Using Discrete-Event Systems for the Automatic Generation of Concurrency Control for Dynamic Threads

by

Anthony R. Auer

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
May 2010

Copyright © Anthony R. Auer, 2010
Abstract

The application of Discrete-Event Systems (DES) theory to the problem of guaranteeably enforcing concurrency constraints in multi-threaded applications has been studied under certain assumptions, namely, the assumption of a static pool of pre-existing instantiated threads, whose creation and termination are not modelled. This work proposes an extension of this case to handle dynamically instantiated and terminated threads using a Petri net formalism and an online limited-lookahead state-space search technique.
Acknowledgments

My thanks are warmly extended to my supervisors, Drs. Karen Rudie and Juergen Dingel. As always, my deepest love and gratitude go out to my parents, Drs. Jan and Janette Auer, and grandparents, Mr. and Mrs. Harvey and Shirley Ross, for their unrelenting support both material and intellectual throughout this endeavour.

To my wife Erica, who has been the emblem of stoic patience and support, I dedicate this work.
# Table of Contents

Abstract i

Acknowledgments ii

Table of Contents iii

List of Figures vii

1 Introduction 1

Chapter 2:

<table>
<thead>
<tr>
<th>Background</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Concurrency Control</td>
<td>6</td>
</tr>
<tr>
<td>2.1.1 Deadlock and Livelock</td>
<td>7</td>
</tr>
<tr>
<td>2.1.2 Concurrency Strategies</td>
<td>8</td>
</tr>
<tr>
<td>2.1.3 Note on Scheduling</td>
<td>10</td>
</tr>
<tr>
<td>2.2 Discrete-Event Systems</td>
<td>10</td>
</tr>
<tr>
<td>2.2.1 Automata Theory</td>
<td>10</td>
</tr>
<tr>
<td>2.2.2 Controllability and Supervisors</td>
<td>13</td>
</tr>
<tr>
<td>2.2.3 The DES Approach to Concurrency Control</td>
<td>15</td>
</tr>
<tr>
<td>2.3 Limited Lookahead</td>
<td>17</td>
</tr>
</tbody>
</table>
2.3.1 Limited Lookahead for DES ................................. 18
2.3.2 Dynamic DES ............................................. 22
2.4 Petri Nets ..................................................... 24
   2.4.1 Petri Net Definitions ................................. 24
   2.4.2 Petri Net Characteristics ......................... 27
   2.4.3 Extended Petri Nets ................................. 28

Chapter 3:

   Method, Model and Algorithm ................................. 30

   3.1 Petri Net Concurrency Model ........................... 30
   3.2 Issues and Modelling Considerations .................. 33
       3.2.1 Control Flow and Controllability ............... 33
       3.2.2 The Marked State Problem ...................... 37
       3.2.3 Lookahead Length ............................... 40
       3.2.4 Specification Expression ....................... 42
   3.3 Online Supervisor Synthesis ............................ 45
       3.3.1 Source code markup .............................. 48
       3.3.2 Thread model generation ....................... 49
       3.3.3 Constraint specification ....................... 49
       3.3.4 Supervisor autogeneration ..................... 50
       3.3.5 Supervisor code translation .................... 52
       3.3.6 Supervisor call insertion ..................... 52
   3.4 Supervisor Runtime Action ............................... 53
       3.4.1 Event Firing .................................... 53
       3.4.2 Lookahead Algorithm ............................ 54
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5</td>
<td>Example</td>
<td>59</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Description</td>
<td>59</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Markup and FSA Generation</td>
<td>61</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Specification Expression</td>
<td>64</td>
</tr>
<tr>
<td>3.5.4</td>
<td>Plant Synthesis</td>
<td>67</td>
</tr>
<tr>
<td>3.5.5</td>
<td>Online Control</td>
<td>68</td>
</tr>
<tr>
<td>Chapter 4:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Results</td>
<td>77</td>
</tr>
<tr>
<td>4.1</td>
<td>Implementation</td>
<td>77</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Size Limitations: Space Versus Time</td>
<td>77</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Note on Caching</td>
<td>79</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Performance</td>
<td>80</td>
</tr>
<tr>
<td>4.2</td>
<td>Validation</td>
<td>81</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Ad Hoc Testing</td>
<td>81</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Java PathFinder</td>
<td>86</td>
</tr>
<tr>
<td>Chapter 5:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Related Literature</td>
<td>87</td>
</tr>
<tr>
<td>5.1</td>
<td>Path Expressions</td>
<td>87</td>
</tr>
<tr>
<td>5.2</td>
<td>Synchronization Skeletons</td>
<td>88</td>
</tr>
<tr>
<td>5.3</td>
<td>Petri Nets and SBPI</td>
<td>90</td>
</tr>
<tr>
<td>5.4</td>
<td>Coordinators and Adapter Synthesis</td>
<td>91</td>
</tr>
<tr>
<td>5.5</td>
<td>Synchronous Concurrency</td>
<td>92</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Sample finite state automaton. ................................. 12
2.2 Intuitive depiction of the lookahead window. ................. 18
2.3 A simple Petri net and its reachability graph. ............... 26
2.4 A simple extended Petri net. ..................................... 28
3.1 Simple two-thread Java code snippet. .......................... 31
3.2 Sample cascading-state-machine Petri net. ....................... 32
3.3 Two Petri net expressions of the \{A, B\} specification. ....... 33
3.4 Thread with two controllable paths. ............................ 34
3.5 Automata with and without control-flow events. ............... 35
3.6 An automaton specification disabling $\alpha$. .................. 35
3.7 Automata under control of specifications. ..................... 36
3.8 An extended Petri net specification disabling $c$. ............ 41
3.9 Representations of the mutex specification in three different structures. 44
3.10 Flow chart depicting the Supervisor autogeneration process. 47
3.11 Modelling of “implicit” events. .................................. 48
3.12 Pseudocode for supervisor call. ................................. 54
3.13 Depiction of lookahead exploration. ............................ 57
3.14 Pseudocode for lookahead algorithm. .......................... 58
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.15</td>
<td>Depiction of dual recursion tracking.</td>
<td>60</td>
</tr>
<tr>
<td>3.16</td>
<td>The Java code for the R, W and RW objects.</td>
<td>61</td>
</tr>
<tr>
<td>3.17</td>
<td>Java code for the R, W and RW objects with Markup.</td>
<td>62</td>
</tr>
<tr>
<td>3.18</td>
<td>The FSA representations of the R, W and RW objects’ behaviour.</td>
<td>63</td>
</tr>
<tr>
<td>3.19</td>
<td>Instrumented Java code for the R, W and RW objects.</td>
<td>65</td>
</tr>
<tr>
<td>3.20</td>
<td>Mutex and priority specifications.</td>
<td>66</td>
</tr>
<tr>
<td>3.21</td>
<td>Spawn map for Readers/Readers/Readers.</td>
<td>67</td>
</tr>
<tr>
<td>3.22</td>
<td>CSM Petri net for the R, W, and RW classes.</td>
<td>69</td>
</tr>
<tr>
<td>3.23</td>
<td>An enumeration of places in the plant Petri net.</td>
<td>70</td>
</tr>
<tr>
<td>3.24</td>
<td>Plant Petri net with potential mutex disablement.</td>
<td>71</td>
</tr>
<tr>
<td>3.25</td>
<td>Mutex specification disabling $\text{startW}$.</td>
<td>72</td>
</tr>
<tr>
<td>3.26</td>
<td>Petri net with contention over $\text{startW}$ and $\text{startR}$ priorities.</td>
<td>73</td>
</tr>
<tr>
<td>3.27</td>
<td>Read priority specification disabling $\text{startW}$.</td>
<td>74</td>
</tr>
<tr>
<td>4.1</td>
<td>Variation of lookahead tree size.</td>
<td>78</td>
</tr>
<tr>
<td>4.2</td>
<td>Variation of lookahead tree size.</td>
<td>80</td>
</tr>
<tr>
<td>4.3</td>
<td>Simple example used in ad hoc testing.</td>
<td>82</td>
</tr>
<tr>
<td>4.4</td>
<td>Finite state automaton representations of threads.</td>
<td>83</td>
</tr>
<tr>
<td>4.5</td>
<td>Simple precedence constraint implemented with a finite state automaton.</td>
<td>83</td>
</tr>
<tr>
<td>4.6</td>
<td>Assertion object for constraint checking.</td>
<td>84</td>
</tr>
<tr>
<td>4.7</td>
<td>Instrumented R and W objects.</td>
<td>85</td>
</tr>
<tr>
<td>6.1</td>
<td>The Java code for Threads A and B in the current supervisor context.</td>
<td>96</td>
</tr>
<tr>
<td>6.2</td>
<td>The Java code for Thread A with Supervisor-Aware event calls.</td>
<td>97</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Concurrent or *multithreaded* software design and development is an ever-increasing priority for study and practice. Not only is concurrency capable of drastically simplifying the design of some algorithms, it is also fast becoming the sole avenue for reaping the rewards of multiprocessor systems. Indeed, as time goes on, with real limits in sight to the historically exponential growth in raw processor speed, the sheer necessity of rethinking software design to accommodate multiple processors—rather than ever-faster single processors—becomes increasingly apparent. The necessity of a paradigm shift has been cited [28] to address this imminent bottleneck.

Aside from hardware and performance considerations, concurrent software design can be indicated when the problem itself is naturally expressible as multiple independent processes, or when the individual subproblems are actually solved in an intrinsically remote manner, possibly on different hardware altogether.

Despite the benefits both conceptual and effectual in the design of concurrent software, its execution remains very difficult for most practitioners; while simplifying the design of each individual thread, the aggregate thread behaviour is considerably more
complex, unpredictable and opaque than the action of each thread alone. Concurrent software is among the most difficult to reason about and in which to anticipate failures. Automatic parallelization is a promising avenue for problem classes with the correct structure (“divide and conquer”, dynamic programming, etc.); divining a target application’s membership in such a class is however a problem in itself [11].

The development of automatically parallelizing compilers is showing some promise, but the technology is presently sparse, dominated by Fortran compilers and not in mainstream use.

In other research, much effort has been invested in attempts to formalize reasoning about concurrent software with an eye toward automatically generating appropriate control mechanisms that avoid the pitfalls generally plaguing its development. One approach to the concurrency problem is the application of formal methods and modelling techniques, lifting the problem from the code to a higher-level modelling domain. A variety of techniques have been explored, most notably using formal logic, but also within the context of modelling paradigms such as discrete-event systems (DES) and finite automata.

The work of Dragert et al. [12, 13] has demonstrated the viability of using a discrete-event systems (DES) framework to model a static set of concurrently executing threads in a high-level programming language and automatically generating source code which imposes control on those threads according to globally-specified constraints. Such control (called *supervisory control* in the DES context) has several guaranteeable characteristics, most notably deadlock freedom (the source under control will never reach a system state from which it cannot proceed) and maximal permissiveness (the control disallows only the smallest subset of possible behaviour
required to ensure the constraints are enforced). The net effect is to free the developer from having to manually devise correct concurrency control code, increasing productivity and software quality.

A central characteristic of the work is its assumption that the modelled threads exist as a static pool throughout the window of interest. This assumption presents a barrier to real-world application. In a typical high-level language with concurrency facility, it is possible to express the dynamic creation and termination of individual threads, usually as procedure calls within existing threads. The precise pattern of thread existence over time is thus unknown, unless a mechanism is developed for modelling the conditions which must be met for their creation and termination. In addition, even if the system is still deterministically modellable, it is conceivable that computation of the entire model be intractable.

This work sets out to accomplish two goals: first, to accommodate the modelling of dynamic threads in a way in which their creation and termination is part of the action of the system under control; second, to maintain the tractability of the control decision in the face of a large state space which cannot be computed in its entirety offline. The first is accomplished through the use of Petri nets, a family of automata which inherently reflects concurrent state transitioning. Petri nets have been used to model resource flow in complex manufacturing systems for decades [18], and the literature abounds with discourse on their use in a variety of applications. Attention is restricted here to a subclass of Petri nets which reflect the structure of a concurrent software application. The second goal is approached by adopting an online search strategy (one which is computed during, rather than prior to, software execution) coupled with a limited-lookahead search [5]. Limited lookahead refers to an incomplete search of
future execution sequences, relying on a policy to heuristically fill in the information gap due to those areas left unsearched.

As an extension to the earlier static case, the dynamic approach represents an increase in generality on two fronts: firstly and more obviously the representation of a more general runtime sequence based on dynamic threads, but also in the use of Petri nets to express global concurrency constraints. In doing so the scope of expressibility of those constraints is increased over the regular language capability inherent in the finite state-based approach of Dragert et al. In particular, a large (proper) subset of non-regular, context-free languages are specifiable using Petri nets. With a further extension of Petri nets (called, unsurprisingly, \textit{extended Petri nets}), expression of arbitrary constraints is achieved, though at some cost as shall be shown (see Section 2.4.3).

The second advantage, increased tractability, is achieved at a cost related to the theoretical guarantees provided by the static approach. The limited lookahead technique guarantees an upper bound on computation time, but with the following disadvantages: the control decision computation must be performed online (during execution), meaning that it must be very quick to avoid performance compromises; and because the state space is not fully explored, there is the possibility of precision loss. More specifically, the static case makes two important guarantees:

- \textit{Deadlock freedom}: the search is guaranteed not to lead the system into a state from which it cannot proceed; and

- \textit{Maximal permissiveness}: the search is guaranteed to allow the largest possible subset of permissible behaviours while still disallowing all constraint violation.

Both these properties will be better quantified in Chapter 2. It will become
evident that the online limited lookahead approach used in this work must compromise between these two guarantees; the ramifications and tradeoffs thereof will be discussed at length.

The next chapter will present a treatment of the necessary theoretical background in concurrency and discrete-event systems, and the actual modelling and control approach will follow in Chapter 3, including a detailed description of the lookahead algorithm. Chapter 4 will discuss a proof-of-concept implementation created as part of this work, evaluate the correctness and performance properties of that implementation and speculate on possible future avenues of research. Chapter 5 will present a survey of existing approaches to software concurrency control; finally, Chapter 6 will present a conclusion with respect to the general viability of the approach.
Chapter 2

Background

2.1 Concurrency Control

The principal issues which arise in concurrent software centre around locally-controlled processes which must conform to a set of global constraints. These constraints typically arise from the necessity of sharing common resources whose integrity must be preserved. *Concurrency control* refers to the enforcement of global constraints on the aggregate behaviour of a collection of autonomous, independently-defined processes. Such constraints typically arise due to contention for shared resources. An early solution to the problem of contention for resources is the implementation of *synchronization locks* or *semaphores*, first proposed by Dijkstra in 1965 [9]. A semaphore is a simple global counter (always nonnegative) which represents complete or partial utilization of a resource by one or more processes. In attempting to access a resource, an associated semaphore must be decremented, if it is not already zero. A semaphore with a value of zero must force the process to wait or give up on the resource access. Upon completion, the process re-increments the semaphore. A synchronization lock
is the degenerate case of a semaphore with a maximum value of 1. A variety of other programming paradigms and abstractions now exist which further support the task of dealing with concurrency. Among these are:

- *Channel contracts*: state-machine-based specifications for asynchronous communication [27];
- *Thread pools*: aggregations of threads designed to perform the same set of tasks which are usually taken in turn from a queue [22];
- *Multiple-condition wait-locks*: synchronization locks with the characteristic that threads waiting upon them can be selectively notified based on multiple criteria when the locks are freed [22].

### 2.1.1 Deadlock and Livelock

Semaphores and synchronization locks do not completely solve the concurrency problem. Instead of competing for a resource (possibly causing loss of integrity of that resource), processes now compete for locks. This can lead to a scenario called *deadlock*, in which two processes each wait indefinitely for the other to give up its respective lock; since both are waiting, neither ever gives up its lock.

Even with a timeout protocol under which the processes eventually give up their existing locks, it is possible to enter a *livelock*: the processes cycle through a number of states, attempting to obtain locks, and preventing each other from progressing toward a desired state of completion.

The deadlock and livelock problems are illustrated succinctly by the Dining Philosopher’s problem, first formulated by Dijkstra in an examination question as a set of
machines and later reformulated by Hoare in its present, more colourful context [17].

2.1.2 Concurrency Strategies

The basic strategies employed in the design and implementation of concurrent code [36] are:

1. Static Analysis

   Rather than a control mechanism *per se*, static analysis attempts to root out deadlock problems through code inspection. Static analysis techniques involve control-flow construct analysis and synchronization lock inspection.

   The lack of runtime information (such as variable values) prohibits static analysis for a large part of the state search space. As a result, false positives (the reporting of constraint violations where none exist) can occur easily.

2. Dynamic Analysis

   Dynamic analysis attempts to execute sample runs of the global system, monitoring system state (such as variable values, etc.) in order to detect violations. Unfortunately, this approach is non-exhaustive for all but the simplest of concurrent programs, and thus cannot provide guarantees as to its ability to detect all constraint violations. Runtime model-checking tools such as Java PathFinder fall into this category.

3. Transaction-Based Control
An underlying framework of memory management could in theory provide deadlock and/or violation detection, rolling back the memory state to a previous state in such cases. Much research is required before such techniques will be ready for use.

4. Runtime Supervisory Control

The control generation techniques examined in this thesis fall squarely into this last category. Some measure of offline reasoning is used to automatically generate a framework which models global system states and anticipates and prevents illegal ones.

Control can be either online (dynamically generated according to runtime context) or offline (synthesized prior to execution of the target code). In addition, the strength of control is generally classified using the following characteristics:

1. Detection:

   The control mechanism detects when desired constraints are violated and initiates some sort of reparatory routine (such as a state reset or transactional rollback), or ends execution.

2. Prevention:

   Control strictly and conservatively prevents sequences of system states that might lead to constraint violation. The work of Dragert [12] (see Section 2.2.3) falls into this category.

3. Avoidance:
Heuristic and state metric information is used to dynamically prevent illegal states at the time they might be entered, usually by delaying specific events. Some measure of nonconformance is considered acceptable. As shall be shown, the introduction of limited lookahead forces the relaxation of some of the strict prevention characteristics of Dragert due to the lack of complete system information; thus the method described herein falls into this category.

2.1.3 Note on Scheduling

Constraints on concurrent control in the present context do not address another related but distinct concept: that of scheduling. Processes are scheduled when they share a single processing mechanism and are given interleaving periods in which to perform their tasks. Optimal scheduling is not treated in this work; most approaches to concurrency control do, however, make a modest assumption of weak/fair scheduling; that is, scheduling which guarantees that every process will always receive a chance at execution within a finite amount of time [24].

For processes running on individual, dedicated processors, scheduling is not relevant.

2.2 Discrete-Event Systems

2.2.1 Automata Theory

Discrete-event systems theory models a system under control as a set of non-overlapping states traversed via sequences of (possibly nondeterministic) occurrences of instantaneous events. Though there exist many variations and embellishments, the original
Ramadge-Wonham discrete-event system framework is generally represented as a finite state automaton with additional features to represent concepts of control.

A standard definition [19] of a finite state automaton is as a 5-tuple:

\[ A = \{ \Sigma, Q, \delta, q_0, Q_m \} \]

where

- \( \Sigma \) is a set of symbols (an alphabet), sequences of which are called strings;
- \( Q \) is a set of states;
- \( \delta : Q \times \Sigma \rightarrow Q \) is a transition function, a partial function defining transitions from state to state by means of symbols from \( \Sigma \);
- \( q_0 \in Q \) is a start state; and
- \( Q_m \subseteq Q \) defines a set of marked or accepting states.

Finite automata are amenable to representation as finite directed graphs with arcs labelled with symbols from the alphabet. By traversing the arcs and noting their labels, symbol sequences (strings) can be seen as either generated or accepted (marked) by the automaton. String generation is represented by any traversal of the graph (equivalent to recursively applying the \( \delta \)-function); string marking is represented by any traversal ending in a state \( q \in Q_m \). The sets of strings generated and marked by the automaton \( A \) are regular languages [19], denoted \( \mathcal{L}(A) \) and \( \mathcal{L}_m(A) \subset \mathcal{L}(A) \) respectively.

Figure 2.2.1 depicts a graph representation of the following automaton:

\[ A = \{ \{\alpha, \beta, \gamma\}, \{1, 2, 3, 4, 5\}, \delta_A, 1, \{4, 5\} \} \]
where $\delta_A$ is defined by:

\[
\begin{align*}
\delta_A(1, \alpha) &= 2 \\
\delta_A(2, \beta) &= 3 \\
\delta_A(3, \alpha) &= 4 \\
\delta_A(3, \beta) &= 5 \\
\delta_A(3, \gamma) &= 1
\end{align*}
\]

and $\delta_A(x, e)$ undefined for all pairs $(x, e) \in Q \times \Sigma$ not enumerated above.

Operations definable on regular languages have analogues definable on finite state automata [4], for example:

- The **parallel composition** of two regular languages $L_1 \cap L_2$ is computable by traversing their respective automata event-by-event in lock-step for shared events and interleaved for unshared;

- The **shuffle** of two regular languages $L_1$ and $L_2$ with disjunct alphabets is the
language generated by all possible asynchronous interleavings of event sequences from $\mathcal{L}_1$ and $\mathcal{L}_2$.

2.2.2 Controllability and Supervisors

In the discrete-event systems (DES) context, the alphabet is viewed as a collection of possible instantaneous events (called the event set) whose occurrence can effect state transitions (as per the transition function $\delta$). In addition, the property of controllability partitions the alphabet into a cover of two disjunct subsets: a subset $\Sigma_c \subseteq \Sigma$ is denoted the set of controllable events, with all other events (those in $\Sigma_u = \Sigma \setminus \Sigma_c$) uncontrollable.

A system under control, called a plant, is modelled as an automaton generating event sequences according to its internal structure. The plant (typically denoted $G$) may be modelled as a monolithic automaton, or as a set of automata the synthesis of whose actions comprise the complete system. The plant generates a language $\mathcal{L}(G)$ as noted above, and marks a sublanguage $\mathcal{L}_m(G) \subseteq \mathcal{L}(G)$.

The object of control is to impose a set of desired constraints upon the behaviour of the plant, which is accomplished by partitioning $\mathcal{L}$ by legality. Legality is generally expressed using a language called a specification; where the specification is regular the constraints can be represented as an automaton $E$, and the language denoted $\mathcal{L}(E)$. The automaton sought out is a supervisor $S$ which, in intersection with the plant, yields a closed-loop language $\mathcal{L}(S/G)$ with the following desired properties:

1. only legal strings generable by the plant are generable by the closed-loop system:

   $$\mathcal{L}(S/G) \subseteq \mathcal{L}(E) \cap \mathcal{L}(G)$$
2. any uncontrollable event appended to any legal string must produce a string which is also legal:

\[ s \in \mathcal{L}(S/G) \land \sigma \in \Sigma_u \Rightarrow s\sigma \in \mathcal{L}(S/G) \]

3. the result must be nonblocking: any event sequence generable by the closed-loop system must be “completable” to—in formal terms, be a prefix\(^1\) of—a sequence marked by it (i.e., in \(\mathcal{L}_m(S/G)\)). The notation \(\overline{\mathcal{L}_m(S/G)}\) denotes the prefix closure of the language \(\mathcal{L}_m(S/G)\): all strings which are prefixes of strings in \(\mathcal{L}_m(S/G)\). A language which is equal to its own prefix closure is called prefix-closed. A nonblocking solution results when:

\[ \mathcal{L}(S/G) = \overline{\mathcal{L}_m(S/G)} \]

Ramadge and Wonham [39] show that such subsets of \(\mathcal{L}(E) \cap \mathcal{L}(G)\) with the above properties form an upper semilattice partially ordered by set inclusion; thus there is a largest such subset, called the supremal controllable sublanguage of \(E\) with respect to \(G\), and denoted \(\text{sup} \mathcal{C}(E)\) (or \(\text{supcon}\) in the parlance of prevalent Discrete Event Systems software such as IDES\(^2\) [26] or TCT [14]). Moreover, they showed that such a language (and a supervisor \(S\) required to enforce it) is algorithmically derivable in polynomial time: specifically, in \(O(|E|^2|G|^2)\) time [25] (where the cardinality notation \(|A|\) denotes the number of states of an automaton \(A\))\(^3\). This result is both nonblocking by construction, and maximally permissive by definition of supremum.

---

\(^1\)A string \(p\) is a prefix of another string \(s\) if there is a string \(x\) such that \(s = px\).

\(^2\)IDES figures prevalently in this work both as a modelling tool and as the source of the finite state automaton representations depicted in this paper. As such, the resolution of such representations is limited by that of IDES.

\(^3\)It may be, of course, that the resultant language is empty or contains no sequences germaine to the useful behaviour of the plant; the point is that the language is computable.
The overall algorithm for supervisor generation is informed by the foregoing operations on automata. The plant and specifications can be synthesized from modules which depict the actions of independent subsystems. The asynchronous product is used to derive the global plant, and the synchronous product to derive the global specification. Finally, the controlled system is derived from the supremal controllable sublanguage of the language marked by the plant with respect to that generable by the specification (by means of the supcon operation).

2.2.3 The DES Approach to Concurrency Control

Dragert [12] provides a method for using specifications expressed directly as automata to automatically generate control code in a given high-level language (the target language). To do so, a transformation of the target code to automaton must first be carried out; this is achieved through the use of instrumentation of relevant events. The automaton transformations described above (synchronous/asynchronous composition, supcon) are then used to construct the appropriate finite state automaton which acts as a supervisor; this automaton can then be used to generate a state machine and control map in the target language.

The method is a six-step process:

1. Source code markup: the points (boundaries) in the target code relevant to control decisions must be indicated through some convention that is ignored by the code’s interpreter/compiler. A manual step, it consists of identifying points in the source code at which there is concurrency relevance. This typically means finding the boundaries of sections in which shared objects are accessed, or certain conditions must be met for the source code to proceed—conditions
not explicit in the code of a single thread.

2. **Thread model generation**: a finite state automaton representation of each thread is generated, the events of which are mapped from the code boundaries indicated by the markup of step 1. This step can be automated with a specialized control-flow graph generator.

3. **Constraint specification**: formal specifications that express the desired aggregate behaviour (constraints) of the threads are collected and translated into finite state automata.

4. **Supervisor autogeneration**: the supervisor’s automaton representation is generated through a series of automated discrete-event system computations:

   (a) the asynchronous product of the thread automata is taken, forming the plant;

   (b) the synchronous product of the constraint automata is taken, forming the specification;

   (c) the supcon operation is applied to the plant/spec pair, constructing an automaton representation of the supervisor.

5. **Supervisor code translation**: The automaton given in step 4 is translated into a control map in the target language which tracks the sequence of events encountered by the markup of step 1, and given a prospective event from that markup returns either true or false based on the legality of the event in its internally tracked current state.

6. **Supervisor call insertion**: the code is refiltered, replacing each event markup with explicit calls to the supervisor supplying the relevant event as an argument.
Apart from eliminating the need to manually design concurrency control, the principal advantage of this approach is the mathematical guarantee of deadlock freedom it provides due to the use of supcon to find the supremal controllable sublanguage (see Section 2.2.2); in addition, by the definition of the supremal controllable sublanguage, it guarantees maximal permissiveness. The deadlock-freedom guarantee offers an advantage over earlier works in which the latter had to be checked using model verification techniques.

2.3 Limited Lookahead

In terms of classification by the concurrency control criteria given in Section 2.1.2, the approach of Dragert falls into the category of strict prevention on an offline basis. The aforementioned advantages in terms of theoretical guarantees are in this sphere offset by computational burden: in many practical applications, the closed-loop behaviour of the system is extremely large, and under some system models impossible or intractable to calculate. Such intractability sets the stage for the application of techniques which limit calculation to a subset of the closed-loop system; limited lookahead is one such technique.

The notion of limited lookahead for discrete-event systems was first explored by Chung, Lafortune and Lin [5] in order to find ways to mitigate the large state-space search cost of computing the plant. It has as its objective the computation of an effective control decision based on exploration of only a finite neighbourhood of the present state. The dynamic DES paradigm of Grigorov and Rudie [15] extends the limited-lookahead idea to encompass plants which change over time. The future plant behaviour, in addition to being conceivably too large to compute at once, is
also unknown.

### 2.3.1 Limited Lookahead for DES

Central to the concept of limited lookahead is the definition of a *window* of depth $N$, up to which exploration of the closed-loop DES system is carried out. Exploration in this sense means the generation of the closed-loop system as described in Section 2.2.2.

![Intuitive depiction of the lookahead window. Source: [5]](image)

As depicted in Figure 2.3.1, the window is therefore a tree or graph comprised of states resultant from possible future event sequences up to length $N$: in effect a neighbourhood of the initial or current state. The edge of that neighbourhood, called the *lookahead depth*, partitions the languages $L(S/G)$ and $L_m(S/G)$ of the closed-loop...
system into sublanguages based on the length $N$. Chung et al. [5] use the following constructs to describe those sublanguages of a given language $\mathcal{L}$:

- The *postlanguage* of a string $s \in \mathcal{L}$:
  $$\mathcal{L}/s = \{ t \in \Sigma^* \mid st \in \mathcal{L} \}$$

- The *right quotient language* of $\mathcal{L}$ with respect to a language $K$:
  $$\mathcal{L}/K = \{ s \in \Sigma^* \mid (\exists t \in K \mid st \in \mathcal{L}) \}$$

- The *truncation* of $\mathcal{L}$ to depth $N$:
  $$\mathcal{L}|_N = \{ s \in \mathcal{L} \mid |s| \leq N \}$$

These language definitions in conjunction with the standard DES language definitions allow the authors to define a *control action* given a past event history $s$ and generable and marked languages $\mathcal{L}(G)$ and $\mathcal{L}_m(G)$. They devise a composition of five functions to effect this action:

1. The generable and marked postlanguages of $s$, truncated to length $N$:
   $$f^N_{\mathcal{L}(G)}(s) = (\mathcal{L}(G)/s|_N, \mathcal{L}_m(G)/s|_N)$$

2. The legal postlanguages (as dictated by the specifications) of $s$, truncated to length $N$:
   $$f^N_K \cdot f^N_{\mathcal{L}(G)}(s) = (K/s|_N, K/s|_N)$$

3. The *lookahead policy* defined on the window boundary. The policy takes as input a set of *pending traces*: those strings of exactly length $N$ in the legal postlanguage $K/s|_N - K/s|_{N-1}$—that is, those whose possible suffixes are not
yet known:

\[ f_a^N \cdot f_K^N \cdot f_{E(G)}^N(s) = f_a (\overline{K}/s|_N - \overline{K}/s|_{N-1}) \]

The action of \( f_a \) is based on the specific decision policy adopted. The authors contemplate both a conservative and an optimistic policy, to be described shortly.

4. The supremal controllable sublanguage of the foregoing:

\[ f^N(s) = \sup \mathcal{C} \left( f_a^N \cdot f_K^N \cdot f_{E(G)}^N(s) \right) \]

5. Finally the control decision, comprised of the union of all events allowed by the supervisor with all uncontrollable events (to ensure feasibility of the supervisor):

\[ \gamma^N(s) = f_u^N \cdot f^N(s) = \overline{f^N(s)}|_1 \cup \Sigma_u|_1 \cap L(G)/s_1 \]

The function \( \gamma^N(s) \) amounts to disablement of all controllable events deemed contrary to the lookahead policy \( f_a \). The policy is necessary because information about the closed-loop system is lost in truncation at the lookahead boundary which may have had an effect on the control decision regarding the next event fired, had the decision been based purely on the supremal controllable sublanguage.

Chung, Lafortune and Lin contemplated a spectrum of policies, the extremes of which they investigated in their paper: a conservative policy regarding all pending traces as illegal, and an optimistic policy regarding all pending traces as both legal and nonblocking. The policies each yield a different function \( f_a \):

\[
\begin{align*}
\gamma_{cons}^N & = f_a^N \cdot f^N(s) = K/s|_N - (\overline{K}/s|_N - \overline{K}/s|_{N-1}) = K/s|_{N-1} \\
\gamma_{optm}^N & = f_a^N \cdot f^N(s) = K/s|_N \cup (\overline{K}/s|_N - \overline{K}/s|_{N-1})
\end{align*}
\]
The conservative policy yields a legal language consisting of all strings of length $N$ from the original legal postlanguage of the current trace $s$—\textit{minus} all pending traces $\left(\overline{K}/s|_N - \overline{K}/s|_{N-1}\right)$—that is, all strings in the legal postlanguage of $s$ of length $N - 1$. The optimistic policy yields a legal language consisting of the same legal postlanguage—\textit{plus} all pending traces $\left(\overline{K}/s|_N - \overline{K}/s|_{N-1}\right)$—that is, all strings in the legal postlanguage of $s$ of length $N - 1$, as well as (possibly) some pending strings \textit{not} in the legal postlanguage. Often state marking is only considered with respect to the plant language—typically in situations in which only the plant is a direct model of the states of a machine to be controlled, while the specifications have states engineered only to produce the correct legal language. In such cases, every state of the specifications is considered marked. Thus $K$ is always prefix-closed, which results in simplification for the optimistic case:

$$f_{a,\text{cons}}(\overline{K}/s|_N - \overline{K}/s|_{N-1}) = K/s|_{N-1}$$

$$f_{a,\text{optm}}(\overline{K}/s|_N - \overline{K}/s|_{N-1}) = K/s|_N$$

The closed-loop behaviour $\mathcal{L}(G, \gamma^N)$ of the plant under the limited-lookahead control policy is defined using recursion:

1. $\epsilon \in \mathcal{L}(G, \gamma^N)$;

2. $\forall s \in \mathcal{L}(G, \gamma^N) \mid \forall \sigma \in \Sigma \cup \{\epsilon\} \mid s\sigma \in \mathcal{L}(G, \gamma^N) \Leftrightarrow \sigma \in \gamma^N(s)$

Finally, $\mathcal{L}_m(G, \gamma^N) = \mathcal{L}(G, \gamma^N) \cap \mathcal{L}_m(G)$ is the language marked by the closed-loop system under the limited lookahead policy. Chung et al. define a limited-lookahead supervisor’s \textit{validity} by its equivalence to a normal complete-lookahead supervisor:

$$\mathcal{L}(G, \gamma^N) = \sup\mathcal{C}(\overline{K})$$
In general, the limited lookahead supervisors presented in this work will not be valid by this definition, as they make tradeoffs between the nonblocking and permissiveness properties described in Section 2.2.2. More relevant is the definition of a runtime error:

$$
\mathcal{L}(G, \gamma^N) \cap f^N(s) = \emptyset
$$

indicating that there are no allowable events from the system state resulting from event sequence $s$. Chung et al. show that in a system with a conservative lookahead policy, if a runtime error does not occur in the initial state, it will never occur in the execution [5].

### 2.3.2 Dynamic DES

Where the work of Chung et al. focuses on retaining in the limited lookahead framework the maximum number of provable properties from standard DES, Grigorov and Rudie [15] generalize the static DES paradigm for the case of a temporally mutable plant—one which can change between event firings. Since the plant can change at any time, any guarantees regarding nonblocking and permissiveness properties are no longer relevant, and the purpose of control moves from strict prevention of constraint violation to avoidance.

This shift in attitude toward supervisorial control is evident in two major formalism changes: first, in the discarding of the marked states concept, which Grigorov and Rudie replace with a value function $v(s)$ in the standard search-space sense; second, representation of plant mutability through the appearance and disappearance of plant modules—whose structure and behaviour are not necessarily known prior to runtime—at each step in the event sequence. At a given time $i$, the plant is denoted
by $G_i = ||M_i||$, the shuffle of all elements of the set:

$$M_i = \{M_{1i}, M_{2i}, \ldots, M_{ni}\}$$

Their work follows with metrics for system reliability and performance. Since the plant neighbourhood generated by limited lookahead represents a forecast based only on the existing set $M_i$, it has an inherent limitation in its reliability with respect to future plant changes. Given a probability $p$ of plant change, Grigorov and Rudie define the plant reliability—intuitively, a measure of how long one can expect the plant to remain stable—as

$$R(t) = (1 - p)^t$$

where $t$ is an integer number representing the future event sequence length. The value function defined for future event sequences can then be normalized to account for the probability such event sequences will actually be accessible:

$$V(s) = v(s)N(R|t|)\frac{v(s)}{v(s)}$$

where $v(s)$ is the value function and $N$ is the normalization function. This function is in effect a limited lookahead policy, based on a criterion other than simply incidence of the lookahead boundary. Such a policy will be shown not to be necessary in this work’s treatment of limited lookahead due to a differing treatment of plant mutability based on Petri nets (see Section 2.4).

Grigorov and Rudie propose a limited-lookahead search algorithm based on an $A^*$/game tree search in which controllable events are treated as protagonist moves, the values of which are to be maximized, and uncontrollable events as antagonist moves, the values of which are to be minimized. They further show this search method to be provably as safe as the optimistic search of Chung et al., and through
the use of a monotonically overoptimistic search heuristic, guarantee that the search is optimal with respect to its value function. The search heuristic employed is the potential:

\[ M = \lceil (tr + v)N(R(t + d))^{\frac{r+d}{r+d}} \rceil \]

where \( v \) is the (unnormalized) value function, \( d \in \{0, N\} \), \( N \) is the lookahead limit and \( t \in \{1, N - d\} \). The assumption is made that the value function increases by at most \( r \) in a single step.

The work by Grigorov and Rudie represents an expansion of limited lookahead to a generalized search function over system states. As shall be shown, some aspects of this paradigm are applicable to software control, while others (specifically state valuation as a continuum) are not.

2.4 Petri Nets

2.4.1 Petri Net Definitions

Many variants on Petri nets have been described in the literature, but the broadest definition is simply as a four-tuple:

\[ N = \{P, T, I, O\} \]

where

- \( P \) is the set of places;
- \( T \) is the set of transitions;
CHAPTER 2. BACKGROUND

- $I : 2^P \rightarrow T$ is a set of inputs from places to transitions; and
- $O : T \rightarrow 2^P$ is a outputs from transitions to places.

The above definition is amenable to representation as a bipartite directed graph with places and transitions as nodes. An arc from a place to a transition specifies an input to the transition; an arc from a transition to a place specifies an output from that transition.

In a marked Petri net, each place can be marked with an integral number of tokens; the overall state of the net is specified by the distribution of tokens in places, called the marking of the Petri net. If the places are indexed, the marking can be represented as a vector of nonnegative integers. A transition may fire if every input place has at least one token; firing entails the removal of a token from each input place for every arc from that place to the transition, and addition of a token to each output place for every arc from the transition to that place. This thesis makes exclusive use of marked Petri nets.

In a deterministic marked Petri net under the nonconcurrency assumption (an assumption that two transitions never fire with exact simultaneity), each marking/-transition pair uniquely defines a subsequent marking via a transition function similar to that of the finite automaton. The transition function therefore defines a graph—called the reachability graph—which is a (possibly infinite) state automaton. The reachability graph is constructed recursively as follows:

1. Define a node corresponding to the initial marking of the Petri net.
2. For each generable event at the current marking, define a node corresponding to the marking resultant from that event, and connect the two nodes with a directed arc labelled with the event symbol.
A Petri net with a bounded number of total tokens is called a *bounded Petri net*; a net with a constant number of total tokens is called *safe*. Clearly a safe Petri net is also bounded.

![Petri Net and Reachability Diagram](image)

**Figure 2.3:** A simple Petri net and its reachability graph.

Figure 2.3(a) depicts a simple Petri net with three places and three transitions. This particular Petri net is safe; there are always exactly two tokens in the net. The initial marking is \([1, 1, 0]\); the language generable by this net is

\[
\mathcal{L}(PN) = (\beta(\alpha\gamma|\gamma\alpha))^* (\beta(\alpha\alpha|\gamma\gamma))
\]

which is a regular language; as such it must be representable by a finite state automaton; indeed, the safety property of the Petri net guarantees a finite number of states in the reachability graph, which is depicted in 2.3(b).

The treatment of start and final (accepting) states of a Petri net differs widely throughout the literature. In the case of marked Petri nets, a common and intuitive approach is simply to define a specific *initial marking* and a set of *accepting markings*. Often the initial marking consists of a single token, which starts in a place designated the *starting place*.
2.4.2 Petri Net Characteristics

The strength of Petri nets is in the modelling of resources which move concurrently through a series of states. As such it is an ideal formalism for concurrent threads, each of which is modellable as finite state automata. The simplest class of Petri nets is that of *state machine* Petri nets: those with exactly one input place and one output place per transition. Such nets mimic the action of finite state automata (whereby the flow of a token in the Petri net informs the state changes of the underlying automaton), and as such accept regular languages, though in the general case, Petri nets have more expressive power than finite state automata; where the latter accept/generate exactly the family of regular languages, Petri nets accept a strict superset of those languages [23]. In fact, the set of Petri net languages is a strict superset of the family of *bounded context-free languages* [23]; however, it is incomparable to the context-free languages in general; there exist context-free languages not generable by Petri nets, and there exist Petri nets which generate languages that are not context free [23].

Additional expressive power generally comes at a cost; that cost manifests itself as a smaller set of questions with tractable answers. In particular, the reachability of an arbitrary marking and general liveness of a Petri net are two questions of obvious value to the concurrency control domain; since constraint specification is expressed in terms of illegal plant states (i.e., Petri net markings), determining the reachability of such states is necessary, and liveness encompasses the question of whether areas of the reachability graph are blocking. While both reachability and liveness have been shown to be decidable in a marked Petri net (they are in fact equivalent), they are not solvable in polynomial time in the general case [16, 21].
2.4.3 Extended Petri Nets

The expressive power of Petri nets is extended even further by means of an added construct: the inhibitor arc. This second class of arc introduces a new element to the firing semantics of the net; namely, it enables the firing of a transition only when the adjoining place is empty. The inhibitor arc allows for “zero-testing” of a place, something not possible in a conventional Petri net; a net with such an arc is called an extended Petri net.

Figure 2.4: A simple extended Petri net.

Figure 2.4.3 depicts a simple extended Petri net. The inhibitor arc is indicated by a circle replacing the usual arrowhead. In this Petri net, the transition $\alpha$ can fire repeatedly until a $\gamma$ transition fires; thereafter, $\gamma$ can fire repeatedly, but $\alpha$ can no longer fire until as many $\beta$ firings occur as $\gamma$ firings; moreover, the number of $\gamma$ firings is restricted to the number of total $\alpha$ firings that have occurred since the initial marking. This yields the language

$$\mathcal{L} = (\alpha^{N_i}(\gamma^{M_{ij}}\beta^{M'_{ij}})^{I_i})^I$$
where

\[ i \in (1, I), \quad j \in (1, J_i), \]
\[ I \in \mathbb{Z}^+, \]
\[ N_i, J_i \in \mathbb{Z}^+, \]
\[ M_{ij} < N_i, \]
\[ \forall J'_i < J_i \quad \Sigma_{j=1}^{J'_i} (M_{ij} - M'_{ij}) \geq 0, \quad \text{and} \]
\[ \Sigma_{j=1}^{J_i} (M_{ij} - M'_{ij}) > 0 \Rightarrow (\forall k > i, \quad j < J_k \quad N_k, M_{kj}, M'_{kj} = 0) \]

Needless to say, inhibitor arcs add considerable expressivity and complexity to a Petri net; indeed, the addition of “zero-testing” yields Turing machine capability in extended Petri nets, with all the attendant expressive power and decidability characteristics entailed. With an extended Petri net, any codable constraint specification is expressible; however, issues such as reachability and liveness move from being merely intractable to insoluble in theory.
Chapter 3

Method, Model and Algorithm

3.1 Petri Net Concurrency Model

In high-level languages, the concurrency model is usually based on the idea of *threads*: individual sequential program descriptions written as procedures (or as object members/methods in object-oriented languages). In C, the *fork* and *join* calls allow a single sequence of code to “split” into two, both continuing execution from the same point—this requires careful crafting of the code in order to make both resultant sequences useful, but has the effect of one thread *spawning* another. In Java, threads are modelled as everything else is—as objects. These objects have specially designated methods which, when called, are run in a separate schedule by the processor. The concept of thread spawning—the idea that “threads beget threads”—is prevalent throughout most high-level languages with concurrency support.

The result is that, unlike the nondeterministic plant mutability assumed in the Dynamic DES paradigm of Grigorov and Rudie (Section 2.3), in software there is the
luxury of knowing exactly the conditions under which a new thread may be created—the point at which the code of another thread starts it. Likewise, the conditions for thread termination are easily determined, at least if the possibility of runtime exception is excluded. A final contrast with Dynamic DES is that modules, since they are derived from predetermined thread classes, come from a finite number of automaton structures, all of which are known in advance of runtime.

As a result, the Petri net structure of thread models is that of a group of Petri subnets of the state machine family connected by arcs from transitions representing thread creation events to places representing the initial states of the new threads. Because the flow of tokens “cascades” through states both within a state machine and from one state machine to another, this type of Petri net will be termed a cascading state machine (CSM) Petri net. For example, a cascading state machine Petri net might be generated from the simple code snippet given in Figure 3.1.

```java
public class ThreadA extends Thread {
    public void run() {
        // sup. event: createB
        ThreadB t = new ThreadB();
        t.start();

        /* Thread A does
         * work here. */

        /* sup. event: A
         * log("Thread_A_terminated.");
         */
    }
}

(a) Thread A

public class ThreadB extends Thread {
    public void run() {
        /* Thread B does
         * work here. */

        // sup. event: B
        log("Thread_B_terminated.");
    }
}

(b) Thread B

Figure 3.1: Simple two-thread Java code snippet.

Figure 3.2 depicts such a cascading state machine Petri net constructed from the
finite state automaton representations of two threads. Thread A is shown as active in the initial state, from which it fires event \texttt{createB} causing it to transition to its second state, and bringing an instance of Thread B into existence. Both threads are then free to fire their subsequent events.

![Sample cascading-state-machine Petri net.](image)

The construction given above suffices for modelling dynamic thread creation in the plant. The specifications can be expressed in virtually any way: via finite state automata, ordinary or extended Petri nets or even arbitrary code in the target language. Since the lookahead mechanism (see Section 2.3.1) eventually searches the closed-loop system by firing plant and specification in lock-step, no specific algorithm is required to compose the plant and specification elements.

For the purposes of the examples detailed here, specifications shall be expressed as Petri nets. In the example above, one may wish to enforce an ordering \{A, B\} on the events A and B fired by the underlying threads. Two specifications effecting such a constraint are given in Figure 3.3.

Worth noting is that these two expressions are not \textit{in general} equivalent, but are equivalent in the context of the given plant. The net of Figure 3.3(a), in addition to ordering, also enforces a bound on the number of firings of B—only one per firing
3.2 Issues and Modelling Considerations

3.2.1 Control Flow and Controllability

The proposed supervisor exerts control upon the target code in only one way: by instructing one or more threads to suspend execution until a condition is met. This action is called *blocking*\(^1\) the thread, and a thread in this state is also said to be *blocking*. The actual control flow of the thread is not exposed to control by the

---

\(^1\)In an unfortunate collision of terminology from the software engineering and discrete-event systems worlds, the term *blocking* has perforce two meanings in this work: in the former sense, simply the suspension of a thread, and in the latter the equivalent of livelock—the inability to reach a terminal state.
supervisor; once a particular execution sequence is chosen by the thread, that sequence must eventually occur. The supervisor controls the overall execution sequence of the plant merely by postponing the individual threads’ sequences in order to realize a global outcome.

In discrete-event systems, the controllability property fits this modelling artefact exactly. During code instrumentation, events are defined for every entry and exit to a control flow block, and for code points which are considered specification-relevant: that is, points that have the potential to cause the specifications to be violated. The former shall be termed control-flow events and the latter concurrency events. A critical design decision for supervisorial control is the mapping of these two classes of events to the controllability partitioning of the prospective model’s alphabet; control-flow events are mapped to $\Sigma_u$ and concurrency events to $\Sigma_c$.

The following example outlines the motivation underlying this design decision. Consider a thread, Java code for which is provided in Figure 3.4.

```java
public class ThreadA extends Thread {

    public void run() {
        if(condition)
            // sup.event: alpha
            executeAlpha();
        else
            // sup.event: beta
            executeBeta();
    }
}
```

Figure 3.4: Thread with two controllable paths.

Under transformation to an automaton, this thread has two concurrency events $\alpha$ and $\beta$, and two control-flow branches. There are three control-flow events (one for
each possible branch, and one for thread termination). Figure 3.5 gives two possible automata for this code under transformation: while 3.5(a) shows the result of a transformation simply representing the two possible concurrency outcomes, 3.5(b) explicitly represents the control flow choice made by the thread independent of the concurrency event actually occurring.

![Diagram](image1.png)

(a) Without Control-Flow Events  
(b) With Control-Flow Events

Figure 3.5: An automaton representation of Thread A with and without explicit representation of control-flow events.

Next, a potential specification of constraints is applied to this thread. Figure 3.6 gives a very simple automaton which represents universal disablement of $\alpha$, and Figure 3.7 shows the subset of plant behaviour that is legal with respect to this specification.

![Diagram](image2.png)

Figure 3.6: An automaton specification disabling $\alpha$.

Finally, we speculate that the value of the boolean $\text{condition}$ is true. In the
Figure 3.7: Automaton representations of Figure 3.5 under the control of the specification depicted in Figure 3.6.

In summary, the scenario omitting explicit control flow mapping admits deadlocked states not detectable by the supervisor; under the adopted paradigm—mapping control flow to uncontrollable events and concurrency-critical points to controllable ones—this class of deadlocks is taken into account by the model.

\[\text{supcon} \] calculation would eliminate state 1 due to its having uncontrollable events leading to the blocking state 2.
3.2.2 The Marked State Problem

The limited lookahead paradigms discussed in Section 2.3 both run into the problem of lack of information with respect to the discovery of marked states. Chung et al. attempt to characterize the behaviour of limited lookahead with respect to policies for dealing with missing information about blocking and coreachability, while Grigorov and Rudie dispense with the notion of marked states altogether for a more generalized value function indicating desirability on a node-by-node basis. Taking into consideration the requirements of concurrency control, it is apparent that neither paradigm is entirely suitable.

The first problem with respect to concurrency control is how to define a marked state to begin with. A marked state is in essence a state in which it is acceptable to block; what a software designer might envision as the criteria for such acceptability is not clear a priori. The most obvious candidate is perhaps the terminal state of each individual thread; blocking is then defined exactly in terms of thread liveness, as one requires a complete path to termination for every thread. However, this path length is indeterminate, and indeed can grow with the number of threads, making it untenable for a limited lookahead of constant-bounded depth.

Some problems may have solutions which involve marking intermediate “idle” states; these may represent simply the cessation of thread activity, a wait state for a server, or the completion of a concurrency-critical task. It is not clear how one might formalize the criteria by which a given plant can be judged controllable within a given lookahead depth.

Like Grigorov and Rudie, then, this work regards the idea of a marked state as fundamentally incompatible with limited lookahead. A new criterion for blocking is
required, therefore, that acts conservatively with respect to control policy (i.e., still guarantees the nonblocking property) but can enact control decisions based only on the information provided within the specified lookahead depth.

The solution to the problem of marked states is the redefinition of “nonblocking state” such that any disabled event is eventually enabled within the lookahead window along any path in the subtree of such a state. This is not only determinable by definition within the lookahead depth, but maintains the blocking guarantee (with a certain qualification to be described—see Section 3.2.3) and can be performed in only the time required for one tree traversal—that is, no extra time is required above that for the state illegality check. We call this definition the re-enablement criterion.

It is instructive to refer to the controllability partitioning described in Section 3.2.1. As a result of control flow being explicitly mapped to uncontrollable events, it is apparent that any module (finite state automaton or Petri net representation of a single thread) has at most one controllable event per state: for multiple concurrency-relevant code points that are part of the same control-flow block (each necessarily annotated with a separate controllable event) imply a control-flow decision made prior to the execution of whichever point is ultimately traversed. This control-flow decision would be annotated with a set of alternative uncontrollable events, separating the controllable ones into different states.

Thus a controllable event that is disabled but generable by the plant indicates the existence of a blocked thread. To prevent a system deadlock (the permanent blocking of one or more threads), the event must be reenabled at some time in the future, unless the event is never generable from the current state.

If the re-enablement criterion is met for every event, then no thread will remain
deadlocked in perpetuity; ergo, the system is nonblocking (again, subject to an important proviso to be outlined in Section 3.2.3). Clearly the criterion is a very conservative one; it may be that an event that remains disabled for a future event sequence length $N$ may be reenabled at step $N+1$, so the system is not necessarily maximally permissive.

So motivated to redefine the property of a nonblocking state, we derive a new policy function $\gamma^N_b(s)$ as per the pipeline outlined in Section 2.3.1 in which the policy function $f^N_a(s)$ uses the re-enablement criterion rather than depend on marked states. The new policy function can be described with the aid of some extra notation:

**Definition 1.** A string $s$ is said to contain an event (or symbol) $\alpha \in \Sigma$ if there are strings $x$ and $y$ whose sequential concatenation with $\alpha$ yield $s$:

$$s \sqsupseteq \alpha \equiv \exists x, y \in \Sigma^* | s = x\alpha y.$$  

**Definition 2.** For a given $\alpha \in \Sigma_c$, the $\alpha$-support of a language $\mathcal{L}$ is the set of all strings in $\mathcal{L}$ containing $\alpha$:

$$\mathcal{L}|^\alpha = \{s \in \mathcal{L} | s \sqsupseteq \alpha \}$$

Finally, we arrive at the language produced by our desired definition of a nonblocking state, which becomes effectively our policy $f^N_a$. Since a state has only one postlanguage (but many possible traces), we avoid introducing explicit notation for states by referring only to their postlanguages. A nonblocking postlanguage of a language $\mathcal{L}$ after $s$ is the right-quotient language with respect to the intersection over all controllable events of their respective event-supports of the legal language, truncated to the window depth. In other words, exactly those strings of the postlanguage of $\mathcal{L}$ which are prefixes of legal strings of less than the lookahead length in every
event-support. We therefore wish to set our lookahead policy to equal this language:

\[
 f_a^K = \left( \frac{L/s}{\cap_{\alpha \in \Sigma_c} K^{[\alpha]}_N} \right)
\]  

(3.1)

The algorithm implementation that achieves this is described in Section 3.4.2.

### 3.2.3 Lookahead Length

Determining the lookahead length is heavily problem-dependent. The two main factors in determining the appropriate length are:

1. System resources, and
2. Blocking characteristics of the plant.

The amount of system resources used is dependent upon the number of nodes generated at a given lookahead depth. The ability to estimate the lookahead length to which the state space can be economically explored is an important avenue of future work (see Section 6.2.1). Winacott et al. [38] have explored ways to quantify the system resources used by lookahead at a given depth; see Section 6.2.1.

Because the lookahead algorithm is conservative with respect to event disablement, if an event is disabled it is assumed to imply blocking unless it is shown to be re-enabled in a future state. Thus in such a scenario the algorithm overall enforces the nonblocking property. However, the algorithm is not guaranteed to be nonblocking in the general case.

To illustrate this, we consider the following small example of a plant/specification Petri net, depicted by Figure 3.8. To show the entire closed-loop system, the plant and specification are included in a single diagram; the dashed lines indicate the specification.
It is clear that the plant initially has two generable choices: \( a \) or \( b \). Were \( a \) to be executed, \( t \) would become generable. With five \( t \) firings, \( c \) would be permanently disabled; to prevent this, \( t \) would have to be permanently disabled. Either case leads to permanent blocking of at least one thread: the one waiting on the disabled event. As a result, blocking is inevitable if \( a \) is generated at the outset; this is not apparent, however, until at least six event firings later. We call this scenario a blocking well: a subtree which, once entered, guarantees blocking regardless of the future event sequence. A blocking well has an associated depth (the blocking well depth); this is the length of the shortest sequence leading to a blocking state.

If the lookahead length is shorter than the blocking well depth, the lookahead algorithm will allow entry to the blocking well. At some point after further event firings, the lookahead will fail to find a safe execution path and will terminate with a runtime error when the current state is marked blocking.
Conversely, as long as the lookahead length is longer than the depth of any blocking well in the closed-loop system, it will be guaranteed to produce a nonblocking result (though maximal permissiveness is not guaranteed).

In formal terms, a blocking well is a nonempty postlanguage $L(G, \gamma^N)/s$ under lookahead to $N$ of some trace $s$ with an empty right-quotient with respect to further lookahead to $N + n$:

$$L(G, \gamma^N)/s \neq \emptyset$$

$$\left(L(G, \gamma^N)/s\right) / \left(L(G, \gamma^{N+n})/s\right) = \emptyset$$

in other words, the entire postlanguage is illegal after $N + n$ steps beyond trace $s$, but this fact is not known until beyond $N$ steps. If a blocking well is entered (i.e., a trace $s$ as described above is executed), then within $n$ the legal postlanguage $L(G, \gamma^N_b(s))$ is empty, yielding a runtime error as described in Section 2.3.

Determining whether a blocking well exists in the reachability graph of a given Petri net may very well not be computable on any order of time shorter than that required to compute the reachability graph itself. Further investigation of this aspect of lookahead is required.

### 3.2.4 Specification Expression

As mentioned in Section 3.1, specifications can be expressed in any way that can be mapped to a boolean function evaluating event legality in a given state. In this work, attention has primarily been given to using Petri nets due to their intermediate expressive power, as well as to maintain a uniform conceptual framework when reasoning about the plant and specifications together. It is trivial to compose a complete closed-loop system of plant and specifications, merely by lining up transitions
labelled with the same event in the individual modules, as long as transitions within each module are uniquely labelled with events, and with only one event per transition. General Petri nets do not have these two properties; nonetheless, the latter is a direct result of our mapping of supervisor calls to transitions and to the fact that each call takes exactly one argument (that being the event label), and the former we are at liberty to enforce in our choice of the value of that argument.

Since this framework can deal with arbitrary numbers of threads, the expressive shortcomings of automata simpler than Petri nets becomes apparent. In particular, finite state automata are insufficient to express any constraint involving unbounded numbers of threads. A good example is an unbounded mutual exclusion (mutex) constraint, involving critical code sections in two classes of threads, A and B. There can be multiple threads running in each of the two classes, and we wish to enforce a maximum of one thread of class B entering its critical section at one time (prohibiting any other thread of either class from doing so), but allow multiple threads of class A to enter their critical sections simultaneously, prohibiting only threads of class B at such times.

This problem is essentially the so-called Readers/Writers problem [8] adapted to a scenario involving multiple dynamically generated readers and writers which access a single shared resource (the interaction with which is denoted by the critical code sections). Intuitively, one would desire the writers to operate in complete exclusion, while readers can operate simultaneously. Any representation as a finite state automaton would require an upper bound on the number of readers, as the automaton would effectively use states to “count” the number of readers present. Less obvious
is that an ordinary Petri net is also incapable of unbounded simultaneity of readers; this is because it is incapable of zero-testing—there is provably no way to check the emptiness of a Petri net place [23]. Zero-testing is exactly the ability for which inhibitor arcs (Section 2.4) are introduced; it allows an unbounded place to be tested.

![Diagram](image)

(a) Finite state automaton

(b) Ordinary Petri net

(c) Extended Petri net

Figure 3.9: Representations of the mutex specification in three different structures. Only 3.9(c) allows an unbounded mutex.

Figure 3.9 depicts the mutex specification as a finite state automaton, an ordinary
Petri net and an extended Petri net. The finite state automaton must reserve at least one state for every possible thread count; since the state count is finite by definition, so is the possible number of threads (two, in the case depicted). The ordinary Petri net can only disable an event when tokens are depleted from a place; the structure must be such that an enabling place contain a token count which is decremented as threads are created. Thus, though the initial token count can be chosen to be arbitrarily large, it remains finite, as does the thread count (nine, in the case depicted). Only the extended Petri net is able to represent an unbounded mutex, since an event can be disabled upon the presence of tokens through use of the inhibitor arc.

As mentioned, the use of extended Petri nets amounts to the expression of Turing machine languages. This in particular motivates the use of limited lookahead, as determining the legality of an arbitrary event sequence is equivalent to determining the membership of a given arbitrary string in the legal language $E$; this is not possible in general for Turing machine languages. A limited lookahead window imposes a sequence length after which it “gives up”, effectively working around the issue of halting.

### 3.3 Online Supervisor Synthesis

The objective of the supervisor is to use the software model and constraints to effect control decisions at runtime. The process followed in creating the dynamic supervisor is largely analogous to that of Dragert [12] with the following caveats. In the static case, the closed-loop system (i.e., the model of the plant under control) is simply converted into a control map (see Section 2.2) which is consulted as events are fired.
by the plant. In the dynamic case, this construct is replaced by an online limited-lookahead search. The supervisor retains separate models for each system component: a composite Petri net for the plant and separate specifications in the chosen modelling formalism. At runtime these are traversed in lockstep by the lookahead algorithm to determine the eligibility of each event to fire.

The methodology is described in terms of a pipeline process. The inputs of the process are:

1. the source code of the software to be controlled—the raw source;
2. knowledge of the points in the code which are concurrency-relevant—the event information;
3. the constraints to be imposed on the software.

The process is composed of:

1. Source code markup;
2. Thread model generation;
3. Constraint specification;
4. Supervisor model generation;
5. Supervisor code translation;
6. Supervisor call insertion.

The flow chart of Figure 3.10 depicts the process and indicates the inputs and outputs of each step of the enhanced methodology based on dynamic lookahead. The steps are individually treated in the subsequent subsections, one subsection for each step. The focus is on highlighting the differences between dynamic lookahead-driven control and the static case.
Figure 3.10: Flow chart depicting the Supervisor autogeneration process.
3.3.1 Source code markup

This phase is largely the same as that of Dragert [12]: the raw source code is taken as input by a developer or designer and annotated to indicate the concurrency-relevant points\(^3\). One principal difference, due to the modelling of control flow via uncontrollable events, is that it is not necessary to explicitly mark up the control flow. Dragert discusses ways to reduce the state space generated by control flow, and mentions the need to manually identify so-called “implicit events”; these are control-flow related events, such as if statements with no else clause, where a possible branch of control flow is not represented by a token in code. Dragert’s solution is to (manually or automatically) append an explicit transition (see Figure 3.11).

\[\text{if (x)}\]
\[\text{foo();}\
\[\text{//Event E}\
\[y = 5;\]

![Figure 3.11](image)

Figure 3.11: An “implicit” event - control flow must be modelled for the else case, but has no direct token representation in code. Source: [12]

The controllability/control-flow partitioning allows for the omission of event markup for control flow altogether; the developer need only reason about and add annotations for concurrency relevance, as the next step produces uncontrollable events for all control flow as part of its process.

\(^3\)Note that such a point is located not on a particular line of code, but between two lines of code.
3.3.2 Thread model generation

From the markup, a control-flow graph is generated representing the possible sequences of code instructions. The result is a collection of finite state automata \( \{G_i\} \) in a machine-readable format: one for each thread in the software. A control-flow graph with a separate transition for every code step is not necessary; only control-flow boundaries, such as entry/exit points to \( \text{for}, \text{while} \) and \( \text{if} \) blocks need be represented by event transitions, in addition to the explicit concurrency points marked up in the first step (the former uncontrollable, the latter controllable).

The control-flow graph generation step can be automated; a full treatment is available in [12]. Generation of a control-flow graph from code is a well-known technique, already used in other discrete-event control methods [37]. The implementation contemplated for this methodology is based on the JavaCC parser generator; example grammars already exist for parsing general Java code, with the ability to add inline Java to output arbitrary translations. The idea is to parse out control-flow-related blocks and produce XML in the format read by the IDES discrete-event systems analysis tool [26]. From IDES, the developer can reason about concurrency using the DES representation of the plant, and generate specifications. The individual thread models are then reimportable to the supervisor code to inform the structure of the lookahead.

3.3.3 Constraint specification

Next, the constraints must be translated from high-level assertions (either in formal logic or plain language) to a collection \( \{S_i\} \) of specifications in a modelling domain upon which parallel composition with the plant can be performed. As discussed in
Section 3.2.4, there is no requirement to use finite state automata as the vehicle for expressing the specifications; indeed, their use limits the expressiveness of the constraints to regular languages. Any formalism meeting the parallel composition criterion above can be used; in the work described herein specification expression is accomplished with extended Petri nets. No explicit translation from other codifications, such as temporal logic, has been explored; it has been left to the software designer to synthesize appropriate Petri nets that accomplish the global constraint required. Petri net design for concurrency-related expression appears to be somewhat intuitive, as it explicitly represents the “flow” of multiple objects through a set of states.

The modelling of specifications as Petri nets is functionality not yet available in IDES; however, with the addition of such functionality, along with the control-flow graph generation, an end-to-end automated process would be available to developers for online-lookahead-based control code generation, as the insertion of supervisor code—in this case the lookahead algorithm with imported finite state automaton and Petri net models—is already automated for Java in the proof-of-concept implementation associated with this work.

### 3.3.4 Supervisor autogeneration

It is in this step that the process for the dynamic case significantly diverges from the static case. Rather than a series of compositions resulting in a single automaton, the plant and specifications are embedded in a model which can be used to dynamically explore the space of the closed-loop system.

As with the static case, the step is broken up into three substeps:
(a) **SM Petri net synthesis:**

For each thread automaton $G_i$, a state machine (SM) Petri net $G'_i$ is generated. First, a one-to-one mapping of states from $G_i$ to places in $G'_i$ is defined. Next, for each state $q$ in $G_i$, the outgoing arcs are enumerated by their event labels $\alpha$ and for each resultant state pair $(q, \delta(q, \alpha))$, a transition $T_{q,\delta(q,\alpha)}$ is defined on $G'_i$ such that $I(T_{q,\delta(q,\alpha)}) = \{q\}$ and $O(T_{q,\delta(q,\alpha)}) = \{\delta(q, \alpha)\}$. Each such state machine Petri net $G'_i$ thus models the flow of instances of $G_i$ through its states.

(b) **CSM Petri net synthesis:**

To complete the plant synthesis, the thread subnets are composed to form a cascading state machine (CSM) Petri net representation by connecting the threads’ Petri subnets via *spawning arcs* which model thread creation. Because the individual finite state automata do not have a built-in mechanism for representing the spawning of new threads, a simple mapping of events to thread creation is required. In the prototype provided, this is effected by a simple key/value mapping in a separate file called a *spawn map*. The keys are a subset of the events generable by the plant; the values are a subset of the start states of the individual thread automata—the restriction to start states is mandatory since instances of a given thread $t$ logically always start in the same initial state $q_{0,t}$.

For each key/value mapping $(\alpha, q_{0,t})$ in the spawn map, an output arc is added to each transition $T_\alpha$ labelled with the event $\alpha$ such that $q_{0,t} \in O(T_\alpha)$. The result is the CSM Petri net required, consisting of subnets corresponding to each underlying thread’s finite state automaton connected by spawning arcs to model the creation of new thread instances—this Petri net is the plant $G$.

(c) **Model generation:**
In the static case, a plant $G$ and specification $S$ are composed via the \texttt{supcon} operation to yield the controlled closed-loop system, which can be used as a supervisor to control the plant. In the dynamic case, such a control map would be generated by parallel composition of the plant Petri net’s reachability graph with that of each individual specification. It is the state space defined by this composition that the limited lookahead algorithm traverses. Since it is not computed beforehand, the control map is not directly generated; instead, a system model $M$ is defined simply to be the pair $(G, \{S_i\})$; $M$ is embedded in a \textit{supervisor template} consisting of the supervisor firing mechanism and the lookahead algorithm (to be explored in Section 3.4).

### 3.3.5 Supervisor code translation

While at the interface to the controlled software the supervisor code is called in much the same way as in the static case, the dynamic supervisor differs drastically in implementation. The supervisor template implementation is pre-coded; translation consists of reading in the embedded model $M$ and producing a representation of it in the target language. It is worth noting that $M$ is “black-boxed” with respect to the supervisor; both the supervisor’s firing mechanism and the lookahead algorithm act only on the interface provided by $M$ for transitioning system state, and neither has any notion of its internal implementation.

### 3.3.6 Supervisor call insertion

The final step is the same in the static and dynamic cases—the target code is re-parsed, and calls to the supervisor’s firing mechanism are inserted at every code-point
representing an event: both the controllable events associated with the instrumenta-
tion of step 1 (Section 3.3.1) and the uncontrollable events associated with control
flow automatically generated by step 2 (Section 3.3.2). The supervisor code itself is
linked (statically or dynamically, depending on the language) to the target code.

The next section outlines the action of the supervisor on the target code during
execution.

3.4 Supervisor Runtime Action

3.4.1 Event Firing

At runtime, the sole interface of the target code with the supervisor is through the
firing mechanism, typically implemented as a function/method call which takes as
its sole argument the label of the prospective event to be fired and enacts control
decisions based on the legality of the event. For clarity, we adopt the Java notation
Supervisor.fire(event) denoting the Supervisor object, its fire method and the
argument event. Event labels are represented as simple tags in the String data
type of the target language, where that exists; alternative data types can be used
where appropriate. The event is the sole argument to the function, and the basis, in
conjunction with the internal state of the model $M$, for the control decision.

Figure 3.12 provides pseudocode for the Supervisor’s condition/lock mechanism
by which threads are queued for event firing. The supervisor uses a single object lock
with wait/notify conditions; one condition for each controllable event. Individual
threads, upon reaching a critical code point, enact the Supervisor.fire(event)
call, at which point an attempt is made to obtain the lock (line 2 of the code). If
another thread is already engaged in a fire call, the current thread blocks.

Upon obtaining the lock, the thread consults the model (lines 5 and 6) to determine event legality. If the event requested is uncontrollable or legal, the event request is automatically granted, and the internal state of the model updated. If the event requested is controllable and currently illegal, the thread is instructed to wait on the lock condition for that event (line 7). If legal, the state is updated and the thread proceeds to send a notify signal to all the lock condition objects corresponding to currently enabled events, waking up any pending threads (lines 9 through 13) just before it releases the lock.

An example of the Supervisor’s fire mechanism can be found in Section 3.5.5.

3.4.2 Lookahead Algorithm

Before delving into the algorithm’s description, it is helpful to establish a clear definition of the space it traverses, followed by some critical terms.
The plant itself is at any time in a specific state embodied by the distribution vector of tokens in the plant’s Petri net. Each individual specification indicates event legality by tracking an internal state as well, regardless of its representation (finite state automaton, Petri net, etc.). The model $M$ thus has a well-defined state composed of the aggregate state of all the underlying components; because the lookahead is designed to be indifferent to these components’ implementations, it is to this model state that the term “state” shall refer hereafter. States of individual components shall be referred to as “substates”. To distinguish between the states of the underlying automata and the overall lookahead, the term “node” shall refer to the nodes of the lookahead tree, in which the states of $M$ are thought to be embedded. Some important definitions follow:

**Definition 3.** An event $\alpha$ is legal with respect to a state $q$ if each specification $S_i$ allows the event to occur after firing the event from its current substate.

**Definition 4.** The empty trace is legal. A trace of length $n > 0$ is legal if its prefix $s$ of length $n - 1$ is legal and its suffix of length 1 (whose equivalence to an event we take as defined) is legal with respect to the state $\hat{\delta}(q_0, s)$ (with initial state $q_0$).

If we define $K$ as the language of all legal traces, the foregoing is expressible as:

$$\epsilon \in K$$

$$s\alpha \in K \iff s \in K \land \alpha \in K/s$$

The next definition is critical to the algorithm producing a closed-loop language which conforms to our definition of “nonblocking”.

**Definition 5.** A generable controllable event $\alpha$ is safe to length $N$ in state $q = \hat{\delta}(q_0, s)$ if it is legal in state $q$ or if it is safe in state $\hat{\delta}(q, s')$ for some $s' \in L(G)/s$ of length
Definition 6. A state $q$ is nonblocking to length $N$ if all its generable controllable events are safe to length $N$. A state without this property is called a blocking state.

If $N$ is the length of the lookahead window, then the blocking property is unknown beyond it; thus without loss of precision we will use the term “nonblocking” to mean “nonblocking to length $N$”, where it is clear that the context includes a lookahead length. This definition of nonblocking yields the language desired from Section 3.2.2, as shown by the following contemplation of a lookahead tree, starting at a node in some state $q = \hat{\delta}(q_0, s_0)$, encountering a state $\hat{\delta}(q, s') = \hat{\delta}(q_0, s)$ after exploring a further trace $s'$ in the postlanguage $\mathcal{L}(G)/s_0$. This scenario is depicted with a subset of the lookahead tree in Figure 3.13.

- If $\hat{\delta}(q_0, s)$ is nonblocking under Definition 6, then for every controllable event $\alpha \in \Sigma_c$ the safety property holds, meaning that one of the following is true:
  1. $\alpha$ is legal at $\hat{\delta}(q_0, s)$, implying that $s\alpha \in K|\alpha|_N$. This means that $s \in f_\alpha^N$;
  2. $\alpha$ is legal at $\hat{\delta}(q_0, st)$ for some $t$ in $\mathcal{L}(G)/s|_N$. This means that $st$ is legal, so $s$ is a prefix of a string in $K|\alpha|_N$; thus $s \in f_\alpha^N$.

- If $\hat{\delta}(q_0, s)$ is blocking under Definition 6, then one of its generable controllable events $\alpha$ is not safe, meaning that it is illegal in every state in the nodes of the subtree of $\hat{\delta}(q_0, s)$ of depth $N$; this means $s$ is not a prefix of $K|\alpha|_N$.

The object then is to determine (with $N$ steps) for each node $n$ in the lookahead tree:

1. whether the node’s embedded state is itself legal;
Figure 3.13: Lookahead exploration starting from a node in state $q = \hat{\delta}(q_0, s_0)$ after a prior trace $s_0$. The node currently being explored has state $\hat{\delta}(q, s) = \hat{\delta}(q, s')$ for some trace $s'$ such that $s = s_0s'$.

2. whether each uncontrollable event from $n$ is legal—if one is illegal, then the event resulting in the current state is also illegal;

3. whether each controllable event from $n$ is safe.

Figure 3.14 gives a pseudocode description of the online lookahead algorithm. It recursively explores the tree by firing events (lines 8–14) up to the recursion depth $N$; all controllable events are marked unsafe in the leaves (lines 2–6). On backtracking out of the recursion, each path to a leaf must traverse a node for every controllable event in which that event is legal (lines 16–23); otherwise the event is marked unsafe, and thus the node is marked blocking. In addition, nodes with uncontrollable events
function lookahead (q, n, n')
    if (n = 0 OR n' = 0)
        for each α ∈ Σ
            q.eventIsLegal(α) = false
        q.isBlocking = true
        return
    else
        for each α ∈ Σ
            if α.isGenerable
                if α.isUncontrollable
                    lookahead(M.nextNode(q, α), n, n' - 1)
                else
                    if q.eventIsLegal(α)
                        lookahead(M.nextNode(q, α), n - 1, N')
                    q.isBlocking = false
            else
                q.eventIsSafe(α) = true
            for each α' ∈ Σ - {α}
                if M.nextNode(q, α').eventIsLegal(α)
                    q.eventIsSafe(α) = true
            if NOT q.eventIsSafe(α)
                q.isBlocking = true
        for each α in Σc
            if M.nextNode(q, α).isBlocking
                q.isBlocking = true
        return

Figure 3.14: Pseudocode for lookahead algorithm.

leading to blocking nodes will be marked blocking (lines 27–29).

The result of the algorithm is the assignment of true or false to the nonblocking property of each node q (at depth n) according to the following criteria:

1. for all generable events α ∈ Σc, there exists a node q' reachable from q in N - n steps, such that α is legal at q', and
2. for all generable events α ∈ Σu, all nodes δ(q, α) are nonblocking.
As the object of principal interest is the interleaving of concurrency-critical points, it is desired to traverse at least $N$ such points in the lookahead. If the depth is tracked for the firing of uncontrollable events, then fewer than $N$ concurrency-critical steps will be forecast. To mitigate this problem, a dual recursion tracking mechanism is used: in addition to the global depth $n$, a separate local depth $n'$ is tracked for the traversal of uncontrollable events. This depth is decremented when the recursion follows such events, which correspond to control flow. The global depth is only decremented for controllable events, at which point the local depth is reset to some maximum contiguous uncontrollable traversal depth $N'$. Thus, in general, the lookahead stops on a given trace when a sequence with a total of $N$ controllable events has been explored, in addition to any intermediate uncontrollable ones, or when a sequence of uncontrollable events, uninterrupted by controllable ones, exceeds length $N'$. Figure 3.15 depicts a lookahead tree constructed using the dual recursion tracking mechanism.

3.5 Example

3.5.1 Description

By way of illustration, we present a larger example and follow the methodology from start to finish. The example is, like others in this thesis, derivative of the Readers/Writers problem. The objective of the example is to demonstrate nonblocking enforcement of constraints on a plant example that demonstrates the cascading thread creation property with more than one thread class composing the plant. The synthesis of the cascading state machine Petri net can thus be illustrated.
Figure 3.15: An example lookahead with its nodes annotated by the integer pairs representing dual recursion tracking. Controllable events are indicated with a crossbar on the arc. The depth $n$ is decremented on a controllable event; the depth $n'$ is decremented on an uncontrollable event and reset to $N''$ on a controllable event. Lookahead is terminated when either counter reaches zero.

In this model, readers and writers are generated by a factory object called RW. The RW object toggles between two modes: one in which it creates readers and writers and one in which it replicates (generates other RW objects). Figure 3.16 gives Java listings for the thread objects. The choiceContinue, choiceReadWrite and choiceReplicate functions provide nondeterministic Boolean results to the control flow branching for loop continuation, read/write and replication choices, respectively.

The readers and writers access a single shared resource, the integrity of which
public class RW extends Thread {
    public void replicate() {
        while (choiceContinue()) {
            (new RW()).start();
        }
    }
    public void rw() {
        while (choiceContinue())
            if (choiceReadWrite()) {
                (new R()).start();
            } else {
                (new W()).start();
            }
    }
    public void run() {
        while (choiceContinue())
            if (choiceReplicate())
                replicate();
            else
                rw();
    }
}

public class R extends Thread {
    public void read() {
    }
    public void run() {
        while (choiceContinue())
            read();
    }
}

public class W extends Thread {
    public void write() {
    }
    public void run() {
        while (choiceContinue())
            write();
    }
}

Figure 3.16: The Java code for the R, W and RW objects.

must be preserved.

3.5.2 Markup and FSA Generation

The first step of the method is to identify points of code in each thread that have concurrency implications—those points in which interaction with other threads or with shared resources require enforcement of ordering across threads. This is a software designer-driven step. Figure 3.17 illustrates where markup would occur in the Readers/Writers example in order to enforce the desired specifications (see Section 3.5.3).
The markup format is fairly flexible; it should be ignored by the compiler of the target language, and indicate the desired event to be fired. In this instance, Java comments are used with the string literal “event: name”.

```java
public class RW extends Thread {
    public void replicate() {
        while(choiceContinue()) {
            // event: createRW
            (new RW()).start();
        }
    }

    public void rw() {
        while(choiceContinue())
        if(choiceReadWrite()) {
            // event: createR
            (new R()).start();
        } else {
            // event: createW
            (new W()).start();
        }
    }

    public void run() {
        while(choiceContinue())
        if(choiceReplicate())
            replicate();
        else
            rw();
    }
}

public class R extends Thread {
    public void read() {
    }
    public void run() {
        while(choiceContinue())
        if(choiceReadWrite()) {
            // event: startR
            read();
            // event: endR
        }
    }
}

public class W extends Thread {
    public void write() {
    }
    public void run() {
        while(choiceContinue())
        if(choiceReplicate())
            replicate();
        else
            rw();
    }
}
```

Figure 3.17: The Java code for the R, W and RW objects marked up with concurrency-relevant events (indicated by // event lines).

The reduced control-flow graphs can be regarded as finite state automata, with the state representing the starting point of code marked as the start state, and the state representing the terminal point as the marked or accepting state. Figure 3.18 depicts the finite state automata resulting from this process for the Readers/Writers.
example threads.

In parallel with control-flow graph production, the instrumentation of the marked-up source is automatable via the parser; it consists of embedding calls to the supervisor for each of the control-flow points represented in the FSAs (immediately after the entry and exit to the blocks), and replacing marked concurrency points with supervisor calls.

Figure 3.18: The FSA representations of the R, W and RW objects’ behaviour.

Figure 3.19 depicts the Java code for the example with supervisor calls inserted.
The event labels for concurrency (controllable) events should be part of the markup of step 1; the labels can then be parsed out and used as the event parameter to the supervisor call. Event labels for control-flow (uncontrollable) events can be generated in any automatable way—using a simple sequence, or an abbreviation of their locations in code. In this example the uncontrollable events have been manually given more intuitive labels for the sake of clarity.

3.5.3 Specification Expression

Specification expression is a manual step based on the software designer’s reasoning, at the modelling level, about what sort of global behaviour constraints are required. Specifications can be expressed in any way that can return a boolean permission value to enable or disable a prospective event; in keeping with our modelling paradigm, Petri nets and/or extended Petri nets are used.

Figure 3.20 depicts the specifications to be imposed. The first (Figure 3.20(a)) is the mutual exclusion specification. The mutual exclusion dictates that while an unlimited number of readers may access the shared resource simultaneously, only one writer can do so at a time, to the exclusion of all other readers and writers. This is a specification that is not expressible without at least an extended Petri net (as described in Section 2.4.3) because of the requirement of zero-testing. The second specification, that of priority (Figure 3.20(b)) is an attempt to enforce fairness\(^4\): the number of pending threads of each type—threads which have been initialized, but not yet engaged in reading/writing activity—is limited using the weights \(R_{\text{pend}}\) and \(W_{\text{pend}}\).

\(^4\)In fact, a much stronger constraint than fairness is enforced; this is in fact a safety constraint limiting to a constant bound the number of reader and writer threads that can bottleneck. A fairness constraint would merely guarantee the continuation of all threads after an arbitrary (but finite) time unit, something not enforceable under this paradigm.
public class RW extends Thread {
    Supervisor sup = Sup.getSupervisor();

    public void replicate() {
        while(choiceContinue()) {
            sup.fire("RWrepl_loopT");
            sup.fire("createRW");
            (new RW()).start();
        }
        sup.fire("RWrepl_loopF");
    }

    public void rw() {
        while(choiceContinue()) {
            sup.fire("RWrw_loopT");
            if(choiceReadWrite()) {
                sup.fire("RWrw_ifT");
                sup.fire("createR");
                (new R()).start();
            } else {
                sup.fire("RWrw_ifF");
                sup.fire("createW");
                (new W()).start();
            }
        }
        sup.fire("RWrw_loopF");
    }

    public void run() {
        while(choiceContinue()) {
            sup.fire("RWrun_loopT");
            if(choiceReplicate()) {
                sup.fire("RWrun_ifT");
                replicate();
            } else {
                sup.fire("RWrun_ifF");
                rw();
            }
        }
        sup.fire("RWrun_loopF");
    }
}

public class R extends Thread {
    Supervisor sup = Sup.getSupervisor();

    public void read() {};

    public void run() {
        while(choiceContinue()) {
            sup.fire("Rrun_continue");
            sup.fire("startR");
            read();
            sup.fire("endR");
        }
        sup.fire("Rrun_term");
    }
}

public class W extends Thread {
    Supervisor sup = Sup.getSupervisor();

    public void read() {};

    public void run() {
        while(choiceContinue()) {
            sup.fire("Wrun_continue");
            sup.fire("startW");
            read();
            sup.fire("endW");
        }
        sup.fire("Wrun_term");
    }
}

Figure 3.19: The Java code for the R, W and RW objects instrumented with supervisor calls.
If more than the given limit of readers, for example, are waiting to read, writers will be disabled until some readers have been given a chance to read (the same applies to writers).

![Mutex and Priority Specifications](image)

Figure 3.20: Mutual exclusion (mutex) and priority specifications for the Readers/Writers problem. $R_{pend}$ and $W_{pend}$ are integer “weights” which describe the multiplicity of arcs connecting the respective places/transitions in each extended Petri net.

The principal motivation behind the priority specification is to introduce the potential for blocking: it is possible for the plant, even under the constraints, to enter a state in which both reading and writing are disabled by the number of currently pending writers and readers, respectively. This allows us to demonstrate that the proposed method will prevent such blocking by preventing entry into such states to begin with.
3.5.4 Plant Synthesis

The synthesis of the cascading state machine Petri net representation of the plant is next undertaken, using as inputs the underlying finite state automaton representations connected via spawning arcs as dictated by the spawn map (see Section 3.3.4). Figure 3.21 gives the appropriate spawn map file for the Readers/Writers example.

\[
\begin{align*}
\text{createR} &= \text{Reader} \\
\text{createW} &= \text{Writer} \\
\text{createRW} &= \text{RW}
\end{align*}
\]

At runtime, the supervisor code reads the individual finite state automata from files in their native modelling format (for the purposes of the prototype, the IDES [26] XML format was used), converts them to state-machine Petri nets and connects them via spawning arcs dictated by the spawn map. Such arcs are transition output arcs, connecting the transitions specified by the left-hand value of each key/value pair to the Petri net places representing the start states of the underlying state machines specified by the right-hand value. In this way, a Petri net is generated in which tokens can “cascade” from state machine to state machine, representing the spawning of new threads.

Figure 3.22 depicts the cascading state machine Petri net resulting from the synthesis of the finite state automaton representations of the Reader, Writer and RW classes. The Petri net consists of state-machine Petri subnets each equivalent to the underlying state machine of its respective module. The boxes in Figure 3.22 indicate the state machine Petri net subnets reflecting each underlying thread, and the arcs
crossing the boxes’ boundaries indicate the transition firings that result in new instances of each thread. These are directly informed by the spawn map, and are called spawning arcs. Note that the flow of tokens within each box represent the state-change activity of the underlying automata and hence of individual thread instances of each thread class.

3.5.5 Online Control

Once synthesized, the plant model is retained in memory by the supervisor. The specifications are read in similarly; for the purposes of the prototype, an XML storage format was devised that was similar to the IDES automaton XML format but that allowed many-to-many relationships between places (states) and transitions (events).

Once read into memory, the lookahead tree can be generated by firing plant-generable events in lock-step with the specifications; those events which lead a given specification to an invalid substate (and are thus disallowed by that specification) are marked as illegal; on backtracking, the lookahead marks as illegal any states uncontrollably leading to illegal states, as well as those marked as blocking (under the safety/nonblocking definitions given in Section 3.4.2). Only the first level of children in the tree are retained for control decisions.

As asserted in Section 2.4.1, the current state can be denoted with a vector, each element giving a thread count for each of the possible states of underlying automata. For example, if the Petri net’s places are enumerated as in Figure 3.23, then the vector

\[ [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] \]

denotes the initial marking of the plant; namely, one RW object in its start state, and
Figure 3.22: The cascading state machine Petri net resulting from synthesis of the R, W and RW classes.
no other threads present.

Figure 3.23: An enumeration of places in the plant Petri net.

To see the effect of control decisions upon possible plant behaviours, we consider
the plant in the following state:

\[ [1, 2, 0, 0, 0, 0, 0, 0, 4, 0, 0, 2, 0] \]

which corresponds to the Petri net marking of Figure 3.24.

Figure 3.24: Plant Petri net with potential mutex disablement.
In this state, the generable plant events are `RWrun_loopT`, `RWrun_loopF`, `RWrun_ifT`, `RWrun_ifF`, `endR` and `startW`. Because there are readers in the reading state, the mutex specification is in the state 

\[ [1, 4] \]

which informs the marking of Figure 3.25, and prohibits execution of `startW` via the inhibitor arc. The `startW` event will be disabled by the supervisor until after the execution of four `endR` events.

![Figure 3.25: Mutex specification disabling startW.](image)

Further to the example, we consider the state

\[ [0, 0, 0, 0, 2, 0, 4, 0, 0, 0, 2, 0] \]

which corresponds to the net of Figure 3.26. In this case, the events `RWrw_ifT`, `RWrw_ifF`, `startR` and `startW` are generable. If the values for `R_{pend}` and `W_{pend}` are 4 and 2 respectively, then `startW` is disabled because the read priority specification has a token count of 4 in its one place (Figure 3.27).
Figure 3.26: Petri net with contention over $\text{startW}$ and $\text{startR}$ priorities.
More importantly, we consider the potential execution of $\text{RWrw_ifF createW}$ (of immediate concern in the given state because $\text{RWrw_ifF}$ is uncontrollable). Were this execution to take place, the write priority specification would have two tokens, disabling the event $\text{startR}$. With both $\text{startR}$ and $\text{startW}$ disabled, neither the write nor read specifications’ sole places can ever be emptied; thus neither $\text{startR}$ nor $\text{startW}$ can ever be enabled, resulting in deadlock for those threads waiting on them. As a result, this state must be avoided. Under the proposed algorithm, since no reenablement of the events are found in the subtrees of this state, the state is declared illegal and the $\text{createW}$ event leading to it disabled.

Using the event enablement scenario above to explore the condition/lock mechanism of the supervisor’s $\text{fire}$ method, we focus on the thread states themselves. With $\text{startW}$ and $\text{createW}$ disabled, we consider the threads currently running: two $\text{RW}$ threads, four readers ($\text{R}$) and two writers ($\text{W}$). We contemplate the scenario in which the two writers attempt to initiate writing first, followed by a reader.

1. The first writer, upon starting an iteration of its loop $\text{write}$ block (having just executed $\text{sup.fire("Wrun_continue")}$), executes the method $\text{sup.fire("startW")}$. Entering the supervisor call, it attempts to obtain the sole supervisor lock.
2. Having obtained the lock, it proceeds to consult the lookahead model to determine if \texttt{startW} is enabled. Finding it disabled, the thread uses it as a key to look up the appropriate \texttt{wait} condition and proceeds to \texttt{wait} pending notification of enablement. Waiting automatically releases the lock, allowing other threads to make event firing attempts.

3. The second writer, also ready to write, performs the same actions as the first, resulting in its entering a waiting state.

4. One of the readers, having just executed \texttt{sup.fire("Rrun_continue")}, executes the method \texttt{sup.fire("startR")}. Entering the supervisor call, it obtains the lock, which has been freed by the waiting writers.

5. Consulting the lookahead model, it finds \texttt{startR} enabled and proceeds to effect the state change. As a result, the lookahead algorithm is engaged to advance to the state indicated by \texttt{startR} and expand the lookahead tree by one step. New node safety information is propagated back up the tree by the recursion, possibly updating the enablement status of the new set of imminent controllable events.

6. The controllable alphabet is then iterated over: for each enabled controllable event, the associated wait condition is \texttt{notified}, causing any threads waiting on the condition to resume in an attempt to regain the lock and execute their respective events. However, the writers are not notified as the \texttt{startW} event remains disabled, now due to the mutex rather than the priority constraint.

7. Since the mutex does not restrict concurrent reading, other readers may fire \texttt{startR}; all must fire \texttt{endR} before \texttt{startW} is enabled. When this finally occurs,
the wait condition associated with \texttt{startW} is notified, causing the writer threads to attempt to regain the lock.

Throughout this process, the action of the two \texttt{RW} objects has not been contemplated; their event firings are free to interleave with those mentioned above, subject to allowance by the specifications through the lookahead, in the same manner.
Chapter 4

Results

4.1 Implementation

4.1.1 Size Limitations: Space Versus Time

Initial implementations of the online-lookahead supervisor made use of a persistent tree whose leaves were extended at each step. The general impression was that online lookahead would be both time-intensive and time-critical, since control decisions would have to be made while minimizing the time during which threads would be bottlenecked while waiting for a decision.

Since time performance was perceived to be the critical factor, the Java object structure was kept to a strict minimum; the FSA and Petri net graph models were stored in optimized structures that used simple array index lookup which could be traversed quickly by the lookahead out to the leaves.

It quickly became apparent that the real limiting factor was space, not time. In the Readers/Writers example, explored to a lookahead depth of 10 controllable events,
with a maximum of 10 readers and/or writers spawned by the code, the number of
nodes in the lookahead at a given time varied sufficiently to cause insufficient memory
scenarios under the maximum RAM available in the environments at hand.

![Figure 4.1: Variation of lookahead tree size over a single execution of the simple
Readers/Writers example.](image)

To fix this problem, persistent arcs in the lookahead were eliminated for all but the
top level of children—the level required for the control decision. Since node legality
and safety are accumulated from leaves to the first-level children, the remaining nodes
would be released and garbage-collected on the backtracking phase of the recursion.
The amount of RAM used went from over 3GB to roughly 81MB, making the explo-
ration tenable in the space available. The time cost turned out to be manageable, with
the only significant control decision delays resulting from those times in the execution
window when many threads were engaging in critical sections simultaneously.

The issue of permissiveness is naturally related to the size of the lookahead: the
shorter the lookahead, the more likely a harmless subtree (i.e., a subtree lacking in
unsafe traces) will be disabled due to lack of information on when a disabled event
is re-enabled. A related issue is that of generability by the plant of a disabled event:
a disabled event is entirely harmless if the event is not generable by the plant while
it is disabled. Determining the generability of an event is non-trivial, but possible:
the use of siphon analysis (see Section 5.3) as in Wang and LaFortune [37] could be
used to determine if a given disabled event is never generable by a plant in a given
lookahead subtree. Siphon analysis was not implemented in the limited-lookahead
prototype, and it is unclear if the added overhead would render the lookahead per-
formance unacceptable. Section 6.2 elaborates on this possibility. However, Wang
and Lafortune can only determine if the event is forever nongenerable; not merely
nongenerable for a specific window such as the duration ofdisablement.

In general, the speed of lookahead is a critical factor, and one which showed
itself to be a performance bottleneck for the Readers/Writers example once the space
problem was solved. The space/time tradeoff was only partially investigated and
warrants further research (see Section 6.2)

4.1.2 Caching

In addition to lookahead-tree persistence, the initial implementation contained a node
caching mechanism. The cache consisted of a hash table indexed by the Petri net
marking vectors. For each node generated, a lookup would be attempted from the
hash table; were the node not present, it would be stored along with the depth of
the already explored subtree of which it was the root. If the node was present in
the table, further recursion from that node would only be performed if the current
remaining lookahead depth exceeded the depth stored in the node (which would be
updated to the new depth). This guaranteed the avoidance of unnecessary traversal
of a given subtree which was already deeper than the remaining lookahead depth.

Without tree persistence, caching was rendered ineffective and was removed from
the implementation.
4.1.3 Performance

Performance was measured with respect to the time taken to complete a lookahead tree traversal. For the example runs of a simplified Reader/Writer example (where the RW object was reduced in complexity), the supervisor was instrumented with timestamps at the initiation and return of the lookahead search algorithm. Figure 4.2 illustrates the variation in time taken (in milliseconds; those near zero had an elapsed time on the order of nanoseconds) for the lookahead algorithm to complete, per iteration, over the execution window. This elapsed time remains remarkably consistent in this example, with the exception of the eleventh iteration (at which a large number of threads are in states with multiple pathways). For the most part, the lookahead execution remains a task with relatively negligible cost, save for a few instances when many threads are in contention.

![Elapsed Time for Lookahead Execution](image)

Figure 4.2: Variation of lookahead tree size over a single execution of the simple Readers/Writers example.
4.2 Validation

4.2.1 Ad Hoc Testing

Preliminary testing was performed by modelling simple examples with template Java code. As the control-flow graph generation step was not fully implemented, care was taken to manually specify code and finite state automaton models which accurately reflected each other. The examples were sufficiently simple and transparent to bear manual scrutiny by the developer.

The examples simulated various concurrency scenarios such as mutual exclusion, precedence constraints and general condition-based waits. Constraint violation was simulated with Java `assert` statements which were only reachable if such violation occurred. The code examples were instrumented with random wait times between concurrency-critical code points to obtain broad coverage of different thread interleavings over multiple test iterations. All examples were written such that they would eventually terminate, with a bounded number of threads created.

The simplest such example investigated was a simple precedence problem involving two classes depicted in Figures 4.3 and 4.4. A thread of class $A$ generates multiple threads of class $B$. Unconstrained, all the threads may progress to termination. A precedence constraint (Figure 4.5) inhibits class $B$’s firing of the $B1$ event until $A$ has fired $A2$. The precedence constraint was checked by means of a boolean $\text{prec}$ variable in class $A$ which is set to `true` after $A2$ has fired, and an `assert` statement in class $B$ which checks the value of $\text{prec}$ after firing $B1$; if the constraint is violated, an assertion exception is raised.

The threads were designed to make constraint violation very likely in the absence
of supervisorial control. Indeed, when run with the supervisor deactivated, assertion exceptions were witnessed almost immediately in all executions. With the supervisor activated and lookahead depths ranging up to $N = 25$, all executions ran to completion (the termination of all threads) without constraint violation.

The Readers/Writers example of Section 3.5 was also implemented and tested. In order to check for assertion violation, a separate `Assertion` object was implemented.
CHAPTER 4. RESULTS

Figure 4.4: Finite state automaton representations of the threads in Figure 4.3.

Figure 4.5: Simple precedence constraint implemented with a finite state automaton.
to track the parameters associated with the constraints (Figure 4.6). The \texttt{R} and \texttt{W} objects (Figure 4.7) were instrumented with calls to the \texttt{Assertion} object to track the number of readers and writers in either a pending or reading/writing state (as per the description of mutex and priority constraints given in Section 3.5.3), and \texttt{assert} statements checking the \texttt{Assertion} attributes to raise exceptions in the same manner as the first example.

\begin{verbatim}
package sup.test.example.rw;

public class Assertion {
    public static final double DEFAULT_RUNTIME = 50;
    public static final double DEFAULT_WAITTIME = 1;
    static final int RPEND = 2;
    static final int WPEND = 3;

    static Set<sup.test.example.rw.Reader> readers
        = Collections.synchronizedSet(new HashSet<sup.test.example.rw.Reader>());
    static Set<sup.test.example.rw.Writer> writers
        = Collections.synchronizedSet(new HashSet<sup.test.example.rw.Writer>());
    static Semaphore wmutex = new Semaphore(1);
    static Set<sup.test.example.rw.Reader> rmutex
        = Collections.synchronizedSet(new HashSet<sup.test.example.rw.Reader>());
}
\end{verbatim}

Figure 4.6: Assertion object used for tracking constraint violation in the Java implementation of the Readers/Writers problem.

Again, when run with the supervisor deactivated, assertion exceptions invariably surfaced. With the supervisor activated, all executions ran to completion (the termination of all threads) without constraint violation. Lookahead length ranges up to \(N = 12\) were successful; beyond this, memory limitations caused the software to fail.
package sup.test.example.rw;

public class Reader extends RW {
    static int count = 0;
    Supervisor sup = Sup.getSupervisor();
    Logger log = sup.getLogger();

    public void run() {
        Verify.beginAtomic();
        readers.add(this);
        for(int i = 0; i < LOOPC; i++) {
            sup.fire("Rrun_continue");
            sup.fire("startR");
            readers.remove(this);
            rmutex.add(this);
            log.log("Reading . . .");
            assert writers.size() < WPEND;
            assert wmutex
                .availablePermits() > 0;
            waitabit(DEFAULT_RUNTIME);
            rmutex.remove(this);
            log.log("RMutex :
                + rmutex.size());
            sup.fire("endR");
            readers.add(this);
            log.log("Done .");
        }
        sup.fire("Rrun_term");
        readers.remove(this);
        log.log("Terminating .&");
        Verify.endAtomic();
    }
}

(b) Thread A

public class Writer extends RW {
    static int count = 0;
    Supervisor sup = Sup.getSupervisor();
    Logger log = sup.getLogger();

    public void run() {
        Verify.beginAtomic();
        writers.add(this);
        for(int i = 0; i < LOOPC; i++) {
            sup.fire("Wrun_continue");
            sup.fire("startW");
            writers.remove(this);
            wmutex.tryAcquire();
            log.log("Writing . . .");
            assert readers.size() < RPEND;
            assert rmutex.size() == 0;
            waitabit(DEFAULT_RUNTIME);
            wmutex.release();
            sup.fire("endW");
            writers.add(this);
            log.log("Done .");
        }
        sup.fire("Wrun_term");
        writers.remove(this);
        log.log("Terminating .&");
        Verify.endAtomic();
    }
}

(a) Thread A

Figure 4.7: R and W objects instrumented with assertion to check for constraint violations.
4.2.2 Java PathFinder

With a view to adding a further level of rigour to the testing process, a model-checking tool called Java PathFinder (JPF) was brought to bear against the test examples. JPF consists of a specially-constructed host Java virtual machine (JVM) and a state-space search traversal mechanism (not unlike the online-lookahead algorithm given) which operates on target Java bytecode itself. The JPF JVM is capable of storing the relevant states in the execution path (trace) of the target code in a stack, and backtracking through states to explore alternative interleavings of code. The state space search can be exhaustive, depth-limited, and/or heuristically directed.

Exhaustive searches were attempted against the simple test examples described in Section 4.2.1. After a run of 44 days on a dual-core system with 4GB of RAM, a JPF test run of a simplified Reader/Writer ran out of memory; however, in the 44 days of state-space exploration, no deadlocks or constraint violations were found.
Chapter 5

Related Literature

5.1 Path Expressions

Early expressions of concurrent constraint specifications made use of logical frameworks within which to state assertions about what could or could not be true of the global configuration. Andler [1] makes use of standard predicate logic embedded within the framework of a hierarchical abstraction mechanism called a path expression [2].

Path expressions can be used for subset selection from any kind of hierarchy, from directories to classification schema. A path expression essentially describes a subset of a graph by marking a path through it; a common example of this is the regular expression, which describes a path through an automaton.

An individual path expression can be used to describe a set of possible sequences of allowed operations. Multiple path expressions, describing separate sequences, however, cannot specify any interactions between their targets; a separate notion of state must be provided. Andler uses predicate logic to specify global conditions under
which subexpressions would be valid.

Andler goes on to construct a transformation for predicate path expressions to a nondeterministic program specification which uses command guards [10] with a `cobegin-coend` block to specify parallel processes, and in this way verifies the validity of the specification. Finally, a transformation is defined to finite state automata augmented with pre-state and post-state contracts (called *prologues* and *epilogues*). The state machine (implemented in Algol 68) consists of runtime routines which are called by the target program depending on the state of the machine; the prologue determines whether entry criteria are met, and the epilogue sets a new system state. If the prologue cannot find a legal path, the process is forced to wait until a system state change caused by another process makes it legal to continue.

Andler’s work was pivotal for the expression of concurrency constraints, but did not provide *a priori* for autogeneration of controls.

### 5.2 Synchronization Skeletons

As demonstrated above, predicate logic does not by itself possess the power to fully express concurrent constraints. Rather than insert predicate logic into an abstraction framework to allow the flexibility required (as with path expressions), it is possible to use a temporal logic, which contains all the assertion mechanisms required to fully express concurrency. Temporal logic includes in large part the same operators as predicate logic, but the quantifiers are altered in definition.

Two significant temporal logic frameworks are Manna and Pnueli’s *linear temporal logic* [20, 24] and the *branching-time temporal logic* of Clarke and Emerson [6, 7].
Manna and Pnueli use linear temporal logic to build a system of proof for model-checking of concurrent programs using an invariance principle to check that constraints hold globally along future state sequences. Thistle and Wonham [29] extend this to a DES control framework, though they do not treat software specifically.

The principal drawback of linear temporal logic is the fact the any assertions made about state sequences must apply to all state sequences. This all or nothing expressiveness means that one cannot make statements about multiple selected future sequences; there is only one timeline about which to reason. Branching-time temporal logic, as the name implies, allows for multiple future event sequences, assertions about various of which may hold different truth values. This logic mirrors the logic implicit in concurrent threads. To further extend branching-time, a binary operator is introduced which allows for truth values that hold before or after certain points. This extended logic, called computation tree logic or CTL, is used by Clarke and Emerson [6] to reason about multiple concurrent timelines arranged as a tree of computation (state) sequences. They use CTL as a language for expressing assertions representing specifications over computation trees which model concurrent processes. From this a decision procedure is derived by which the validity of the assertions (hence the conformance of the specifications) is tested against the action of the computation tree. Finally a synchronization skeleton is produced which applies synchronization controls at the appropriate points in the concurrent code.

Clark and Emerson’s method is as follows:

1. Specify the desired behavior of the concurrent system using CTL.
2. Apply the decision procedure to the resulting CTL formula in order to obtain a finite model of the formula.
3. Factor out the synchronization skeletons of the individual processes from the global system flowgraph defined by the model.

Their method represents one of the earliest attempts at control autogeneration, rather than post-facto conformance checking or constraint expressions (as per Andler).

5.3 Petri Nets and SBPI

The most recent work in discrete-event systems for concurrency control is by Wang and Lafortune [34, 35, 36, 37]. In these papers the authors make use of supervision based on place invariants (SBPI), a supervisory control method leveraging structural analysis of a Petri net representation. While the intuitive mapping of processes to tokens does apply, tokens do not represent only processes; they also represent the states of specifications, modelled as segments of the Petri net. Finally, control is effected by means of control places, in which the presence or absence of tokens denotes enablement or disablement of connected transitions.

The authors employ a technique central to SBPI called siphon analysis to determine deadlock properties of a Petri net. A siphon in a Petri net is a place from which tokens that are removed can never be replaced. In a Petri net under fair transition scheduling, eventually any siphon will contain zero tokens, after which any transitions for which it is an input can never fire again. Petri nets with such submarking structures may be in deadlock, under circumstances outlined in [37].

In contrast to our work, the SBPI-based control of Wang and Lafortune addresses only the prevention of mutex deadlock; this is equivalent to incorporating only mutex specifications in our model. In general, our technique is aimed at deadlock-freedom in the context of more complex arbitrary specifications.
5.4 Coordinators and Adapter Synthesis

Further evidence of the predominance of automata-based implementation in this area of research is the Coordinators Synthesis of Tivoli and Inverardi [30]. The authors principal claim is the ability to automatically generate a coordinator for components (under a specific component interface scheme) such that the concurrent action of those components is guaranteed deadlock free.

From a given set of components, built according to a coordinator architecture (sharing a common method signature contract), the authors give a method for generating a labelled transition system (essentially a state machine). Using operations very similar to those of discrete-event systems theory such as shuffle and parallel composition, they are able to combine multiple parallel transition systems.

The authors next describe a method whereby a system of freely interacting components is rearchitected such that all components communicate with each other through a stub coordinator—essentially a module that passes all communication through passively. Such a module therefore allows the full set of interleaving state sequences. A deadlock analysis (consisting of a procedure similar to the supremal control algorithm of discrete-event systems theory) reduces the passivity of the coordinator, disabling component interactions that lead to deadlock.

The resultant coordinated behaviour is deadlock-free and can provide enforcement of a range of global policies.

This method is principally intended to supplement a particular architectural model or design pattern, and as such requires the adoption of an adapter architecture at the outset. This is in contrast to our method, which can be applied post facto to a source code base under arbitrary design.
5.5 Synchronous Concurrency

All of the foregoing rests on the assumption that individual threads are executing asynchronously, and that global constraints dictate synchronisation at critical junctions in the code. Synchronous models, in which code steps are executed simultaneously across processes\(^1\), are widely used in many frameworks [31]; such systems are not in need of the concurrency control mechanisms discussed in this work.

Some research has been undertaken into the development of techniques for embedding synchronous models into the standard asynchronous architectures used today. The problem is one of ensuring that the semantics exhibited by shared memory are preserved across multiple executions of the same code [3]. In the case of distributed systems, however, the problem is more endemic, as each component of the system is intrinsically asynchronous in the absence of a shared clock. The authors of [32] implement a synchronicity layer using a set of queues on top of the normal inter-process communication channels, similar to Kahn Process networks.

---

\(^1\)The term “threads” implies asynchronicity and is generally not used in this context.
Chapter 6

Conclusion and Future Exploration

6.1 Conclusion

Discrete-event systems have already been shown to be an effective framework in which to autogenerate concurrency control code for high-level languages, given a set of assumptions including static foreknowledge of the processes (threads) to be controlled. The framework extension provided in this work accomplishes two goals:

- The accommodation of dynamic thread creation and termination, without such \textit{a priori} knowledge as the numbers and lifetimes of individual threads;

- A solution to the general state-space tractability problem inherent to DES methods, in the form of online limited-lookahead search as a substitute to attempts at precalculation of impossibly large state spaces.

A proof-of-concept implementation and its validation were described, and the limitations of the technique explored. While the basic validity of the technique was established, general performance evaluation was left pending future work.
The technique given is envisioned by its author as a framework upon which additional analysis can be built. In particular, much literature has been produced analysing the structural properties of Petri nets which could be used to aid in the control decision process described herein (such as siphon analysis [37] or the theory of regions [33]). The application of such literature to this particular domain has also been left as future work.

6.2 Future Exploration

6.2.1 Performance

The object of this work being to construct and validate a proof-of-concept of the dynamic-thread control autogeneration technique, a number of critical avenues were left unexplored, particularly in terms of establishing performance metrics. Without this research, the general practicality of the technique is still an open question. In particular, the space/time tradeoff needs better measurement. Work by Winacott et al. [38], currently underway, makes use of the incidence matrix of finite state automata and their eigenvalues to put estimates on the branching factor of a given lookahead tree, and may provide insight into the estimation of the required lookahead depth and by extension, the required system resources.

6.2.2 Petri Net Classes

A very large corpus of literature has been generated studying the classification of Petri nets by structure and function. Petri nets in general, by virtue of their high level of expressivity, are more difficult to characterize than finite state automata. In
this application, the use of extended Petri nets drastically limits the properties that can be proven about the system behaviour.

However, the plant models used in this technique have a very simple and regular structure which is very similar to so-called S*PR nets, which have been widely studied [36]. Further study is warranted to determine if the known properties of S*PR nets can be leveraged to make conclusions concerning the cascading state-machine Petri-net structure.

6.2.3 Supervisor-Aware Plant Control

As mentioned (Section 3.2.1), the only method by which the supervisor controls the plant is through the postponement of thread activity. Once a thread has adopted a particular control-flow path, the supervisor cannot change that path under the current paradigm. An avenue of research well worth investigation is the possibility of allowing the plant to react to control decisions to adopt alternative control-flow pathways. In a high-level language such as Java, this may take the form of a return value from the supervisor call—rather than a denied event transition resulting in simple blocking, a negative return value could indicate to the thread that the event is unavailable, allowing an alternative event to be attempted. This is equivalent to increasing the 1:1 mapping of available (generable) controllable events to threads. In addition, it may aid in the improvement of general concurrent performance, since threads unable to execute their initial choice of control flow can opt for an alternative processing pathway rather than simply blocking, decreasing the level of serialization due to concurrency control.

A simple example demonstrating the potential for increased parallelization is a
pair of threads, $A$ and $B$, with two shared resources under mutex lock (depicted in Figure 6.1). If each thread has to update both resources (and order dependence is not an issue), under the current setting a given thread would choose an order in which to update them. For instance, thread $A$, having to update both shared resources 1 and 2, would attempt to update 1, followed by 2; if 1 is currently locked by thread $B$, thread $A$ must wait.

```java
public class A extends Thread {
    
    public void run() {
        sup.fire("obtainmutex1");
        sup.fire("obtainmutex2");
    }
}

public class B extends Thread {
    
    public void run() {
        sup.fire("obtainmutex2");
        sup.fire("obtainmutex1");
    }
}
```

**Figure 6.1:** The Java code for Threads $A$ and $B$ in the current supervisor context.

Under an SAP paradigm, thread $A$ (likewise thread $B$) could be rewritten to attempt a lock on resource 1; if this fails, a lock could be attempted on resource 2. Such a construct requires a supervisor event call which returns a result indicating the success or failure upon which the calling thread can act. If an event call is denied by the supervisor, the thread can continue without waiting. Figure 6.2 shows how Thread $A$ might be altered to allow for a supervisor-aware event call (assuming the supervisor is rewritten to provide an immediate return value).

A supervisor which could handle both the current and SAP-driven event requests would be akin to other concurrency constructs, such as semaphores and locks, which
public class A extends Thread {

    ...

    public void run() {
        ...
        if (!sup.fire("obtainmutex1"))
            sup.fire("obtainmutex2");
        ...
    }
}

Figure 6.2: The Java code for Thread A with Supervisor-Aware event calls.

can be accessed synchronously (blocking until available) or asynchronously (attempting to obtain without waiting). Indeed, a supervisor of this type might justifiably be seen as a generalization of more primitive concurrency locking mechanisms—one in which arbitrary lock retention rules are specifiable.

Although fairly trivial to implement, the implications with respect to predictable plant behaviour are unknown. Further research is required to characterize the effects of allowing awareness in the plant of supervisory decisions.
Bibliography


Appendix A

Supervisor Source Code

```java
package sup.controller;

import gov.nasa.jpf.jvm.Verify;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.HashMap;
import java.util.Map.Entry;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

import sup.controller.des.AlphabetException;
import sup.importer.BuilderException;
import sup.util.ExceptionHandler;
import sup.util.Logger;

/**
 * Central coordinator for concurrency control. Uses a Model object to
 * track concurrency-relevant events and system state, and renders control
 * decisions upon request of Thread objects.
 * The Supervisor is implemented as a <i>singleton</i> object; it is accessed
 * via the method {@link #getSupervisor()}, not via a constructor (of which
 * none are public).
 * @author aauer
 *
 */
public class Supervisor {
    /**
 */
```

104
The model of the tracked system, generally a Petri Net plant and specifications (though this is not mandatory).

```java
public Model model;
private Logger logger;
private ExceptionHandler handler;
int trans = 0;
public boolean SUP_ACTIVE;
public boolean SUP_JPF;
private HashMap<String, Condition> conditions;
private ReentrantLock suplock;

private static Supervisor ref;
/**
 * Method for returning the global Supervisor, a <i>singleton</i> object with no constructors. In any given application runtime, only one Supervisor can exist.
 * @return The global Supervisor object.
 */
public static synchronized Supervisor getSupervisor() {
    if (ref == null)
        try {
            ref = new Supervisor();
        } catch (Exception e) {
            System.out.println("Supervisor while creating Supervisor:");
            + e.toString();
            e.printStackTrace();
        }
    return ref;
}

private Supervisor() {
    handler = new ExceptionHandler();
    logger = new Logger();
    logger.addPrinter(System.out);
    try {
        logger.addPrinter(new PrintStream(new File("logs/log.txt")));
    } catch (FileNotFoundException e) {
        e.printStackTrace();
        System.exit(1);
    }
    logger.log(Logger.LOG_SUPERVISOR, "Supervisor being generated.", false);
    logger.log(Logger.LOG_SUPERVISOR, "HeapSpace available:");
    + Runtime.getRuntime().totalMemory();
    + "(" + Runtime.getRuntime().maxMemory(), true);
    suplock = new ReentrantLock();
}
/**
 * The Supervisor uses a Logger as its logging mechanism, and makes it 
 * available to other objects, including the threads under control, so they 
 * can all log to the same place(s). 
 * 
 * @return the Supervisor’s own Logger 
 */
public Logger getLogger() {
    return logger;
}

/**
 * The Supervisor uses an ExceptionHandler as its error handling facility, 
 * and makes it available to other objects, 
 * including the threads under control, so they 
 * can respond uniformly to exception scenarios. 
 * 
 * @return the Supervisor’s own ExceptionHandler 
 */
public ExceptionHandler getHandler() {
    return handler;
}

/**
 * Configures the Supervisor using pre-existing Java classes, 
 * called Builders, to construct the automaton 
 * representations of the plant and specifications, 
 * rather than importing them from IDES. This is to facilitate 
 * Java PathFinder which chokes on file I/O. 
 * 
 * @param plantBuilders 
 *   FiniteAutomatonBuilder objects for FSAs representing underlying 
 *   modules of the Plant under control 
 * @param specBuilders 
 *   AutomatonBuilder objects for arbitrary automata representing 
 *   specifications to be imposed 
 * @param spawnMap 
 *   The mapping of events in the prospective global alphabet 
 *   to the creation of specific threads 
 * @param searchmethod 
 *   The Class object for the Search subclass to be used 
 * @param master 
 *   The index of the master thread in the plantBuilders array 
 * @throws BuilderException 
 * @throws AlphabetException 
 */
public void setModel(Model m) {
model = m;
conditions = new HashMap<String, Condition>();
for (String s: m.getControllable())
    conditions.put(s, suplock.newCondition());

try {
    model.search();
} catch (SearchException e) {
    //model.dump(logger);
    handler.handle(e);
}
logger.log(Logger.LOG_SUPERVISOR, "Initial state: "+
    + model.current().toString(), false);

/*@see java.lang.Object#clone()*/
public Object clone() throws CloneNotSupportedException {
    throw new CloneNotSupportedException();
}

/**
 * Main method exposed to the Threads under control;
 * when called, the Thread locks the Supervisor and
 * attempts to request a state change via a given event
 * uniquely defining its concurrency-relevant point in the code.
 *
 * @param obj Thread object making the request
 * (it just passes <code>this</code>, usually)
 * @param event label for the event being requested
 * /

public void next(Thread obj, String event) {
    Verify.endAtomic();
    Verify.beginAtomic();
    if (SUP_ACTIVE) {
        boolean done = false;

        Condition condition = conditions.get(event);
        Verify.endAtomic();
        suplock.lock();
        Verify.beginAtomic();
        try {
            while (!done) {
                logger.log(Logger.LOG_SUPERVISOR, "Attempting state transition: "+
                    + event + "(" + model.current() + ")": ", true);
try {
    done = model.next(event);
} catch (SearchException e1) {
    model.dump();
    handler.handle(e1);
    System.exit(1);
}

if (done) {
    logger.log(1, "success\(\text{transition}\#\)" + ++trans + ").\text{Growing}\_\text{tree...}", true);
    try {
        Verify.beginAtomic();
        model.search();
        Verify.endAtomic();
    } catch (SearchException e) {
        model.dump();
        handler.handle(e);
        System.exit(1);
    } catch (OutOfMemoryError e) {
        System.out.println(model.getSize() + "nodes.");
        System.out.println(Math.round(Runtime.getRuntime().freeMemory()) + "free\"
                       + Math.round(Runtime.getRuntime().totalMemory()) + "total\"
                       + Math.round(Runtime.getRuntime().maxMemory()) + "max.\";
        throw e;
    }

    for (Entry<String, Condition> s : conditions.entrySet()) {
        if (model.check(s.getKey())) {
            logger.log(1, "UNLOCKING\_\text{\text{\textit{\text{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\textit{\text{\texti
Verifying endAtomic();
suplock.unlock();
Verifying beginAtomic();
}
logger.log(Logger.LOG SUPERVISOR, "Moved to state:
 + model.current().toString(), true);
}

/**
 * Causes thread to sleep for a random time period up to {code duration} seconds.
 */
public static void waitabit(double duration) {
    if (!Supervisor.getSupervisor().SUP_JPF) {
        try {
            Thread.sleep((int)(Math.random()*1000*duration));
        } catch (InterruptedException e) {
            Supervisor.getSupervisor().getHandler().handle(e);
        }
    }
}
Appendix B

Lookahead Algorithm Code

```java
package sup.controller;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintStream;
import java.text.DecimalFormat;
import java.text.NumberFormat;
import java.util.ArrayList;
import sup.controller.des.*;
import sup.importer.AutomatonBuilder;
import sup.importer.BuilderException;
import sup.importer.FiniteAutomatonBuilder;
import sup.importer.IDESImportManager;
import sup.importer.ImportSpawnMap;
import sup.importer.PetriNetBuilder;
import sup.importer.SpawnMap;
import sup.util.Logger;

import sup.controller.des.*;
import sup.importer.AutomatonBuilder;
import sup.importer.BuilderException;
import sup.importer.FiniteAutomatonBuilder;
import sup.importer.IDESImportManager;
import sup.importer.ImportSpawnMap;
import sup.importer.PetriNetBuilder;
import sup.importer.SpawnMap;
import sup.util.Logger;

/*
 * Implements a system model tracker on the basis of limited lookahead search.
 * The Model comprises a tree/graph structure (the latter contingent on an
 * external caching mechanism) of components of its inner class Node.
 * 
 * @author aauer
 */
public class Search implements Model {

    Alphabet alpha;
    private Node node;
    int ctlmax = 3;
    int uctlmax = 2;
    Logger log;

    /**********
    * Implements a system model tracker on the basis of limited lookahead search.
    * The Model comprises a tree/graph structure (the latter contingent on an
    * external caching mechanism) of components of its inner class Node.
    *
    * @author aauer
    *
    **********/
```

110
Logger dump;
Node[] next;

/**
 * Configures the Supervisor using XMD files imported from IDES.
 * @param plantFiles
 * Locations of XMD files for the plant's underlying FSAs
 * @param specFiles
 * Locations of the XMD files for the specs' automata (FSAs/PNs)
 * @param spawnMap
 * The mapping of events in the prospective global alphabet
 * @param searchmethod
 * The Class object for the Search subclass to be used
 * @param master
 * The index of the master thread in the plantBuilders array
 * @throws BuilderException
 * @throws AlphabetException
 * @throws PetriException
 */
public Search(String[] plantFiles, String[] specFiles,
               String spawnMap, int master)
    throws BuilderException, AlphabetException, PetriException {
    IDESImportManager imp;
    AutomatonBuilder[] s = new AutomatonBuilder[specFiles.length];
    FiniteAutomatonBuilder[] p = new FiniteAutomatonBuilder[plantFiles.length];
    for(int i = 0; i < specFiles.length; i++) {
        imp = new IDESImportManager();
        switch(imp.init(specFiles[i])) {
            case IDESImportManager.TYPE_AUTOMATON:
                s[i] = imp.importFSA(specFiles[i]);
                break;
            case IDESImportManager.TYPE_PETRINET:
                s[i] = imp.importPN(specFiles[i]);
                break;
        }
    }
    for(int i = 0; i < plantFiles.length; i++) {
        imp = new IDESImportManager();
        p[i] = imp.importFSA(plantFiles[i]);
    }
    setParameters(p, s, new ImportSpawnMap(spawnMap), master);
}

public Search(FiniteAutomatonBuilder[] plantBuilders,
               AutomatonBuilder[] specBuilders, SpawnMap spawnMap,
int master)
    throws BuilderException, AlphabetException, PetriException {
    setParameters(plantBuilders, specBuilders, spawnMap, master);
}

public void setSearchDepth(int ctrl, int utcl) {
    ctlmax = ctrl;
    uctlmax = utcl;
}

/**
 * Main helper for Model constructor.
 * @param a The global alphabet to be used; Does not need to
 * be complete at the time of Model construction.
 * @param s A {@link sup.controller.des.Specification} of the initial Node.
 * @param p A {@link sup.controller.des.Plant} of the initial Node.
 * @throws PetriException
 */
private void setParameters(FiniteAutomatonBuilder[] plantBuilders, 
        AutomatonBuilder[] specBuilders, 
        SpawnMap spawnMap, int master) throws BuilderException, AlphabetException, PetriException {
    log = Supervisor.getSupervisor().getLogger();
    alpha = new Alphabet();
    ArrayList<FiniteAutomaton> fsas = new ArrayList<FiniteAutomaton>();
    Automaton[] s = new Automaton[specBuilders.length];
    for (int i = 0; i < specBuilders.length; i++) {
        switch(specBuilders[i].getBuilderType()) {
            caseIDESImportManager.TYPE_AUTOMATON:
                s[i] = new FiniteAutomaton(alpha, (FiniteAutomatonBuilder)specBuilders[i]);
                break;
            case IDESImportManager.TYPE_PETRINET:
                s[i] = new ExtendedPetriNet(alpha, (PetriNetBuilder)specBuilders[i]);
                break;
        }
    }
    for (int i = 0; i < plantBuilders.length; i++) {
        fsas.add(new FiniteAutomaton(alpha, plantBuilders[i]));
    }
    PetriNet p = new PetriNet(fsas, spawnMap, master);
    //if(SUP_LOG) p.dump(logger);
}

ArrayList<Specification> tmpspec = new ArrayList<Specification>();
for(Automaton fsa: s)
    tmpspec.add(fsa.getNewSpec());
next = new Node[alpha.getEvents().length];
node = new Node(tmpspec.toArray(new Specification[tmpspec.size()]),
    p.getNewPlant());
}

/**
 * @return {@link sup.controller.des.Alphabet} associated with the model
 */
public Alphabet getAlphabet() {
    return alpha;
}

/**
 * @return current {@link sup.controller.Node}
 */
public String current() {
    return node.toString();
}

/**
 * Attempts to 'fire' an event, making the associated child node current.
 * @param event index of the event to be fired
 * @return <code>true</code> if event was successfully fired.
 * @throws SearchException
 */
public boolean next(String eventstr) throws SearchException {
    int event = alpha.getEventIndex(eventstr);
    if (!next[event].isGenerable())
        throw new SearchException("Non-generable event fired: "+ eventstr);
    if (next[event].isLegal() && next[event].safe) {
        node = next[event];
        return true;
    }
    else {
        if (!alpha.isControllable(event))
            throw new SearchExceptionhException("Uncontrollable event "+
                "improperly blocked by Supervisor: "+
                + eventstr);
        return false;
    }
}

public boolean check(String eventstr) {
    int event = alpha.getEventIndex(eventstr);
    return next[event].isLegal() && next[event].safe;
}
public void dump() {
    dump = new Logger();
    try {
        dump.addPrinter(new PrintStream(new File("logs/dump.txt")));
    } catch (FileNotFoundException e) {
        log.log(Logger.LOG_SUPERVISOR,
                "Could not open file logs/dump.txt...dumping to console", true);
        dump.addPrinter(System.out);
    }
    dump.PRINT_THREAD = false;
    dump.activateSource(Logger.LOGMODEL);
    dump.activate();
    try {
        search();
    } catch (SearchException e) {
        log.log(Logger.LOG_SUPERVISOR,
                "Search exception while trying to dump model", true);
        e.printStackTrace();
    }
}

public void search() throws SearchException {
    log.log(Logger.LOG_SUPERVISOR, "Searching ...", false);
    int[] events = alpha.getSortedEvents();
    int uclength = alpha.getUncontrollable().length;
    boolean[] blocking = new boolean[events.length];
    for (int i = 0; i < events.length; i++) {
        next[events[i]] = node.next(events[i]);
        if (next[events[i]].isGenerable() && next[events[i]].isLegal())
            lookahead(next[events[i]], ctlmax, uctlmax);
        blocking[i] = !next[events[i]].safe;
        if (i < uclength && blocking[i] && dump == null)
            throw new SearchException("Current node not safe!");
    }
    NumberFormat formatter = new DecimalFormat("#####");
    log.log(Logger.LOG_SUPERVISOR, node.subcount + " nodes, 
            formatter.format(Runtime.getRuntime().freeMemory())
            + "free ",
            formatter.format(Runtime.getRuntime().totalMemory())
            + "total ",
            formatter.format(Runtime.getRuntime().maxMemory())
            + "max.", true);
}

public int getSize() {
return node.subcount;
}

private void lookahead(Node n, int ctld, int uctld) throws SearchException {
    int[] events = alpha.getSortedEvents();
    int uclength = alpha.getUncontrollable().length;
    int i; int j;
    Node[] next = new Node[events.length];

    if(dump != null) dump.log(Logger.LOG_MODEL, n.toString(), true);
    if(dump != null) dump.indent();
    for(i = 0; i < events.length; i++) {
        next[i] = n.next(events[i]);
        if(next[i].isGenerable()) {
            if(dump != null)
                dump.log(Logger.LOG_MODEL, alpha.getEvent(events[i]) + "−−>", false);
            if(i >= uclength) {
                if(next[i].isLegal() && ctld > 1)
                    lookahead(next[i], ctld - 1, uctlmax);
            } else {
                if(!next[i].isLegal())
                    if(dump == null)
                        throw new SearchException("Generable uncontrollable event violates "
                        + "specifications."," + n.toString() + "->" + alpha.getEvent(events[i])
                        + "->" + next[i].toString() + ");
            } else
                dump.log(Logger.LOG_MODEL, "*****", false);
        }
    }
    if(dump != null && next[i].subcount == 1)
        dump.log(Logger.LOG_MODEL, next[i].toString() + "<leaf>", true);
}

n.safe = true;
StringBuffer sb = new StringBuffer("[");
boolean b = false;
for(i = 0; i < events.length-uclength; i++) {
    n.subcount += next[i+uclength].subcount;
    n.subEnabled[i] = next[i+uclength].isLegal();
    if(!n.subEnabled[i])
        for(j = 0; !n.subEnabled[i] && j < next.length; j++)
if(i+uclength != j) n.subEnabled[i] &= next[j].subEnabled[i];
n.safe &= n.subEnabled[i];
if(!n.subEnabled[i]) sb.append(alpha.getEvent(events[i+uclength]) + "\n");
}
sb.append(""");
b = !n.safe;
for(i = 0; i < uclength; i++)
n.subcount += next[i].subcount;
for(i = 0; n.safe && i < uclength; i++)
n.safe &= next[i].safe;
if(dump != null) dump.outdent();
if(dump != null) dump.log(Logger.LOG_MODEL, "<−−
+ (n.safe?"<safe>":"<unsafe>") + (b?sb.toString():""), true);

/**
 * Implements the component for the Model class.
 * Each node consists of a Plant and Specification, 
 * both implementation-agnostic.
 * The nodes contain all the information required to 
 * inform control decisions.
 *
 * @author aauer
 */

class Node {
  int subcount;
  boolean[] subEnabled;
  boolean safe;

  // Underlying plant and specifications
  protected Specification[] specs;
  protected Plant plant;

  /**
   * Main constructor for Model.Node.
   * A <code>null</code> {@link sup.controller.des.Plant}
   * indicates a nongenerable Node.
   * A <code>null</code> {@link sup.controller.des.Specification}
   * indicates an illegal Node.
   * The {@link sup.controller.des.Specification}
   * array should never itself be <code>null</code>.
   *
   * @param s Array of Specifications
   * @param p Plant
   * @param model TODO
   */
public Node(Specification[] s, Plant p) {
    subcount = 1;
    specs = s;
    plant = p;
    subEnabled = new boolean[alpha.getControllable().length];
    safe = plant == null;
    subcount = 1;
}

/**
 * Returns the child of the current Node under <b>event</b>; generates the child if this has not already been done.
 *
 * @param event event index
 * @return child node under <b>event</b>
 */
protected Node next(int event) {
    Plant newP = plant.generate(event);
    Specification[] newS = new Specification[specs.length];
    for(int j = 0; j < specs.length; j++) {
        newS[j] = specs[j].check(event, newP);
    }
    return new Node(newS, newP);
}

/**
 * Tests if the current node is generable by the plant in the node’s parent states. Nongenerable nodes have a <code>null</code> plant.
 *
 * @return <code>true</code> if node is generable
 */
public boolean isGenerable() {
    return plant != null;
}

/**
 * Tests whether the current node is {@link #safe}. A node is safe if:
 * <ul>
 * <li>The node:
 * <ul>
 * <li>does not lead uncontrollably to any illegal nodes,
 * <b>AND</b>
 * <li>is not itself illegal; <b>OR</b>
 * <li>The node is not generable in the first place.</ul>
 * </li>
 * </ul>
 *
 * @return Safety value of nodes
 */
public boolean isSafe() {
return safe;
}

/**
 * Tests whether the event sequence leading to a node is legal. An illegal node will have at least one <code>null</code> specification.
 *
 * @return <code>true</code> if node is legal
 */
public boolean isLegal() {
    if (plant == null) return true;
    for (Specification s: specs)
        if (s == null) return false;
    return true;
}

/**
 * Tests whether the node is marked, as dictated by whether the plant is marked.
 *
 * @return <code>true</code> if node is marked
 */
public boolean isMarked() {
    return plant.isMarked();
}

_REPLACE_HERE_NON_JAVADOC_PART

/* (non-Javadoc)
 * @see java.lang.Object#toString()
 */
public String toString() {
    StringBuffer sb = new StringBuffer(plant == null ? "[NULL]" : plant.toString());
    for (Specification spec: specs) {
        sb.append("/");
        sb.append(spec == null ? "[NULL]" : spec.toString());
    }
    //if (expanded) sb.append("*");
    return sb.toString();
}

/* (non-Javadoc)
 * @see java.lang.Object#hashCode()
 */
public int hashCode() {
    int code = plant.hashCode();
    for (int i = 0; i < specs.length; i++) {
        code *= 31;
    }
code += specs[i] == null ? 0 : specs[i].hashCode();
}
return code;
}

/* (non-Javadoc)
 @see java.lang.Object#equals(java.lang.Object)
 */
public boolean equals(Object o) {
    try {
        Node n = (Node) o;
        if (plant == null && n.plant != null) return false;
        if (!plant.equals(n.plant)) return false;
        if (specs.length != n.specs.length)
            for (int i = 0; i < specs.length; i++)
                if (specs[i] == null && n.specs[i] != null) return false;
        if (!specs[i].equals(n.specs[i])) return false;
    }
    return true;
} catch (ClassCastException e) {
    return false;
}
}

@Override
public String[] getControllable() {
    String[] ctrl = new String[alpha.getControllable().length];
    for (int i = 0; i < alpha.getControllable().length; i++)
        ctrl[i] = alpha.getEvent(alpha.getControllable()[i]);
    return ctrl;
}
Appendix C

Automaton Model Code

C.1 Automaton Interface

```java
package sup.controller.des;
import java.util.Arrays;
import sup.controller.Supervisor;

/**
 * Abstraction of an automaton suitable for plant and/or specification
 * tracking through lookahead.
 *
 * @author aauer
 */

public abstract class Automaton {

  protected String[] nodes;
  protected int[] events;
  protected int[] transitions;
  protected Alphabet alpha;
  protected boolean[] marked;
  protected String name;

  /**
   * Main constructor.
   *
   * @param n A name for the automaton
   * @param a The global {@link Alphabet} to be used
   */
  public Automaton(String n, Alphabet a) {
    name = n;
    alpha = a;
  }
}"
```
*/
* Fills in reference arrays for retrieving various subsets of
* the alphabet
*/
protected void fillLocalEvents() {
    transitions = new int[alpha.getEventNames().length];
    Arrays.fill(transitions, -1);
    for (int i = 0; i < events.length; i++)
        transitions[events[i]] = i;
}

/**
 * Gets the local transition index of an event
 * @param e Event index from {@link Alphabet}
 * @return Local index of transition within the automaton
 */
protected int getTransition(int e) {
    if (e >= transitions.length) return -1;
    else return transitions[e];
}

/**
 * Called by subclasses as a standard method for adding
 * events to the associated {@link Alphabet}
 * @param e Array of event labels
 * @param c Controllability information
 */
protected void addEvents(String[] e, boolean[] c) {
    try {
        events = alpha.addEvents(e, c);
        fillLocalEvents();
    } catch (AlphabetException e1) {
        Supervisor.getSupervisor().getHandler().handle(e1);
    }
}

/**
 * Called by subclasses as a standard method for adding nodes information
 * to the automaton.
 * @param s
 * @param m
 */
protected void addNodes(String[] s, boolean[] m) {
    nodes = s;
    marked = m;
}

/**
 * Retrieves an event label based on the transition index local
to the automaton.
 * @param e Automaton's local transition index for the event.
 * @return Event label from global {@link Alphabet}.
 */
protected String getEventName(int e) {
    return alpha.getEvent(events[e]);
}

/**
 * @return Event indices used by this automaton
 */
protected int[] getEvents() {
    return events;
}

/**
 * @return Local transition indices in this automaton
 */
protected int[] getLocalEvents() {
    return transitions;
}

/**
 * @return Count of events used by this automaton
 */
protected int getNumEvents() {
    return events.length;
}

/**
 * @return Count of nodes in this automaton
 */
protected int getNumStates() {
    return nodes.length;
}

/**
 * Retrieves the label on a node by index
 * @param s Index of node
 * @return Node label
 */
protected String getStateName(int s) {
    return nodes[s];
}
/**
 * @return Names of all nodes in the automaton
 */
protected String[] getStates() {
    return nodes;
}

/**
 * Tests whether a given node is marked.
 * @param s Index of node
 * @return {code true} if marked
 */
protected boolean isMarked(int s) {
    return marked[s];
}

/**
 * @return The name of the automaton
 */
public String getName() {
    return name;
}

/**
 * @return The global {link Alphabet} used by the automaton
 */
public Alphabet getAlphabet() {
    return alpha;
}

/**
 * Overridden to generate an appropriate start state for the automaton
 * when used to implement a {link Specification}.
 * @return {link Specification} representing the start state.
 */
public abstract Specification getNewSpec();

C.2 Finite State Automaton

package sup.controller.des;
import sup.importer.FiniteAutomatonBuilder;
import sup.util.Logger;

/**
 * Implementation of a graph consisting of 'states', and labelled arcs
 * (called 'events') connecting them. This is a standard finite
 * state automaton representation.
 */
public class FiniteAutomaton extends Automaton {

    /*
     * Array storage for transition arcs
     */
    public int[][] arcs; // [state][event]

    /*
     * The initial state.
     */
    protected int initial;

    /**
     * Builds a FiniteAutomaton from scratch using a Builder.
     *
     * @param a Associated global {@link Alphabet}
     * @param i Builder with graph information
     */
    public FiniteAutomaton(Alphabet a, FiniteAutomatonBuilder i) {
        super(i.getName(), a);
        addNodes(i.getNodes(), i.getMarked());
        addEvents(i.getEvents(), i.getControllable());
        arcs = i.getArcs();
    }

    /**
     * Gets the state resultant from firing event {code e} in state {code s}.
     *
     * @param s State from which to fire
     * @param e Event to fire
     * @return Next state
     */
    public int next(int s, int e) {
        int t = getTransition(e);
        if (t == -1)
            return -1;
        return arcs[s][t];
    }

    /**
     * Retrieves the initial state of the parent automaton as an int index.
     *
     * @return Initial state
     */
    public int getInitialState() {
        return initial;
    }
}
```java
public Specification getNewSpec() {
    return new Specification();
}

/**
 * Sets the initial state of the automaton.
 * @param i Initial state
 */
public void setInitialState(int i) {
    initial = i;
}

/**
 * Wrapper for an FSA state providing hashing overrides for node caching.
 * @author aauer
 */
public class State {

    int state;

    /**
     * Constructs a new State object with the initial state vector of the
     * parent FSA.
     */
    public State() {
        state = initial;
    }

    /**
     * Constructs a new State object with the supplied initial state.
     */
    public State(int s) {
        state = s;
    }

    public String toString() {
        return name + "(" + alpha.getEvent(state) + ")";
    }

    /**
     * Tests whether the state is marked.
     * @return {@code true} is marked
     */
```
public boolean isMarked() {
    return marked[state];
}

public int hashCode() {
    return 31^state;
}

public boolean equals(Object o) {
    try {
        if(o == null) return false;
        State s = (State)o;
        if(s.state != state) return false;
        return true;
    }
    catch(ClassCastException e) {
        return false;
    }
}

/**
 * Specification object backed by a finite state automaton.
 */
public class Specification extends State
    implements sup.controller.desSpecification {
    /**
     * Constructs a Specification with the initial state of the parent FSA.
     */
    public Specification() {
        state = initial;
    }

    /**
     * Constructs a Specification with the supplied initial state.
     * @param s Initial state
     */
    public Specification(int s) {
        state = s;
    }

    @Override
    public sup.controller.desSpecification[] check(int[] events,
        Plant[] generated) {
        Specification[] s = new Specification[generated.length];
        int nextstate;
for(int i = 0; i < generated.length; i++) {
    if(generated[i] == null)
        s[i] = null;
    else if(transitions[events[i]] == -1)
        s[i] = this;
    else {
        nextstate = next(state, events[i]);
        if(nextstate == -1)
            s[i] = null;
        else
            s[i] = new Specification(nextstate);
    }
}
return s;
}

@Override
public sup.controller.desSpecification check(int event, Plant generated) {
    int nextstate;
    if(generated == null)
        return null;
    if(getTransition(event) == -1)
        return this;
    nextstate = next(state, event);
    if(nextstate == -1)
        return null;
    else
        return new Specification(nextstate);
}

@Override
public String toString() {
    StringBuffer s = new StringBuffer(name);
    s.append("[ ");
    s.append(nodes[state]);
    s.append(" ]");
    s.append(" -");
    s.append(" ]");
    return s.toString();
}

C.3 Petri Net

package sup.controller.des;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.HashMap;
import java.util.Map;
import sup.controller.Supervisor;
import sup.importer.PetriNetBuilder;
import sup.importer.SpawnMap;
import sup.util.Counter;
import sup.util.Logger;

/**
 * Implementation of a bipartite graph consisting of 'places' and 'transitions', with arcs connecting them in a many:many relationship. PetriNet objects can be built from underlying FSAs (a concurrency model) or directly from XMD files for use as specifications (a generalized constraint model).
 * @author aauer
 */

public class PetriNet extends Automaton {

    /**
     * The initial marking.
     */
    protected int[] initial;

    /**
     * For logging and diagnostics – used for pretty-printing the marking vector.
     */
    protected int[] numStatesPerModule = null;

    /**
     * Array storage for place/transition arcs
     */
    protected int[][][] arcs; // [transition][in/out][state][index/count]

    /**
     * Ensures consistency in Alphabets when building a PN from FSAs
     */
    private static Alphabet getAlpha(FiniteAutomaton[] fsas) {
        for (int i = 1; i < fsas.length; i++)
            if (fsas[i - 1].getAlphabet() != fsas[i].getAlphabet())
                return null;
        return fsas[0].getAlphabet();
    }

    /**
     * Builds a Petri net from scratch using a Builder.
     */
public PetriNet(Alphabet a, PetriNetBuilder p) {
    super(p.getName(), a);
    addNodes(p.getPlaces(), p.getMarked());
    addEvents(p.getEvents(), p.getControllable());
    arcs = p.getArcs();
    initial = p INITIALMARKING();
    numStatesPerModule = new int[1];
    numStatesPerModule[0] = nodes.length;
}

/**
 * Builds a Petri net from collection of underlying FSAs.
 * The alphabets of the FSAs must be consistent or
 * an {@link sup.controller.des.AlphabetException} will be thrown.
 *
 * @param fsas Underlying finite state automata
 * @param spawnMap Map of events to thread creation
 * @param master Index of the automaton representing the
 * master thread (the one with a {@code main} method)
 * @throws AlphabetException
 */
 public PetriNet(Collection&lt;FiniteAutomaton> fsas, SpawnMap spawnMap,
                     int master) throws PetriException {
    this(fsas.toArray(new FiniteAutomaton[fsas.size()]), spawnMap, master);
}

/**
 * Builds a Petri net from an array of underlying FSAs.
 * The alphabets of the FSAs must be consistent or
 * an {@link sup.controller.des.AlphabetException} will be thrown.
 *
 * @param fsas Underlying finite state automata
 * @param spawnMap Map of events to thread creation
 * @param master Index of the automaton representing the
 * master thread (the one with a {@code main} method)
 * @throws AlphabetException
 */
 public PetriNet(FiniteAutomaton[] fsas, SpawnMap spawnMap,
                     int master) throws PetriException {
    super("Plant", getAlpha(fsas));
    if(alpha == null) throw new PetriException("Alphabet is null!");
    int initialPlace = -1;
int i; int j; int k; int l;

int numEvents = 0;
int numStates = 0;
numStatesPerModule = new int[fsas.length];
i = 0;
for(FiniteAutomaton fsa : fsas) {
    numStates += fsa.getNumStates();
    numEvents += fsa.getNumEvents();
    numStatesPerModule[i++] = fsa.getNumStates();
}

nodes = new String[numStates];
marked = new boolean[numStates];
events = new int[numEvents];
int[][] initials = new int[fsas.length];
int[][] nodeIndexMap = new int[fsas.length][];
int[][] eventIndexMap = new int[fsas.length][];

numStates = 0;
umEvents = 0;
for(j = 0; j < fsas.length; j++) {
    nodeIndexMap[j] = new int[fsas[j].getStates().length];
    for(i = 0; i < fsas[j].getStates().length; i++) {
        nodes[numStates] = fsas[j].getName() + " ( "
            + fsas[j].getStates()[i] + " ) " ;
        marked[numStates] = fsas[j].isMarked(i);
        nodeIndexMap[j][i] = numStates++;
    }
    initials[j] = nodeIndexMap[j][fsas[j].getInitialState()];
    eventIndexMap[j] = new int[fsas[j].getEvents().length];
    for(i = 0; i < fsas[j].getEvents().length; i++) {
        events[numEvents] = fsas[j].getEvents()[i];
        eventIndexMap[j][i] = numEvents++;
    }
}
fillLocalEvents();
initialPlace = initials[master];

arcs = new int[events.length][3][3][3];

int[] ss = new int[2];
int xref;
ArrayList<.HashMap<Integer,Counter>> arcbin =
    new ArrayList<HashMap<Integer,Counter>>();
for(i = 0; i < 2; i++) arcbin.add(new HashMap<Integer,Counter>());
String evname;
boolean isjpf = Supervisor.getSupervisor().SUP_JPF;
for(l = 0; l < fsas.length; l++) {
    for(j = 0; j < fsas[l].getNumEvents(); j++) {
        for(HashMap<Integer, Counter> a: arcbin)
            a.clear();
        for(i = 0; i < fsas[l].getNumStates(); i++) {
            ss[0] = nodeIndexMap[l][i];
            ss[1] = fsas[l].next(i, fsas[l].getEvents()[j]);
            if(ss[1] >= 0) {
                ss[1] = nodeIndexMap[l][ss[1]];
                for(k = 0; k < ((isjpf && marked[ss[1]])?1:2); k++)
                    if(arcbin.get(k).containsKey(ss[k]))
                        arcbin.get(k).get(ss[k]).incr();
                    else arcbin.get(k).put(ss[k], new Counter(1));
            }
        }
        evname = spawnMap.get(fsas[l].getEventName(j));
        for(xref = 0; xref < fsas.length && !fsas[xref].getName().equalsIgnoreCase(evname); xref++)
            if(xref < fsas.length) {
                xref = initials[xref];
                if(arcbin.get(1).containsKey(xref))
                    arcbin.get(1).get(xref).incr();
                else arcbin.get(1).put(xref, new Counter(1));
            }
        xref = eventIndexMap[1][j];
        for(k = 0; k < 2; k++) {
            arcs[xref][k] = new int[arcbin.get(k).size()][2];
            i = 0;
            for(Map.Entry<Integer, Counter> arc: arcbin.get(k).entrySet()) {
                arcs[xref][k][i][0] = arc.getKey();
                arcs[xref][k][i++][1] = arc.getValue().value();
            }
        }
        arcs[xref][2] = new int[0][0];
    }
}

initial = new int[nodes.length];
for(i = 0; i < initial.length; i++)
    initial[i] = (i == initialPlace ? 1:0);

/**
 * Simulates firing a transition, returning a new marking vector
 * for the Petri net.
 */
```java
/*
* @param m Marking from which to fire
* @param e Event associated with transition to fire
* @return New marking resultant from firing
*/

public int[] fire(int[] m, int e) {
    int i;
    // int[] n = m.clone();
    // WORKAROUND for JPF (can't handle clone() function)
    int[] n = new int[m.length];
    for (i = 0; i < m.length; i++) n[i] = m[i];
    int t = getTransition(e);
    if (t == -1) return null;
    for (i = 0; i < arcs[t][0].length; i++) {
        n[arcs[t][0][i][0]] -= arcs[t][0][i][1];
        if (n[arcs[t][0][i][0]] < 0)
            return null;
    }
    for (i = 0; i < arcs[t][1].length; i++)
        n[arcs[t][1][i][0]] += arcs[t][1][i][1];
    return n;
}

public void dump(Logger lg) {
    lg.log(Logger.LOG_MODEL, "PetriNet", name, true);
    lg.log(Logger.LOG_MODEL, "\(\mathcal{P}\) + nodes.length
        + \(\mathcal{P}\) + nodes.length + \(\mathcal{T}\) + events.length
        + \(\mathcal{T}\) + events.length + \(\mathcal{P}\) + transitions.",
        true);
    lg.log(Logger.LOG_MODEL, "\(\text{Initial marking:}\)",
        true);
    for (int i = 0; i < nodes.length; i++)
        lg.log(Logger.LOG_MODEL, "\(P:\)" + nodes[i]
            + " + String.valueOf(initial[i]) + (marked[i]?
                "(m)" : ""), true);
    lg.log(Logger.LOG_MODEL, "", true);
    lg.log(Logger.LOG_MODEL, "\(\mathcal{P}\) Transitions:",
        true);
    for (int i = 0; i < events.length; i++) {
        lg.log(Logger.LOG_MODEL, "\(\mathcal{T}:\)"
            + alpha.getEvent(events[i])
            + (alpha.isControllable(i)?"(c)" : ""), true);
        lg.log(Logger.LOG_MODEL, "", false);
    }
    for (int j = 0; j < 2; j++) {
        for (int k = 0; k < arcs[i][j].length; k++)
            lg.log(Logger.LOG_MODEL, "\(k=0?\)",
                " + nodes[arcs[i][j][k][0]]
                + (arcs[i][j][k][1] > 1?"x" + arcs[i][j][k][1] + "" : "")",
                false);
        if (j == 0) lg.log(Logger.LOG_MODEL, "\(\mathcal{P}\rightarrow\)",
            false);
        else lg.log(Logger.LOG_MODEL, "\(\mathcal{T}\rightarrow\)", true);
    }
}
```
```java
// Retrieves the initial marking of the Petri net as an array of int.
// @return Initial marking
public int[] getInitialMarking() {
    return initial;
}

// Retrieves the initial marking of the Petri net as
// a {@link PetriNet.Plant}
// for use with a {@link sup.controller.SearchAllerton}.
// @return Initial marking
public Plant getNewPlant() {
    return new Plant();
}

// Retrieves the initial marking of the Petri net as a
// {@link PetriNet.Specification}
// for use with a {@link sup.controller.SearchAllerton}.
// @return Initial marking
public Specification getNewSpec() {
    return new Specification();
}

// Wrapper for a PN marking vector encapsulating functionality about
// the PN system state, and providing hashing overrides
// for node caching.
// @author aauer
//
public class Marking {
    int[] marking;

    // Constructs a new Marking object with the initial marking vector
    // of the parent Petri net.
    public Marking() {
```
marking = initial;
}

/**
 * Constructs a new Marking object with the supplied initial marking vector.
 */
public Marking(int[] m) {
    marking = m;
}

/**
 * Tests whether the current marking is 'marked', as determined by whether all unmarked places are empty.
 * @return {true} if marked
 */
public boolean isMarked() {
    for(int i = 0; i < marked.length; i++)
        if(!marked[i] && marking[i] > 0) return false;
    return true;
}

@Override
public String toString() {
    StringBuffer s = new StringBuffer(name);
    s.append("[");
    int j = 0; int k = 0;
    for(int i: marking) {
        s.append(i);
        if(++k >= numStatesPerModule[j]) {
            s.append("][");
            j++;
            k = 0;
        } else
            s.append("_");
    }
    return s.toString().substring(0, s.length() - 1) + (isMarked()?"(m)":"");
}

@Override
public int hashCode() {
    int code = 0;
    for(int i: marking) {
        code += i;
        code *= 31;
    }
}
return code;
}

@Override
public boolean equals(Object o) {
    try {
        if (o == null) return false;
        Marking m = (Marking) o;
        if (m.marking.length != marking.length) return false;
        for (int i = 0; i < m.marking.length; i++)
            if (m.marking[i] != marking[i]) return false;
        return true;
    }
    catch (ClassCastException e) {
        return false;
    }
}

/**
 * Plant object backed by a Petri net.
 */
protected class Plant extends Marking
    implements sup.controller.des.Plant {

    int nextGenerate;

    /**
     * Constructs a Plant with the initial marking of the parent Petri net.
     */
    public Plant() {
        marking = initial;
        nextGenerate = 0;
    }

    /**
     * Constructs a Plant with the supplied initial marking.
     */
    public Plant(int[] m) {
        marking = m;
        nextGenerate = 0;
    }

    @Override
    public Plant[] generate(int[] events) {
        Plant[] g = new Plant[events.length];
        Arrays.fill(g, null);

        int[] newmarking;
for (int i = 0; i < events.length; i++) {
    newmarking = fire(marking, events[i]);
    if (newmarking != null)
        g[i] = new Plant(newmarking);
}
return g;

@Override
public Plant generate(int event) {
    int[] newmarking;
    newmarking = fire(marking, event);
    if (newmarking == null)
        return null;
    else
        return new Plant(newmarking);
}

/**
 * Specification object backed by a Petri net.
 */
protected class Specification extends Marking
    implements sup.controller.des.Specification {

    /**
     * Constructs a Specification with the initial marking of
     * the parent Petri net.
     */
    public Specification() {
        marking = initial;
    }

    /**
     * Constructs a Specification with the supplied initial marking.
     *
     * @param m Initial marking
     */
    public Specification(int[] m) {
        marking = m;
    }

    @Override
    public Specification[] check(int[] events, sup.controller.des.Plant[] generated) {
        Specification[] s = new Specification[generated.length];
        int[] newmarking;
for (int i = 0; i < generated.length; i++) {
    if (generated[i] == null) {
        s[i] = null;
    } else if (getTransition(events[i]) == -1) {
        s[i] = this;
    } else {
        newmarking = fire(marking, events[i]);
        if (newmarking == null) {
            s[i] = null;
        } else {
            s[i] = new Specification(newmarking);
        }
    }
    return s;
}

@Override
public sup.controller.des.Specification check(int event, sup.controller.des.Plant generated) {
    int[] newmarking;
    if (generated == null) {
        return null;
    } else if (getTransition(event) == -1) {
        return this;
    } else {
        newmarking = fire(marking, event);
        if (newmarking == null) {
            return null;
        } else {
            return new Specification(newmarking);
        }
    }
}
}