4" Target="10" Event="M1addB1-start"
Transition: Source="4" Target="11" Event="TUgetB2-start"
Transition: Source="5" Target="10" Event="M2getB1-finish"
Transition: Source="5" Target="1" Event="M1addB1-finish"
Transition: Source="5" Target="12" Event="TUgetB2-start"
Transition: Source="6" Target="11" Event="M2getB1-finish"
Transition: Source="6" Target="12" Event="M1addB1-start"
Transition: Source="6" Target="13" Event="TUgetB2-finish"
Transition: Source="7" Target="12" Event="M2getB1-start"
Transition: Source="7" Target="3" Event="M1addB1-finish"
Transition: Source="7" Target="14" Event="TUgetB2-finish"
Transition: Source="8" Target="13" Event="M2getB1-start"
Transition: Source="8" Target="14" Event="M1addB1-start"
Transition: Source="8" Target="15" Event="TUaddB1-start-u"
Transition: Source="8" Target="0" Event="i9"
Transition: Source="9" Target="0" Event="M2addB2-finish"
Transition: Source="9" Target="16" Event="M1addB1-start"
Transition: Source="9" Target="17" Event="TUgetB2-start"
Transition: Source="10" Target="16" Event="M2addB2-start"
Transition: Source="10" Target="4" Event="M1addB1-finish"
Transition: Source="10" Target="18" Event="TUgetB2-start"
Transition: Source="11" Target="17" Event="M2addB2-start"
Transition: Source="11" Target="18" Event="M1addB1-start"
Transition: Source="11" Target="19" Event="TUgetB2-finish"
```

Figure B.8: Transfer-Line: Transitition listing for the plant.

```
Transition: Source="12" Target="18" Event="M2getB1-finish"
Transition: Source="12" Target="6" Event="M1addB1-finish"
Transition: Source="12" Target="20" Event="TUgetB2-finish"
Transition: Source="13" Target="19" Event="M2getB1-finish"
Transition: Source="13" Target="20" Event="M1addB1-start"
Transition: Source="13" Target="21" Event="TUaddB1-start-u"
Transition: Source="13" Target="1" Event="i9"
Transition: Source="14" Target="20" Event="M2getB1-start"
Transition: Source="14" Target="8" Event="M1addB1-finish"
Transition: Source="14" Target="22" Event="TUaddB1-start-u"
Transition: Source="14" Target="2" Event="i9"
Transition: Source="15" Target="21" Event="M2getB1-start"
Transition: Source="15" Target="22" Event="M1addB1-start"
Transition: Source="15" Target="0" Event="TUaddB1-finish-u"
Transition: Source="16" Target="2" Event="M2addB2-finish"
Transition: Source="16" Target="9" Event="M1addB1-finish"
Transition: Source="16" Target="23" Event="TUgetB2-start"
Transition: Source="17" Target="3" Event="M2addB2-finish"
Transition: Source="17" Target="23" Event="M1addB1-start"
Transition: Source="17" Target="24" Event="TUgetB2-finish"
Transition: Source="18" Target="23" Event="M2addB2-start"
Transition: Source="18" Target="11" Event="M1addB1-finish"
Transition: Source="18" Target="25" Event="TUgetB2-finish"
Transition: Source="19" Target="24" Event="M2addB2-start"
Transition: Source="19" Target="25" Event="M1addB1-start"
Transition: Source="19" Target="26" Event="TUaddB1-start-u"
Transition: Source="19" Target="4" Event="i9"
Transition: Source="20" Target="25" Event="M2getB1-finish"
Transition: Source="20" Target="13" Event="M1addB1-finish"
Transition: Source="20" Target="27" Event="TUaddB1-start-u"
Transition: Source="20" Target="5" Event="i9"
Transition: Source="21" Target="26" Event="M2getB1-finish"
Transition: Source="21" Target="27" Event="M1addB1-start"
Transition: Source="21" Target="1" Event="TUaddB1-finish-u"
Transition: Source="22" Target="27" Event="M2getB1-start"
Transition: Source="22" Target="15" Event="M1addB1-finish"
Transition: Source="22" Target="2" Event="TUaddB1-finish-u"
```

Figure B.8 continued.

```
Transition: Source="23" Target="7" Event="M2addB2-finish"
Transition: Source="23" Target="17" Event="M1addB1-finish"
Transition: Source="23" Target="28" Event="TUgetB2-finish"
Transition: Source="24" Target="8" Event="M2addB2-finish"
Transition: Source="24" Target="28" Event="M1addB1-start"
Transition: Source="24" Target="29" Event="TUaddB1-start-u"
Transition: Source="24" Target="9" Event="i9"
Transition: Source="25" Target="28" Event="M2addB2-start"
Transition: Source="25" Target="19" Event="M1addB1-finish"
Transition: Source="25" Target="30" Event="TUaddB1-start-u"
Transition: Source="25" Target="10" Event="i9"
Transition: Source="26" Target="29" Event="M2addB2-start"
Transition: Source="26" Target="30" Event="M1addB1-start"
Transition: Source="26" Target="4" Event="TUaddB1-finish-u"
Transition: Source="27" Target="30" Event="M2getB1-finish"
Transition: Source="27" Target="21" Event="M1addB1-finish"
Transition: Source="27" Target="5" Event="TUaddB1-finish-u"
Transition: Source="28" Target="14" Event="M2addB2-finish"
Transition: Source="28" Target="24" Event="M1addB1-finish"
Transition: Source="28" Target="31" Event="TUaddB1-start-u"
Transition: Source="28" Target="16" Event="i9"
Transition: Source="29" Target="15" Event="M2addB2-finish"
Transition: Source="29" Target="31" Event="M1addB1-start"
Transition: Source="29" Target="9" Event="TUaddB1-finish-u"
Transition: Source="30" Target="31" Event="M2addB2-start"
Transition: Source="30" Target="26" Event="M1addB1-finish"
Transition: Source="30" Target="10" Event="TUaddB1-finish-u"
Transition: Source="31" Target="22" Event="M2addB2-finish"
Transition: Source="31" Target="29" Event="M1addB1-finish"
Transition: Source="31" Target="16" Event="TUaddB1-finish-u"
```

Figure B.8 continued.

B.4 Specifications

The monolithic specification was built. It contains 52 states and 176 transitions. A transition listing is given in Fig. B.9 below. State 0 is the initial state, and is the only marked state.

```
Transition: Source="0" Target="1" Event="M1addB1-start"
Transition: Source="0" Target="2" Event="TUaddB1-start-u"
Transition: Source="0" Target="3" Event="M2addB2-start"
Transition: Source="0" Target="0" Event="i9"
Transition: Source="1" Target="4" Event="M1addB1-finish"
Transition: Source="1" Target="5" Event="M2addB2-start"
Transition: Source="1" Target="1" Event="i9"
Transition: Source="2" Target="4" Event="TUaddB1-finish-u"
Transition: Source="2" Target="6" Event="M2addB2-start"
Transition: Source="2" Target="2" Event="i9"
Transition: Source="3" Target="5" Event="M1addB1-start"
Transition: Source="3" Target="6" Event="TUaddB1-start-u"
Transition: Source="3" Target="7" Event="M2addB2-finish"
Transition: Source="3" Target="3" Event="i9"
Transition: Source="4" Target="8" Event="M1addB1-start"
Transition: Source="4" Target="9" Event="M2getB1-start"
Transition: Source="4" Target="10" Event="TUaddB1-start-u"
Transition: Source="4" Target="11" Event="M2addB2-start"
Transition: Source="4" Target="4" Event="i9"
Transition: Source="5" Target="11" Event="M1addB1-finish"
Transition: Source="5" Target="12" Event="M2addB2-finish"
Transition: Source="5" Target="5" Event="i9"
Transition: Source="6" Target="11" Event="TUaddB1-finish-u"
Transition: Source="6" Target="13" Event="M2addB2-finish"
Transition: Source="6" Target="6" Event="i9"
Transition: Source="7" Target="12" Event="M1addB1-start"
Transition: Source="7" Target="13" Event="TUaddB1-start-u"
Transition: Source="7" Target="14" Event="TUgetB2-start"
```

Figure B.9: Transfer-Line: Transition listing for the monolithic specification.

```
Transition: Source="7" Target="7" Event="i9"
Transition: Source="8" Target="15" Event="M1addB1-finish"
Transition: Source="8" Target="16" Event="M2addB2-start"
Transition: Source="8" Target="8" Event="i9"
Transition: Source="9" Target="0" Event="M2getB1-finish"
Transition: Source="9" Target="17" Event="M2addB2-start"
Transition: Source="9" Target="9" Event="i9"
Transition: Source="10" Target="15" Event="TUaddB1-finish-u"
Transition: Source="10" Target="18" Event="M2addB2-start"
Transition: Source="10" Target="10" Event="i9"
Transition: Source="11" Target="16" Event="M1addB1-start"
Transition: Source="11" Target="17" Event="M2getB1-start"
Transition: Source="11" Target="18" Event="TUaddB1-start-u"
Transition: Source="11" Target="19" Event="M2addB2-finish"
Transition: Source="11" Target="11" Event="i9"
Transition: Source="12" Target="19" Event="M1addB1-finish"
Transition: Source="12" Target="20" Event="TUgetB2-start"
Transition: Source="12" Target="12" Event="i9"
Transition: Source="13" Target="19" Event="TUaddB1-finish-u"
Transition: Source="13" Target="21" Event="TUgetB2-start"
Transition: Source="13" Target="13" Event="i9"
Transition: Source="14" Target="20" Event="M1addB1-start"
Transition: Source="14" Target="21" Event="TUaddB1-start-u"
Transition: Source="14" Target="0" Event="TUgetB2-finish"
Transition: Source="14" Target="14" Event="i9"
Transition: Source="15" Target="22" Event="M1addB1-start"
Transition: Source="15" Target="23" Event="M2getB1-start"
Transition: Source="15" Target="24" Event="TUaddB1-start-u"
Transition: Source="15" Target="25" Event="M2addB2-start"
Transition: Source="15" Target="15" Event="i9"
Transition: Source="16" Target="25" Event="M1addB1-finish"
Transition: Source="16" Target="26" Event="M2addB2-finish"
Transition: Source="16" Target="16" Event="i9"
Transition: Source="17" Target="3" Event="M2getB1-finish"
Transition: Source="17" Target="27" Event="M2addB2-finish"
Transition: Source="17" Target="17" Event="i9"
Transition: Source="18" Target="25" Event="TUaddB1-finish-u"
```

Figure B.9 continued.

```
Transition: Source="18" Target="28" Event="M2addB2-finish"
Transition: Source="18" Target="18" Event="i9"
Transition: Source="19" Target="26" Event="M1addB1-start"
Transition: Source="19" Target="27" Event="M2getB1-start"
Transition: Source="19" Target="28" Event="TUaddB1-start-u"
Transition: Source="19" Target="29" Event="TUgetB2-start"
Transition: Source="19" Target="19" Event="i9"
Transition: Source="20" Target="29" Event="M1addB1-finish"
Transition: Source="20" Target="1" Event="TUgetB2-finish"
Transition: Source="20" Target="20" Event="i9"
Transition: Source="21" Target="29" Event="TUaddB1-finish-u"
Transition: Source="21" Target="2" Event="TUgetB2-finish"
Transition: Source="21" Target="21" Event="i9"
Transition: Source="22" Target="30" Event="M1addB1-finish"
Transition: Source="22" Target="31" Event="M2addB2-start"
Transition: Source="22" Target="22" Event="i9"
Transition: Source="23" Target="4" Event="M2getB1-finish"
Transition: Source="23" Target="32" Event="M2addB2-start"
Transition: Source="23" Target="23" Event="i9"
Transition: Source="24" Target="30" Event="TUaddB1-finish-u"
Transition: Source="24" Target="33" Event="M2addB2-start"
Transition: Source="24" Target="24" Event="i9"
Transition: Source="25" Target="31" Event="M1addB1-start"
Transition: Source="25" Target="32" Event="M2getB1-start"
Transition: Source="25" Target="33" Event="TUaddB1-start-u"
Transition: Source="25" Target="34" Event="M2addB2-finish"
Transition: Source="25" Target="25" Event="i9"
Transition: Source="26" Target="34" Event="M1addB1-finish"
Transition: Source="26" Target="35" Event="TUgetB2-start"
Transition: Source="26" Target="26" Event="i9"
Transition: Source="27" Target="7" Event="M2getB1-finish"
Transition: Source="27" Target="36" Event="TUgetB2-start"
Transition: Source="27" Target="27" Event="i9"
Transition: Source="28" Target="34" Event="TUaddB1-finish-u"
Transition: Source="28" Target="37" Event="TUgetB2-start"
Transition: Source="28" Target="28" Event="i9"
Transition: Source="29" Target="35" Event="M1addB1-start"
```

Figure B.9 continued.

```
Transition: Source="29" Target="36" Event="M2getB1-start"
Transition: Source="29" Target="37" Event="TUaddB1-start-u"
Transition: Source="29" Target="4" Event="TUgetB2-finish"
Transition: Source="29" Target="29" Event="i9"
Transition: Source="30" Target="38" Event="M2getB1-start"
Transition: Source="30" Target="39" Event="M2addB2-start"
Transition: Source="30" Target="30" Event="i9"
Transition: Source="31" Target="39" Event="M1addB1-finish"
Transition: Source="31" Target="40" Event="M2addB2-finish"
Transition: Source="31" Target="31" Event="i9"
Transition: Source="32" Target="11" Event="M2getB1-finish"
Transition: Source="32" Target="41" Event="M2addB2-finish"
Transition: Source="32" Target="32" Event="i9"
Transition: Source="33" Target="39" Event="TUaddB1-finish-u"
Transition: Source="33" Target="42" Event="M2addB2-finish"
Transition: Source="33" Target="33" Event="i9"
Transition: Source="34" Target="40" Event="M1addB1-start"
Transition: Source="34" Target="41" Event="M2getB1-start"
Transition: Source="34" Target="42" Event="TUaddB1-start-u"
Transition: Source="34" Target="43" Event="TUgetB2-start"
Transition: Source="34" Target="34" Event="i9"
Transition: Source="35" Target="43" Event="M1addB1-finish"
Transition: Source="35" Target="8" Event="TUgetB2-finish"
Transition: Source="35" Target="35" Event="i9"
Transition: Source="36" Target="14" Event="M2getB1-finish"
Transition: Source="36" Target="9" Event="TUgetB2-finish"
Transition: Source="36" Target="36" Event="i9"
Transition: Source="37" Target="43" Event="TUaddB1-finish-u"
Transition: Source="37" Target="10" Event="TUgetB2-finish"
Transition: Source="37" Target="37" Event="i9"
Transition: Source="38" Target="15" Event="M2getB1-finish"
Transition: Source="38" Target="44" Event="M2addB2-start"
Transition: Source="38" Target="38" Event="i9"
Transition: Source="39" Target="44" Event="M2getB1-start"
Transition: Source="39" Target="45" Event="M2addB2-finish"
Transition: Source="39" Target="39" Event="i9"
Transition: Source="40" Target="45" Event="M1addB1-finish"
```

Figure B.9 continued.

```
Transition: Source="40" Target="46" Event="TUgetB2-start"
Transition: Source="40" Target="40" Event="i9"
Transition: Source="41" Target="19" Event="M2getB1-finish"
Transition: Source="41" Target="47" Event="TUgetB2-start"
Transition: Source="41" Target="41" Event="i9"
Transition: Source="42" Target="45" Event="TUaddB1-finish-u"
Transition: Source="42" Target="48" Event="TUgetB2-start"
Transition: Source="42" Target="42" Event="i9"
Transition: Source="43" Target="46" Event="M1addB1-start"
Transition: Source="43" Target="47" Event="M2getB1-start"
Transition: Source="43" Target="48" Event="TUaddB1-start-u"
Transition: Source="43" Target="15" Event="TUgetB2-finish"
Transition: Source="43" Target="43" Event="i9"
Transition: Source="44" Target="25" Event="M2getB1-finish"
Transition: Source="44" Target="49" Event="M2addB2-finish"
Transition: Source="44" Target="44" Event="i9"
Transition: Source="45" Target="49" Event="M2getB1-start"
Transition: Source="45" Target="50" Event="TUgetB2-start"
Transition: Source="45" Target="45" Event="i9"
Transition: Source="46" Target="50" Event="M1addB1-finish"
Transition: Source="46" Target="22" Event="TUgetB2-finish"
Transition: Source="46" Target="46" Event="i9"
Transition: Source="47" Target="29" Event="M2getB1-finish"
Transition: Source="47" Target="23" Event="TUgetB2-finish"
Transition: Source="47" Target="47" Event="i9"
Transition: Source="48" Target="50" Event="TUaddB1-finish-u"
Transition: Source="48" Target="24" Event="TUgetB2-finish"
Transition: Source="48" Target="48" Event="i9"
Transition: Source="49" Target="34" Event="M2getB1-finish"
Transition: Source="49" Target="51" Event="TUgetB2-start"
Transition: Source="49" Target="49" Event="i9"
Transition: Source="50" Target="51" Event="M2getB1-start"
Transition: Source="50" Target="30" Event="TUgetB2-finish"
Transition: Source="50" Target="50" Event="i9"
Transition: Source="51" Target="43" Event="M2getB1-finish"
Transition: Source="51" Target="38" Event="TUgetB2-finish"
Transition: Source="51" Target="51" Event="i9"
```

Figure B.9 continued.

B.5 Supervisor

Using the plant and the monolithic specification, a supervisor was synthesized in TCT using the `SUPCON` operation. It has 69 states and 114 transitions. State 0 is the initial state, and is the only marked state. The transition structure is given in Fig. B.10.

```
Transition: Source="0" Target="1" Event="M1addB1-start"
Transition: Source="1" Target="2" Event="M1addB1-finish"
Transition: Source="2" Target="3" Event="M1addB1-start"
Transition: Source="2" Target="4" Event="M2getB1-start"
Transition: Source="3" Target="5" Event="M1addB1-finish"
Transition: Source="4" Target="6" Event="M2getB1-finish"
Transition: Source="5" Target="7" Event="M1addB1-start"
Transition: Source="5" Target="8" Event="M2getB1-start"
Transition: Source="6" Target="9" Event="M1addB1-start"
Transition: Source="6" Target="10" Event="M2addB2-start"
Transition: Source="7" Target="11" Event="M1addB1-finish"
Transition: Source="8" Target="12" Event="M2getB1-finish"
Transition: Source="9" Target="13" Event="M2addB2-start"
Transition: Source="9" Target="12" Event="M1addB1-finish"
Transition: Source="10" Target="13" Event="M1addB1-start"
Transition: Source="10" Target="14" Event="M2addB2-finish"
Transition: Source="11" Target="15" Event="M2getB1-start"
Transition: Source="12" Target="16" Event="M1addB1-start"
Transition: Source="12" Target="17" Event="M2addB2-start"
Transition: Source="13" Target="18" Event="M2addB2-finish"
Transition: Source="13" Target="17" Event="M1addB1-finish"
Transition: Source="14" Target="19" Event="TUgetB2-start"
Transition: Source="14" Target="18" Event="M1addB1-start"
Transition: Source="15" Target="20" Event="M2getB1-finish"
Transition: Source="16" Target="21" Event="M2addB2-start"
Transition: Source="16" Target="20" Event="M1addB1-finish"
Transition: Source="17" Target="21" Event="M1addB1-start"
Transition: Source="17" Target="22" Event="M2addB2-finish"
```

Figure B.10: Transfer-Line: Monolithic supervisor transition listing.

```
Transition: Source="18" Target="22" Event="M1addB1-finish"
Transition: Source="19" Target="23" Event="TUgetB2-finish"
Transition: Source="20" Target="24" Event="M1addB1-start"
Transition: Source="20" Target="25" Event="M2addB2-start"
Transition: Source="21" Target="26" Event="M2addB2-finish"
Transition: Source="21" Target="25" Event="M1addB1-finish"
Transition: Source="22" Target="27" Event="TUgetB2-start"
Transition: Source="22" Target="26" Event="M1addB1-start"
Transition: Source="22" Target="28" Event="M2getB1-start"
Transition: Source="23" Target="29" Event="TUaddB1-start-u"
Transition: Source="23" Target="0" Event="i9"
Transition: Source="24" Target="30" Event="M2addB2-start"
Transition: Source="24" Target="31" Event="M1addB1-finish"
Transition: Source="25" Target="30" Event="M1addB1-start"
Transition: Source="25" Target="32" Event="M2addB2-finish"
Transition: Source="26" Target="32" Event="M1addB1-finish"
Transition: Source="27" Target="33" Event="TUgetB2-finish"
Transition: Source="28" Target="34" Event="M2getB1-finish"
Transition: Source="29" Target="2" Event="TUaddB1-finish-u"
Transition: Source="30" Target="35" Event="M2addB2-finish"
Transition: Source="30" Target="36" Event="M1addB1-finish"
Transition: Source="31" Target="36" Event="M2addB2-start"
Transition: Source="32" Target="37" Event="TUgetB2-start"
Transition: Source="32" Target="35" Event="M1addB1-start"
Transition: Source="32" Target="38" Event="M2getB1-start"
Transition: Source="33" Target="39" Event="TUaddB1-start-u"
Transition: Source="33" Target="2" Event="i9"
Transition: Source="34" Target="40" Event="TUgetB2-start"
Transition: Source="34" Target="41" Event="M1addB1-start"
Transition: Source="35" Target="42" Event="M1addB1-finish"
Transition: Source="36" Target="42" Event="M2addB2-finish"
Transition: Source="37" Target="43" Event="TUgetB2-finish"
Transition: Source="38" Target="44" Event="M2getB1-finish"
Transition: Source="39" Target="5" Event="TUaddB1-finish-u"
Transition: Source="40" Target="45" Event="TUgetB2-finish"
Transition: Source="41" Target="44" Event="M1addB1-finish"
Transition: Source="42" Target="46" Event="M2getB1-start"
```

Figure B.10 continued.

```
Transition: Source="43" Target="47" Event="TUaddB1-start-u"
Transition: Source="43" Target="5" Event="i9"
Transition: Source="44" Target="48" Event="TUgetB2-start"
Transition: Source="44" Target="49" Event="M1addB1-start"
Transition: Source="45" Target="50" Event="TUaddB1-start-u"
Transition: Source="45" Target="6" Event="i9"
Transition: Source="45" Target="51" Event="M2addB2-start"
Transition: Source="46" Target="52" Event="M2getB1-finish"
Transition: Source="47" Target="11" Event="TUaddB1-finish-u"
Transition: Source="48" Target="53" Event="TUgetB2-finish"
Transition: Source="49" Target="52" Event="M1addB1-finish"
Transition: Source="50" Target="12" Event="TUaddB1-finish-u"
Transition: Source="50" Target="54" Event="M2addB2-start"
Transition: Source="51" Target="54" Event="TUaddB1-start-u"
Transition: Source="51" Target="10" Event="i9"
Transition: Source="51" Target="55" Event="M2addB2-finish"
Transition: Source="52" Target="56" Event="TUgetB2-start"
Transition: Source="53" Target="57" Event="TUaddB1-start-u"
Transition: Source="53" Target="12" Event="i9"
Transition: Source="53" Target="58" Event="M2addB2-start"
Transition: Source="54" Target="17" Event="TUaddB1-finish-u"
Transition: Source="54" Target="59" Event="M2addB2-finish"
Transition: Source="55" Target="59" Event="TUaddB1-start-u"
Transition: Source="55" Target="14" Event="i9"
Transition: Source="56" Target="60" Event="TUgetB2-finish"
Transition: Source="57" Target="20" Event="TUaddB1-finish-u"
Transition: Source="57" Target="61" Event="M2addB2-start"
Transition: Source="58" Target="61" Event="TUaddB1-start-u"
Transition: Source="58" Target="17" Event="i9"
Transition: Source="58" Target="62" Event="M2addB2-finish"
Transition: Source="59" Target="22" Event="TUaddB1-finish-u"
Transition: Source="60" Target="63" Event="TUaddB1-start-u"
Transition: Source="60" Target="20" Event="i9"
Transition: Source="60" Target="64" Event="M2addB2-start"
Transition: Source="61" Target="25" Event="TUaddB1-finish-u"
Transition: Source="61" Target="65" Event="M2addB2-finish"
Transition: Source="62" Target="65" Event="TUaddB1-start-u"
```

Figure B.10 continued.

```
Transition: Source="62" Target="22" Event="i9"
Transition: Source="63" Target="31" Event="TUaddB1-finish-u"
Transition: Source="63" Target="66" Event="M2addB2-start"
Transition: Source="64" Target="66" Event="TUaddB1-start-u"
Transition: Source="64" Target="25" Event="i9"
Transition: Source="64" Target="67" Event="M2addB2-finish"
Transition: Source="65" Target="32" Event="TUaddB1-finish-u"
Transition: Source="66" Target="36" Event="TUaddB1-finish-u"
Transition: Source="66" Target="68" Event="M2addB2-finish"
Transition: Source="67" Target="68" Event="TUaddB1-start-u"
Transition: Source="67" Target="32" Event="i9"
Transition: Source="68" Target="42" Event="TUaddB1-finish-u"
```

Figure B.10 continued.

B.6 Generated Code

The following class listings show the final version of the transfer-line code with automatically generated concurrency control inserted. A `Supervisor` class was created and added to the program, and is shown in Fig. B.11. Modified were the `Machine1` class, the `Machine2` class, and the `TestUnit` class, which are shown in Figs. B.12, B.13, and B.14, respectively. The `Main` class was also modified to call the `Supervisor.init()` method. The method call was added just before the `M1`, `M2`, and `TU` threads were started.

```
package transferLine;

import java.util.concurrent.*;

public class Supervisor {

  static Semaphore M1addB1start;
  static Semaphore M2getB1start;
  static Semaphore TUgetB2start;
  static Semaphore M2addB2start;
  static int stateTracker;

  public static void init() {
    Supervisor.M1addB1start = new Semaphore(1);
    Supervisor.M2getB1start = new Semaphore(0);
    Supervisor.TUgetB2start = new Semaphore(0);
    Supervisor.M2addB2start = new Semaphore(0);
    Supervisor.stateTracker = 0;
  }

  public static synchronized boolean observeAndReact(String event,
                                          Semaphore eventBlocker) {
    if (!(eventBlocker == null )) {
      if (!eventBlocker.tryAcquire()) {
        return false;
      }
      eventBlocker.release();
```

Figure B.11: Transfer-Line: `Supervisor` class.

```
    }
    stateChange(event);
    return true;
  }
  private static void updateSupervisorState(String event) {
    if (event.equals("TUaddB1startu")) {
      switch(Supervisor.stateTracker) {
        case(23):
          Supervisor.stateTracker = 29;
          break;
        case(33):
          Supervisor.stateTracker = 39;
          break;
        case(43):
          Supervisor.stateTracker = 47;
          break;
        case(45):
          Supervisor.stateTracker = 50;
          break;
        case(51):
          Supervisor.stateTracker = 54;
          break;
        case(53):
          Supervisor.stateTracker = 57;
          break;
        case(55):
          Supervisor.stateTracker = 59;
          break;
        case(58):
          Supervisor.stateTracker = 61;
          break;
        case(60):
          Supervisor.stateTracker = 63;
          break;
        case(62):
          Supervisor.stateTracker = 65;
          break;
```

Figure B.11 continued.

```
      case(64):
        Supervisor.stateTracker = 66;
        break;
      case(67):
        Supervisor.stateTracker = 68;
        break;
    }
  }
  else if (event.equals("M1addB1start")) {
    switch(Supervisor.stateTracker) {
      case(0):
        Supervisor.M1addB1start.drainPermits();
        Supervisor.stateTracker = 1;
        break;
      case(2):
        Supervisor.M1addB1start.drainPermits();
        Supervisor.M2getB1start.drainPermits();
        Supervisor.stateTracker = 3;
        break;
      case(5):
        Supervisor.M1addB1start.drainPermits();
        Supervisor.M2getB1start.drainPermits();
        Supervisor.stateTracker = 7;
        break;
      case(6):
        Supervisor.M1addB1start.drainPermits();
        Supervisor.stateTracker = 9;
        break;
      case(10):
        Supervisor.M1addB1start.drainPermits();
        Supervisor.stateTracker = 13;
        break;
      case(12):
        Supervisor.M1addB1start.drainPermits();
        Supervisor.stateTracker = 16;
        break;
      case(14):
```

Figure B.11 continued.

```
    Supervisor.M1addB1start.drainPermits();
    Supervisor.TUgetB2start.drainPermits();
    Supervisor.stateTracker = 18;
    break;
  case(17):
    Supervisor.M1addB1start.drainPermits();
    Supervisor.stateTracker = 21;
    break;
  case(20):
    Supervisor.M1addB1start.drainPermits();
    Supervisor.stateTracker = 24;
    break;
  case(22):
    Supervisor.M1addB1start.drainPermits();
    Supervisor.M2getB1start.drainPermits();
    Supervisor.TUgetB2start.drainPermits();
    Supervisor.stateTracker = 26;
    break;
  case(25):
    Supervisor.M1addB1start.drainPermits();
    Supervisor.stateTracker = 30;
    break;
  case(32):
    Supervisor.M1addB1start.drainPermits();
    Supervisor.M2getB1start.drainPermits();
    Supervisor.TUgetB2start.drainPermits();
    Supervisor.stateTracker = 35;
    break;
  case(34):
    Supervisor.M1addB1start.drainPermits();
    Supervisor.TUgetB2start.drainPermits();
    Supervisor.stateTracker = 41;
    break;
  case(44):
    Supervisor.M1addB1start.drainPermits();
    Supervisor.TUgetB2start.drainPermits();
    Supervisor.stateTracker = 49;
```

Figure B.11 continued.

```
        break;
    }
  }
  else if (event.equals("i1")) {
    switch(Supervisor.stateTracker) {
      case(23):
        Supervisor.M1addB1start.release();
        Supervisor.stateTracker = 0;
        break;
      case(33):
        Supervisor.M1addB1start.release();
        Supervisor.M2getB1start.release();
        Supervisor.stateTracker = 2;
        break;
      case(43):
        Supervisor.M1addB1start.release();
        Supervisor.M2getB1start.release();
        Supervisor.stateTracker = 5;
        break;
      case(45):
        Supervisor.M1addB1start.release();
        Supervisor.stateTracker = 6;
        break;
      case(51):
        Supervisor.M1addB1start.release();
        Supervisor.stateTracker = 10;
        break;
      case(53):
        Supervisor.M1addB1start.release();
        Supervisor.stateTracker = 12;
        break;
      case(55):
        Supervisor.M1addB1start.release();
        Supervisor.TUgetB2start.release();
        Supervisor.stateTracker = 14;
        break;
      case(58):
```

Figure B.11 continued.

```
        Supervisor.M1addB1start.release();
        Supervisor.stateTracker = 17;
        break;
      case(60):
        Supervisor.M1addB1start.release();
        Supervisor.stateTracker = 20;
        break;
      case(62):
        Supervisor.M1addB1start.release();
        Supervisor.M2getB1start.release();
        Supervisor.TUgetB2start.release();
        Supervisor.stateTracker = 22;
        break;
      case(64):
        Supervisor.M1addB1start.release();
        Supervisor.stateTracker = 25;
        break;
      case(67):
        Supervisor.M1addB1start.release();
        Supervisor.M2getB1start.release();
        Supervisor.TUgetB2start.release();
        Supervisor.stateTracker = 32;
        break;
    }
  }
  else if (event.equals("M2getB1finish")) {
    switch(Supervisor.stateTracker) {
      case(4):
        Supervisor.M1addB1start.release();
        Supervisor.M2addB2start.release();
        Supervisor.stateTracker = 6;
        break;
      case(8):
        Supervisor.M1addB1start.release();
        Supervisor.M2addB2start.release();
        Supervisor.stateTracker = 12;
        break;
```

Figure B.11 continued.

```
      case(15):
        Supervisor.M1addB1start.release();
        Supervisor.M2addB2start.release();
        Supervisor.stateTracker = 20;
        break;
      case(28):
        Supervisor.M1addB1start.release();
        Supervisor.TUgetB2start.release();
        Supervisor.stateTracker = 34;
        break;
      case(38):
        Supervisor.M1addB1start.release();
        Supervisor.TUgetB2start.release();
        Supervisor.stateTracker = 44;
        break;
      case(46):
        Supervisor.TUgetB2start.release();
        Supervisor.stateTracker = 52;
        break;
    }
  }
  else if (event.equals("M1addB1finish")) {
    switch(Supervisor.stateTracker) {
      case(1):
        Supervisor.M1addB1start.release();
        Supervisor.M2getB1start.release();
        Supervisor.stateTracker = 2;
        break;
      case(3):
        Supervisor.M1addB1start.release();
        Supervisor.M2getB1start.release();
        Supervisor.stateTracker = 5;
        break;
      case(7):
        Supervisor.M2getB1start.release();
        Supervisor.stateTracker = 11;
        break;
```

Figure B.11 continued.

```
case(9):
  Supervisor.M1addB1start.release();
  Supervisor.stateTracker = 12;
  break;
case(13):
  Supervisor.M1addB1start.release();
  Supervisor.stateTracker = 17;
  break;
case(16):
  Supervisor.M1addB1start.release();
  Supervisor.stateTracker = 20;
  break;
case(18):
  Supervisor.M1addB1start.release();
  Supervisor.M2getB1start.release();
  Supervisor.TUgetB2start.release();
  Supervisor.stateTracker = 22;
  break;
case(21):
  Supervisor.M1addB1start.release();
  Supervisor.stateTracker = 25;
  break;
case(24):
  Supervisor.stateTracker = 31;
  break;
case(26):
  Supervisor.M1addB1start.release();
  Supervisor.M2getB1start.release();
  Supervisor.TUgetB2start.release();
  Supervisor.stateTracker = 32;
  break;
case(30):
  Supervisor.stateTracker = 36;
  break;
case(35):
  Supervisor.M2getB1start.release();
  Supervisor.stateTracker = 42;
```

Figure B.11 continued.

```
        break;
      case(41):
        Supervisor.M1addB1start.release();
        Supervisor.TUgetB2start.release();
        Supervisor.stateTracker = 44;
        break;
      case(49):
        Supervisor.TUgetB2start.release();
        Supervisor.stateTracker = 52;
        break;
    }
  }
  else if (event.equals("TUgetB2finish")) {
    switch(Supervisor.stateTracker) {
      case(19):
        Supervisor.stateTracker = 23;
        break;
      case(27):
        Supervisor.stateTracker = 33;
        break;
      case(37):
        Supervisor.stateTracker = 43;
        break;
      case(40):
        Supervisor.M2addB2start.release();
        Supervisor.stateTracker = 45;
        break;
      case(48):
        Supervisor.M2addB2start.release();
        Supervisor.stateTracker = 53;
        break;
      case(56):
        Supervisor.M2addB2start.release();
        Supervisor.stateTracker = 60;
        break;
    }
  }
```

Figure B.11 continued.

```
else if (event.equals("M2getB1start")) {
  switch(Supervisor.stateTracker) {
    case(2):
      Supervisor.M1addB1start.drainPermits();
      Supervisor.M2getB1start.drainPermits();
      Supervisor.stateTracker = 4;
      break;
    case(5):
      Supervisor.M1addB1start.drainPermits();
      Supervisor.M2getB1start.drainPermits();
      Supervisor.stateTracker = 8;
      break;
    case(11):
      Supervisor.M2getB1start.drainPermits();
      Supervisor.stateTracker = 15;
      break;
    case(22):
      Supervisor.M1addB1start.drainPermits();
      Supervisor.M2getB1start.drainPermits();
      Supervisor.TUgetB2start.drainPermits();
      Supervisor.stateTracker = 28;
      break;
    case(32):
      Supervisor.M1addB1start.drainPermits();
      Supervisor.M2getB1start.drainPermits();
      Supervisor.TUgetB2start.drainPermits();
      Supervisor.stateTracker = 38;
      break;
    case(42):
      Supervisor.M2getB1start.drainPermits();
      Supervisor.stateTracker = 46;
      break;
  }
}
else if (event.equals("TUgetB2start")) {
  switch(Supervisor.stateTracker) {
    case(14):
```

Figure B.11 continued.

```
        Supervisor.M1addB1start.drainPermits();
        Supervisor.TUgetB2start.drainPermits();
        Supervisor.stateTracker = 19;
        break;
      case(22):
        Supervisor.M1addB1start.drainPermits();
        Supervisor.M2getB1start.drainPermits();
        Supervisor.TUgetB2start.drainPermits();
        Supervisor.stateTracker = 27;
        break;
      case(32):
        Supervisor.M1addB1start.drainPermits();
        Supervisor.M2getB1start.drainPermits();
        Supervisor.TUgetB2start.drainPermits();
        Supervisor.stateTracker = 37;
        break;
      case(34):
        Supervisor.M1addB1start.drainPermits();
        Supervisor.TUgetB2start.drainPermits();
        Supervisor.stateTracker = 40;
        break;
      case(44):
        Supervisor.M1addB1start.drainPermits();
        Supervisor.TUgetB2start.drainPermits();
        Supervisor.stateTracker = 48;
        break;
      case(52):
        Supervisor.TUgetB2start.drainPermits();
        Supervisor.stateTracker = 56;
        break;
    }
  }
  else if (event.equals("TUaddB1finishu")) {
    switch(Supervisor.stateTracker) {
      case(29):
        Supervisor.M1addB1start.release();
        Supervisor.M2getB1start.release();
```

Figure B.11 continued.

```
    Supervisor.stateTracker = 2;
    break;
  case(39):
    Supervisor.M1addB1start.release();
    Supervisor.M2getB1start.release();
    Supervisor.stateTracker = 5;
    break;
  case(47):
    Supervisor.M2getB1start.release();
    Supervisor.stateTracker = 11;
    break;
  case(50):
    Supervisor.M1addB1start.release();
    Supervisor.stateTracker = 12;
    break;
  case(54):
    Supervisor.M1addB1start.release();
    Supervisor.stateTracker = 17;
    break;
  case(57):
    Supervisor.M1addB1start.release();
    Supervisor.stateTracker = 20;
    break;
  case(59):
    Supervisor.M1addB1start.release();
    Supervisor.M2getB1start.release();
    Supervisor.TUgetB2start.release();
    Supervisor.stateTracker = 22;
    break;
  case(61):
    Supervisor.M1addB1start.release();
    Supervisor.stateTracker = 25;
    break;
  case(63):
    Supervisor.stateTracker = 31;
    break;
  case(65):
```

Figure B.11 continued.

```
        Supervisor.M1addB1start.release();
        Supervisor.M2getB1start.release();
        Supervisor.TUgetB2start.release();
        Supervisor.stateTracker = 32;
        break;
      case(66):
        Supervisor.stateTracker = 36;
        break;
      case(68):
        Supervisor.M2getB1start.release();
        Supervisor.stateTracker = 42;
        break;
    }
  }
  else if (event.equals("M2addB2finish")) {
    switch(Supervisor.stateTracker) {
      case(10):
        Supervisor.TUgetB2start.release();
        Supervisor.stateTracker = 14;
        break;
      case(13):
        Supervisor.stateTracker = 18;
        break;
      case(17):
        Supervisor.M2getB1start.release();
        Supervisor.TUgetB2start.release();
        Supervisor.stateTracker = 22;
        break;
      case(21):
        Supervisor.stateTracker = 26;
        break;
      case(25):
        Supervisor.M2getB1start.release();
        Supervisor.TUgetB2start.release();
        Supervisor.stateTracker = 32;
        break;
      case(30):
```

Figure B.11 continued.

```
        Supervisor.stateTracker = 35;
        break;
      case(36):
        Supervisor.M2getB1start.release();
        Supervisor.stateTracker = 42;
        break;
      case(51):
        Supervisor.stateTracker = 55;
        break;
      case(54):
        Supervisor.stateTracker = 59;
        break;
      case(58):
        Supervisor.stateTracker = 62;
        break;
      case(61):
        Supervisor.stateTracker = 65;
        break;
      case(64):
        Supervisor.stateTracker = 67;
        break;
      case(66):
        Supervisor.stateTracker = 68;
        break;
    }
  }
  else if (event.equals("M2addB2start")) {
    switch(Supervisor.stateTracker) {
      case(6):
        Supervisor.M2addB2start.drainPermits();
        Supervisor.stateTracker = 10;
        break;
      case(9):
        Supervisor.M2addB2start.drainPermits();
        Supervisor.stateTracker = 13;
        break;
      case(12):
```

Figure B.11 continued.

```
    Supervisor.M2addB2start.drainPermits();
    Supervisor.stateTracker = 17;
    break;
  case(16):
    Supervisor.M2addB2start.drainPermits();
    Supervisor.stateTracker = 21;
    break;
  case(20):
    Supervisor.M2addB2start.drainPermits();
    Supervisor.stateTracker = 25;
    break;
  case(24):
    Supervisor.M2addB2start.drainPermits();
    Supervisor.stateTracker = 30;
    break;
  case(31):
    Supervisor.M2addB2start.drainPermits();
    Supervisor.stateTracker = 36;
    break;
  case(45):
    Supervisor.M2addB2start.drainPermits();
    Supervisor.stateTracker = 51;
    break;
  case(50):
    Supervisor.M2addB2start.drainPermits();
    Supervisor.stateTracker = 54;
    break;
  case(53):
    Supervisor.M2addB2start.drainPermits();
    Supervisor.stateTracker = 58;
    break;
  case(57):
    Supervisor.M2addB2start.drainPermits();
    Supervisor.stateTracker = 61;
    break;
  case(60):
    Supervisor.M2addB2start.drainPermits();
```

Figure B.11 continued.

```
          Supervisor.stateTracker = 64;
          break;
        case(63):
          Supervisor.M2addB2start.drainPermits();
          Supervisor.stateTracker = 66;
          break;
      }
    }
    else {
      //should never reach this!
      System.out.println("Supervisor failure");
      assert (false);
    }
  }
}
```

Figure B.11 continued.

```
package transferLine;

//creates new parts and puts them into the first buffer
public class Machine1 extends Thread {

  //the target for created  parts
  Buffer target;

  public Machine1(Buffer target) {
    new Thread();
    this.target = target;
    this.setName("Machine1");
  }

  public void doWork() {
    try {
      Thread.sleep((int)(Math.random()*1000));
    }
    catch (Exception e) {
      //do nothing
    }
  }

  public void run() {
    while (true) {
      //wait for some random period
      doWork();
      //create a new Part
      Part newPart = new Part();
      //event marking: M1addB1-start
      while (true)
        if (Supervisor.observeAndReact("M1addB1start",
                                       Supervisor.M1addB1start))
          break;
        Supervisor.M1addB1start.acquireUninterruptibly();
        Supervisor.M1addB1start.release();
```

Figure B.12: Transfer-Line: `Machine1` with added concurrency control.

```
      //put it in the target buffer
      System.out.println("Machine1 tries to put a part in " + target);
      target.addPart(newPart);
      //event marking: M1add-B1-finish
      Supervisor.observeAndReact("M1addB1finish", null);
    } // loop forever
  }
}
```

Figure B.12 continued.

```
package transferLine;

public class Machine2 extends Thread {

  //the source and target for parts used by this machine
  Buffer source;
  Buffer target;

  public Machine2(Buffer source, Buffer target) {
    new Thread();
    this.source = source;
    this.target = target;
    this.setName("Machine2");
  }

  public void doWork() {
    try {
      Thread.sleep((int)(Math.random()*10));
    }
    catch (Exception e) {
      //do nothing
    }
  }

  public void run() {
    while (true) {
      //get part from source Buffer
      //event marking: M2getB1-start
      while (true)
        if (Supervisor.observeAndReact("M2getB1start",
                                       Supervisor.M2getB1start))
          break;
        Supervisor.M2getB1start.acquireUninterruptibly();
        Supervisor.M2getB1start.release();

      System.out.println("Machine2 tries to get a part from "
                                                  + source);
```

Figure B.13: Transfer-Line: `Machine2` with added concurrency control.

```
      Part currentPart = source.removePart();
      //event marking: M2getB1-finish
          Supervisor.observeAndReact("M2getB1finish", null);
      //we have a part - do some work on it
      doWork();
      //put the part in the next buffer
      //event marking: M2addB2-start
      while (true)
        if (Supervisor.observeAndReact("M2addB2start",
                                       Supervisor.M2addB2start))
          break;
        Supervisor.M2addB2start.acquireUninterruptibly();
        Supervisor.M2addB2start.release();

      System.out.println("Machine2 tries to put a part in "
                         + target);
      target.addPart(currentPart);
      //event marking: M2addB2-finish
          Supervisor.observeAndReact("M2addB2finish", null);
      //reset machine 2
      doWork();
    } //loop forever
  }
}
```

Figure B.13 continued.

```
package transferLine;

public class TestUnit extends Thread{

  Buffer source;
  Buffer rejectBuffer;
  double rejectionChance;

  public TestUnit(Buffer source, Buffer rejectBuffer,
                  double rejectionChance) {
    new Thread();
    this.source = source;
    this.rejectBuffer = rejectBuffer;
    this.rejectionChance = rejectionChance;
    this.setName("TestUnit");
  }

  public void doWork() {
    try {
      Thread.sleep((int)(Math.random()*10));
    }
    catch (Exception e) {
      //do nothing
    }
  }

  public void run() {
    while (true) {
      //get a part from the source
      //event marking: TUgetB2-start
      while (true)
        if (Supervisor.observeAndReact("TUgetB2start",
                                       Supervisor.TUgetB2start))
          break;
        Supervisor.TUgetB2start.acquireUninterruptibly();
        Supervisor.TUgetB2start.release();
```

Figure B.14: Transfer-Line: `TestUnit` with added concurrency control.

```
      System.out.println("TestUnit tries to get a part from "
                         + source);
      Part currentPart = source.removePart();
      //event marking: TUgetB2-finish
      Supervisor.observeAndReact("TUgetB2finish", null);
      //test that part for some period of time
      doWork();
      if (Math.random() > rejectionChance) {
        //no good!  send back to rejectBin
        //event marking: TUaddB1-start-u
        Supervisor.observeAndReact("TUaddB1startu", null);
        System.out.println("TestUnit tries to put a part in "
                           + rejectBuffer);
        rejectBuffer.addPart(currentPart);
        //event marking: TUaddB1-finish-u
        Supervisor.observeAndReact("TUaddB1finishu", null);
      }
      else {
        //event marking: i9
        Supervisor.observeAndReact("i9", null);
      }
    } //loop forever
  }
}
```

Figure B.14 continued.