Generation of Concurrency Control Code using Discrete-Event Systems Theory

Christopher Dragert
School of Computing
Queen's University
Kingston, Ontario, Canada
dragert@cs.queensu.ca

Juergen Dingel
School of Computing
Queen's University
Kingston, Ontario, Canada
dingel@cs.queensu.ca

Karen Rudie
Dept. of Elec. & Comp. Engineering
Queen's University
Kingston, Ontario, Canada
karen.rudie@queensu.ca

ABSTRACT
The development of controls for the execution of concurrent code is non-trivial. We show how existing discrete-event system (DES) theory can be successfully applied to this problem. From code without concurrency controls and a specification of desired behaviours, concurrency control code is generated. By applying rigorously proven DES theory, we guarantee that the control scheme is nonblocking (and thus free of both deadlock and livelock) and minimally restrictive. Some conflicts between specifications and source can be automatically resolved without introducing new specifications. Moreover, the approach is independent of specific programming or specification languages. Two examples using Java are presented to illustrate the approach. Additional applicable DES results are discussed as future work.

Categories and Subject Descriptors
D.1.3 [**Concurrent Programming**]: *Parallel Programming*. D.2.2 [**Design Tools and Techniques**]: *General.* F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: *Specification Techniques*.

General Terms
Design, Reliability, Verification.

Keywords
Discrete-Event Systems, Model-Driven Development, Automatic Code Generation, Concurrency, Control Theory

1. INTRODUCTION
Concurrency is going mainstream. Leaders in the hard- and software industries and in academia agree that in just a few years even average programmers will have to be able to write concurrent code effectively and efficiently [25,31]. Although concurrent programming has been studied for over four decades, current software development and programming language technology has not yet succeeded in making the design and implementation of correct concurrent code in everyday practice an easy undertaking. At present, our ability to develop concurrent code is still insufficient:

"I conjecture that most multithreaded-general purpose applications are so full of concurrency bugs that—as multicore architectures become commonplace—these bugs will begin to show up as system failures" [22]

Two research directions have been pursued to address this problem:

1) New programming models and programming language abstractions: The goal is to conceive high-level concepts that allow the benefits of concurrency to be reaped while keeping its complexities in check. Seminal work by Dijkstra on semaphores [11], by Hoare on monitors [17], and by Brinch Hansen on languages [7] falls into this category; as does more recent work to incorporate ideas from Hoare's CSP into Java [35] or to extend Java's concurrency library [14].

2) Automatic generation of concurrency control code from specifications: Rather than develop the concurrency control code manually, the programmer instead specifies the desired concurrent behaviour. Concurrent control code is then generated automatically. An example for early work on this idea is based on Habermann's path expressions [2], [8]. Similar work exists with approaches differing in the kind of specification notations supported and the guarantees that the generated code provides. A brief survey will be given in Section 5.

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

This paper targets the automatic generation of control code and thus falls under the second approach. The central distinguishing feature of our work is the way in which the concurrency control code is computed. To this end, we leverage well-established work in the domain of control theory. More precisely, we employ "supervisory control synthesis" which was first proposed by Ramadge and Wonham [29] in order to facilitate the design of discrete-event systems (DES). In short, this process works as follows: From a system P generating events and a specification describing allowed event sequences, a supervisor S is generated such that the composition of P and S exhibits a "minimally restrictive" subset of the allowed event sequences and is nonblocking. We leverage this process by showing how the event-generating system P can be constructed from code that is devoid

SIGSOFT 2008/FSE-16, November 9–15, Atlanta, Georgia, USA

of concurrency controls, but contains user markup indicating events relevant to the specifications. The supervisor generated is then transformed into concurrency control code that enforces the specification. The central features of the resulting approach are as follows:

- **Precision**: Strong, precise, theoretically proven guarantees can be given about the generated code (adherence to specification, deadlock-freedom, and maximal permissiveness).
- **Generality**: Any notation allowing the specification of event orderings can be used; moreover, the approach is programming language independent in the sense that control code for any language offering basic synchronization primitives can be generated.
- **Potential for extension**: Many extensions to the DES supervisory control problem have been developed. It is very likely that many of these will be applicable in future work.

While results from DES control theory have already been used for verification and analysis [38], to the best of our knowledge, this paper is the first to suggest the use of supervisor synthesis for the generation of concurrency control code.

The remainder of the paper is structured as follows: A brief review of the most relevant parts of DES control theory and of supervisor synthesis will be provided in Section 2. Our approach will be described using a running example in Section 3. In Section 4 a more substantial case study is described which also discusses the analysis of the generated Java code using the Java Pathfinder model checker [20]. Section 5 reviews related work. Limitations, future work, and conclusions are given in Section 6. Figures are drawn using IDES [18], and DES operations are performed in IDES and TCT [33].

2. DES Introduction

Discrete-event systems theory, as developed by Ramadge and Wonham [29], is a time-independent language theory based upon discrete event occurrences. Cassandras and LaFortune give an excellent introduction to the field in [9]. Finite-state automata (FSA) are used to model the system, called the *plant G*. Transitions in *G* are events, and the generated language is referred to as the behaviour of *G*.

A model is *nonblocking* if all states are reachable and can reach a terminal state. Nonblocking is an important concept, since it implies both deadlock-freeness and livelock-freeness. Essentially, a nonblocking system is always capable of reaching a final (or marked) state.

Each event is either controllable, meaning it can be disabled, or uncontrollable, meaning it cannot or should not be prevented from occurring. For example, "send message" is typically a controllable event, while "message transmit fail" is an uncontrollable event, since it cannot be prevented. A specification, given as an FSA, gives the desired behaviour of the plant, and is called the *legal language E*. A plant *G* is called *controllable with respect to a specification E* if, for any string *s* from the prefix closure of *E*, there are no uncontrollable events σ that could be generated by *G* at the state reached by *s* such that $s\sigma$ would not be in the prefix closure of E. In other words, if something cannot be prevented, it must be legal, or else the plant is uncontrollable.

In supervisory control, a *supervisor S* (also an FSA) is introduced to control the plant by enabling and disabling events based upon the events that occur in the plant. Figure 1 shows the basic relation between supervisor and plant. The decision to disable and enable events at a given supervisory state is called a *control action*, while the set of all control actions is called the *control policy*.

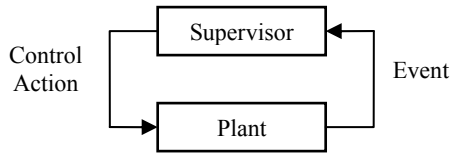

Figure 1. Supervisor-Plant Relation.

The *closed-loop system S*/*G* given by the synchronous product of *S* and *G,* recognizes the largest controllable subset of the legal language. If *E* is controllable, the *closed-loop system* generates exactly *E*; otherwise the largest controllable subset of *E* is generated. Standard DES supervisor synthesis construction is guaranteed to produce an *S* such that *S*/*G* is nonblocking and minimally restrictive [29]. Events not appearing in the supervisor because they are impossible once the supervisor is synchronized with the plant will not appear in the control action.

Modular supervisory control, introduced in [34], allows for multiple specifications and multiple supervisors to act independently on a single plant. However, for the closed-loop system to be nonblocking, the supervisors must be *non-conflicting*, a property which is easily testing using DES tools. Conflicting languages will posses at least one prefix that leads to a marked state in each language, but while that prefix is in the intersection, it does not lead to a marked state in the intersection language and thus leads to blocking. The solution is to find the intersection of the specifications. While the blocking is still present, supervisor synthesis solves the problem this as it finds a nonblocking supervisor to enforce the largest controllable subset of the specifications.

3. PROCESS DESCRIPTION

Given source code without any concurrency control, and a set of informal specifications, we want to generate concurrency control code and weave it back into the original code with as much automation as possible. The connection between abstract DES theory and actual code is made through the identification of relevant events in the code. Our process creates concurrency control code such that the ordering of relevant events falls within the specifications upon execution.

First, we instrument the code, providing events used to create models of the code and construct formal specifications. Standard DES operations are then used to generate a supervisor that satisfies the specifications. The control scheme in the supervisor acts as input for an algorithm that automatically generates concurrency control code. Figure 2 diagrams the process. This approach is designed with the intent to maximize automation. Marking relevant events and formalizing specifications must be done manually, but all other steps can be automated using a variety of methods.

3.1 Process Steps

Our process to create and inject concurrency controls follows:

Input: Source code without concurrency control, and specifications (at any level of formality) describing the desired concurrent behavior of the program.
1. Build the set of relevant events. These are the set of software events that affect the concurrent behavior of the software. (§3.3)
2. For each thread in the software, build an FSA that contains transitions for each relevant event in the software, and introduces necessary structure-preserving irrelevant events. (§3.4)
3. Build the specifications. Specifications must be expressed formally in a format that can be used to automatically build a DES Supervisor. (§3.5)
4. Use the supervisors from step 3 and the FSAs from step 4 to build the closed-loop system. (§3.6)
5. Build concurrency control code that implements the control scheme from the closed-loop system. Insert these into the given source code. (§3.7)
Output: Source code with concurrency controls that enforce the given specifications.

The following sections of the paper describe each step in detail. Once again, we note that the description of our process is programming language independent. An implementation would solve concurrency problems in some specific language (e.g., Java, C++, etc.). We have implemented some stages of the process, and will comment on this in §3.8.

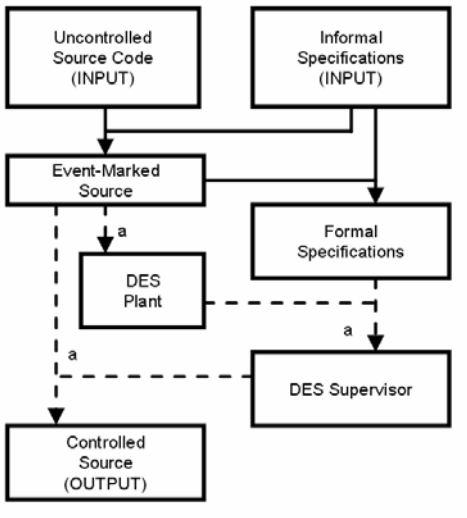

Figure 2. Process to create and inject concurrency controls into source code (Automated steps are marked with an 'a').

3.2 Running Example Introduction

Our running example is a straightforward precedence problem. It comprises five threads governed by four specifications as shown in the precedence graph in Fig. 3. While all threads start together, threads 2, 3, and 4 (*T2*, *T3*, and *T4*, resp.) must wait for thread 1 (*T1*) to finish before executing their code. Additionally, *T4* must wait for thread 5 (*T5*) to finish. Throughout this section, each step in the process is illustrated using this example. Readers interested in a more substantial example are referred to §4.

3.3 Creating the Event Set

To model code using DES, we must isolate events in the code that are both discrete and instantaneous. We define a software event as the instant between the completion of execution of one program statement and the start of execution of the next. This is instantaneous (by definition), and is also discrete, since the completion of the execution acts as a clear demarcation.

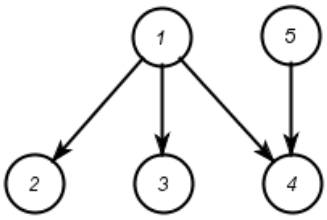

Figure 3. Precedence graph for the running example.

Some events are relevant to concurrent control while most others are not. For instance, the transition leading into the first statement of a critical section of code is *relevant*, while accessing an unshared variable is *irrelevant*. Let E_S be the set of all software events for a given piece of code. We define a set of relevant events, E_R, as the set of events necessary to specify the desired concurrent behaviour. The set of irrelevant events, E_I, is given by $E_I = E_S - E_R$, and is disjoint with E_R.

Relevant events are noted in the code as event markings. These event markings form the basis of the operations to transform the problem into a suitable DES model. In addition, these markings serve as targets for the insertion of concurrency controls. Determining relevant events is part of the inherent complexity of the problem. A developer writing concurrency controls must decide where to place them. Our process is no different in that the user must specify a set of relevant events.

There is a clear mapping from software events as defined to transitions in a control-flow graph (CFG). An event in code has a preceding and successive statement, which directly corresponds to the transition between the same two nodes in the CFG. Figure 4 shows an example with a relevant event marking.

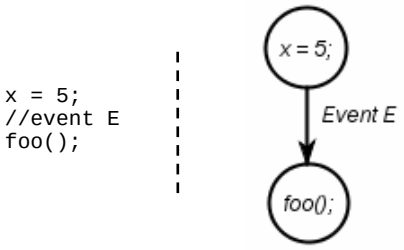

Figure 4. Relation between an event in the code and CFG.

However, the mapping is not always easily inverted in the presence of conditionals. Figure 5 shows code with an if-branch, with event A occurring when the if-branch is not followed. With no else-branch defined, there does not exist a location in the code that corresponds solely to this path. We call this an *implicit path*. An implicit path is a path in the CFG with no location in the code that is unique to that path. A source code transformation can be

employed to render the implicit paths explicit. In this example, an else-branch can be added to make explicit the implicit path.

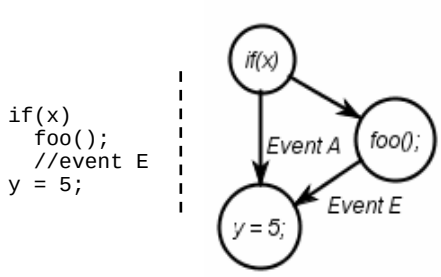

Figure 5. One-way relation between events in the code and CFG.

Each specification implies a set of relevant events, some of which may appear in multiple specifications. A specification speaks about the desired behaviour of the code, which is the effect of interaction from one or more sections of code. Determining a relevant event is done by pinpointing the place in the code where control should be placed. As something that developers already do, it is realistic to expect that developers can perform this task.

Example: The precedence problem presents a set of specifications, from which the relevant events can be extracted. *T2*, *T3*, and *T4* all have conditions placed upon starting. Thus, control needs to be placed at the very beginning of each of their run methods. A relevant event is marked at the first line of code for each of those threads. This is shown for *T3* in Fig. 6. Threads *T2* and *T4* are identical in structure and result. The same specifications tell us that the end of *T1* is relevant and should have an event marking after the last line of code in the run method. Similarly, the end of *T5* is also marked as a relevant event. In total, there are 5 relevant events: *T1-finish*, *T2-start*, *T3-start*, *T4-start*, and *T5-finish*.

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

Figure 6. Code for thread 3 with event marking

3.4 Building the DES Model

The plant, modelled as an FSA, must faithfully represent the events that can be produced by the system being modelled, and it must match the event ordering in the real system. The behaviour of the code need not be duplicated in an abstract model. Instead, the model must merely generate the same relevant events as the code, and in the same ordering.

We accomplish this through the use of CFGs using the link with software events as noted above. A CFG can always be represented as an FSA, since there are a finite number of program locations, and thus a finite number of possible transitions (from each node to each other node, worst case). Polymorphism in object-oriented languages raises no special issue, since all possible transitions can appear in the CFG. We do not perform a static analysis to determine the feasibility of paths at run-time. There is thus the possibility that the system we produce is over-constrained due to non-feasible paths in the CFG. Construction of CFGs can be automated using tools, such as 'Σοφία' [30].

Each resulting FSA can be reduced through a structure-preserving transformation that maintains all relevant event orderings and introduces no new orderings of relevant events. First, method or function calls can be left as unexpanded nodes in situations where no path through the call contains a relevant event. Starting threads can also be left unexpanded, as a different FSA will track the behaviour for the new thread.

The main reduction proceeds by collapsing a branchless chain of (irrelevant event)-(node)-(irrelevant event) into a single irrelevant event, using the same label as the first irrelevant event. A branchless chain of (relevant event)-(node)-(irrelevant event) or a branchless chain of (irrelevant event)-(node)-(relevant event) both become a single relevant event. If no path through a branch contains a relevant event, the branch and all paths may be reduced to a single node. Node labels are not important and do not need to be maintained.

Figure 7 presents Algorithm 1, which builds a reduced FSA version of the event-marked source code.

```
Algorithm 1 transforms relevant event labeled
source code into an FSA that can be used for
DES operations
For each thread, including the main thread:
   1. Build the control-flow graph for the code
   executed by that thread.
   2. Set the entry node as the initial state,
   and all exit nodes as marked states.
   3. Label any edges that are also relevant
   events with the relevant event name.
   4. Discard any CFGs that contain no relevant
   events.
   5. Label all remaining unlabeled edges using
   {i1, i2, i3, ...}. Use the labels in
   increasing order, and do not repeat labels
   across threads.
   6. Apply reductions.
```

Figure 7: An algorithm that transforms source code with event markings into an FSA.

Ideally, an FSA would be reduced such that no irrelevant events remain. However, branching behaviour of the code may leave some irrelevant events that cannot be removed. In Fig. 8, we see a code snippet on the left, the generated FSA model of the CFG in the middle, and the reduced FSA on the right. Only upon following the if-branch does the relevant event occur. When reducing this CFG, we must maintain that branching structure. Thus, *i1* must remain a part of the CFG.

The plant is given by the synchronous product of all the resulting reduced FSAs. This process forces shared events to happen in unison, and allows unshared events to interleave freely. The operation combines the initial states of the threads into one initial state, where all threads begin from their initial states. This is based on an assumption. If a thread is dynamically created by another thread only after a relevant event occurrence, this assumption would be violated, since that shared initial state cannot occur. Dynamic DES theory [15] proposes a DES model wherein event-generating modules can appear and disappear over

time. This is uninvestigated, but seems to offer an alternative to this assumption.

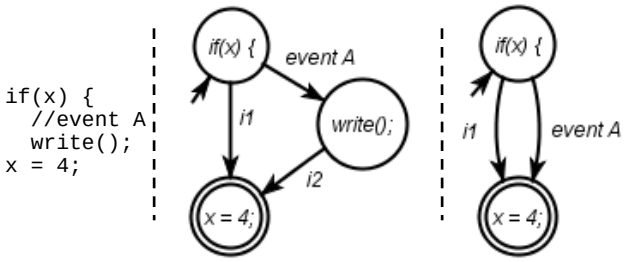

Figure 8. (From l. to r.) A code snippet, an FSA-converted CFG for that snippet, the reduced FSA.

Example: Our running example allows for a very efficient reduction. Since there is no branching behaviour in any of the threads, all irrelevant events may be removed. Each thread is reduced to two nodes and a single relevant event. Fig. 9 shows the results on *T3*. Dynamic thread creation does not occur, so the thread timing assumption holds. The plant in the introductory example contains 32 states and 80 transitions, representing all possible interleavings of the 5 relevant events across the five threads.

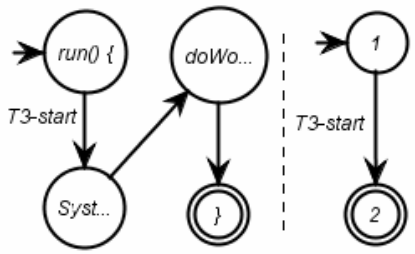

Figure 9. FSA-converted CFG and reduced FSA for *T3*

3.5 Formalizing Specifications

The DES plant made in the previous section generates all possible event sequences of the system. A specification describes the subset of event sequences that are to be allowed. As noted in the DES section, modular DES theory allows for the usage of multiple specifications. Only behaviours permitted by all specifications will be allowed in the controlled plant. Specifications must be each given as an FSA since we are using FSA-based DES theory. This does limit the types of specifications to those that can be expressed in a regular language, thus allowing safety properties, but precluding liveness properties. We choose to build FSAs directly from the informal specifications, instead of using an intermediary formalization.

Since allowed behaviours must be present in all specifications, it is vital that a specification only restricts the behaviours intended, while permitting all others. In other words, all events possible in the plant should be permitted in the specification except where an event must be explicitly restricted by the specification. In an FSA, this is accomplished through the addition of a self-loop at all states for every event in the plant that is not a relevant event arising from that specification. This includes any irreducible irrelevant events.

Example: Our running example provides two simple safety properties to enforce. Threads that must wait on *T1* to finish are restricted by the specification in Fig. 10, and the thread waiting on *T5* is restricted by the specification in Fig. 11.

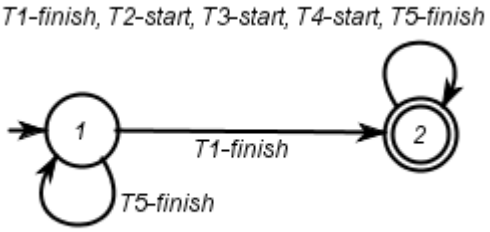

Figure 10. *T2*, *T3*, and *T4* must wait for *T1* to finish.

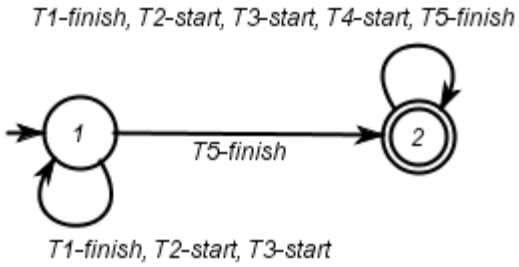

Figure 11. *T4* must wait for *T5* to finish.

3.6 Constructing the Supervisor

With the plant and a set of modular specifications we can proceed to synthesize a supervisor. The supervisor construction algorithm in [29] performs this task in polynomial time for a single specification. Thus, a choice must be made. If supervisors are created for each specification, the possibility exists for the resulting closed-loop system to have a conflict and therefore a possible deadlock. This can be avoided by combining the specifications using the synchronous product operation. This single specification is called the *monolithic specification*. However, the resulting monolithic supervisor may exhibit an exponential increase in the number of states, which would inhibit scalability.

In general, only the specifications that contribute to the conflict should be combined, thus achieving a nonblocking solution while mitigating state-space explosion. Multiple supervisors could be implemented through repeated use of the code generation method in section 3.7.

In our implementation, we simply build the monolithic specification and proceed with one supervisor. The result is a nonblocking supervisor guaranteed to enforce the behaviours allowed by the monolithic specifications. If the specification is uncontrollable with respect to the plant, then a supervisor is generated that enforces the largest controllable subset of behaviours. Several software packages exist that can perform these operations, including TCT [33] and IDES [18].

Example: The monolithic specification was produced in IDES from the specifications in Figs. 10 and 11. The resulting specification contains 4 states and 13 transitions. This, along with

the plant from section 3.4, was exported to TCT, where the supervisor construction operation was performed. The resulting supervisor has 14 states and 23 transitions.

3.7 Code Generation

Using the supervisor as input, along with the original source code, we generate concurrency control code such that, when woven into the source code, the resulting program behaviour is a subset of the specifications. In fact, the generated code is primarily a realization of the abstract supervisor. We now analyze several properties of supervisory control, eventually leading to an algorithm to generate and insert code.

Supervisory control requires the ability to allow and prevent the occurrence of an event. A dedicated semaphore can be used to provide this capability. For a semaphore to prevent an event occurrence, the call to wait on that semaphore must occur at the event location in the code. Uncontrollable events do not require a semaphore, as they are never disabled. We thus use semaphores to indicate the current state of a controllable event. If the event is enabled, the associated semaphore has one or more permits in it. The semaphore for a disabled event has no permits. Most modern concurrent programming languages provide semaphores in some form, which allows this solution to stay general.

In any supervisor state, some events are enabled and some are disabled. Only upon an event occurrence is there the potential for change in the current control action. However, event occurrences do not uniquely determine the supervisor state. Changes to the control action are based on the current state of the supervisor. An implemented supervisor must therefore be able to track its internal state during execution. To maintain data integrity, accesses to state tracking information must be mutually exclusive. Updating the supervisory control action must also be done in mutual exclusion. Additionally, if the state tracking and stage changing is not done atomically, then a thread could interleave a second state change with the first, causing the supervisor to become corrupted.

While the control policy tells us what events must be disabled in each state, it does not include information on transitions. We must explicitly build a construct, which we will call the *change map*, to track changes in the control action based upon transitions. For each transition in the supervisor, we compare the set of disabled events and enabled events at the source s and target t of that transition. The change map gives us the pair (Δ_E, Δ_D) for each transition, where Δ_E is the set of events that are disabled at s and become enabled at t, and Δ_D is the set of events that are enabled at s and become disabled at t. When a transition occurs, we can reference the change map to correctly update the current control action.

Algorithm 2 in Fig. 12 generates and weaves concurrency control code into the original source using the supervisor. First, we generate the change map from the control map and the supervisor. Next, we create semaphores in shared memory that are initialized to the control action in the initial state of the supervisor. On a controllable event occurrence, we check a `stateChangeTest` method to determine if the event is enabled. If it is, the supervisor state is updated along with the control action. An uncontrollable event always proceeds to the supervisor state change method.

Algorithm 2 is easily proven correct in general. Consider the initial program state s. The control action at this state is read directly from the supervisor, and since the supervisor is correct-by-design [29], the control action must be correct. A subsequent transition to a child program state s' causes an update to the control action from the change map, which is built directly from the control policy from the correct-by-design supervisor. Thus, s' also enforces the correct control action. This argument generalizes to all states, showing that updating control actions from the change map is a viable approach for supervisor implementation.

```
Algorithm 2 generates concurrency control code,
then inserts it into the marked source code.
  1. Build the change map from the control map
  and the supervisor.
  2. Create a semaphore for each controllable
  event, initialized to 0 if the event is
  disabled in the initial state of the
  supervisor, and 1 if it is enabled.
  3. Create a stateChangeTest method/function,
  accessed in mutual exclusion that determines
  if an event is enabled.
        1. If yes, update the state and control
        action from the change map. If an event
        becomes enabled, signal the associated
        semaphore. If it becomes disabled,
        remove all permits.
        2. If no, exit.
  4. At each controllable event occurrence,
  consult the stateChangeTest. If the event is
  disabled, wait on the associated semaphore.
  If enabled, the stateChangeTest will change
  the supervisor state.  The thread proceeds.
  5. At each uncontrollable event, insert code
  to access the stateChangeTest. This will
  update the supervisory state, then return true
  as an uncontrollable event never blocks.
```

Figure 12. Concurrency code generation algorithm.

Errors in this algorithm could arise in the implementation, however. Is the change map correctly constructed and consulted? One must be careful to read the current state, determine if an event is allowed, and then update the control action without introducing the possibility for deadlock due to the consulting process. Also, it is important to note that supervisors do not cause events to occur—they only allow them. Thus, when a waiting thread is signaled, it is possible for another thread to execute an event and "re-disable" the waiting thread before it acts. Thus, the first action of the awoken thread must be to check if the event it was waiting on is still enabled.

3.7.1 Java Implementation/Example

We have implemented Algorithm 2 in Java. The change map is constructed from the supervisor as given by IDES [18]. A static `Synchronizer` class is introduced to act as a shared memory location. It initializes all semaphores, and contains the public `stateChangeTest` method along with the private `changeSupervisorState` method.

At each controllable event, a block of code is inserted that differs only by the relevant event name. Relating this to our running example, Fig. 13 shows the code inserted at the *T3-start* event. At uncontrollable events, there is a call to `stateChangeTest` to notify the supervisor of that event occurrence. Figure 14 shows the `stateChangeTest` method, which never changes, and Figure 15 shows an excerpt of the `changeSupervisorState` from our running example.

```
//relevant event: T3start
while (true) {
  if(Synchronizer.stateChangeTest("T3start",
     Synchronizer.T3start))
    break;
  Synchronizer.T3start.acquireUninterruptibly();
  Synchronizer.T3start.release();
}
```

Figure 13. Inserted code at T3-start.

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

Figure 14. `stateChangeTest` in `Synchronizer`.

```
private static void changeSupervisorState
                          (String event){
  if (event.equals("T1finish")) {
    switch(Synchronizer.stateTracker) {
      case(0):
        Synchronizer.T3start.release();
        Synchronizer.T2start.release();
        Synchronizer.stateTracker = 1;
        break;
      case(2):
        [...]
    }
  }
  else if (event.equals("T3start")) {
    switch(Synchronizer.stateTracker) {
      case(1):
        Synchronizer.stateTracker = 4;
        break;
      case(3):
        [...]
}
```

Figure 15. `changeSupervisorState` in `Synchronizer`.

Once a controllable event calls the `stateChangeTest` method, the test checks the current status of the semaphore associated with that event. If available, the program proceeds into changing the supervisor state through a call to `changeSupervisorState`. If not available, the state change fails, and the thread is forced to wait on the event semaphore until another state change releases that semaphore. Upon releasing, the process is repeated until the state change eventually takes place. This repeating ensures that the plant and supervisor maintain synchronization as noted above. The `changeSupervisorState` method encodes the change map, and, on a state change, uses it to update control actions and the current supervisor state. Two sequence diagrams in Figs. 16 and 17 show how the implemented process is carried out.

3.8 Verification of Generated Code

Given that DES theory is rigorously proven to generate correct, nonblocking supervisors, it is implied that the produced control policy satisfies the specifications and is nonblocking. However, the DES work is only as reliable as the input, and the concurrency control code is only reliable to the extent that it correctly implements the generated control policy. Ideally, the algorithms to create the DES model and synthesize code would both be proven correct with a formal correctness proof, but this not yet been done. In the short term, we provide assurance that the generated code is in fact correct through the use of a model-checker.

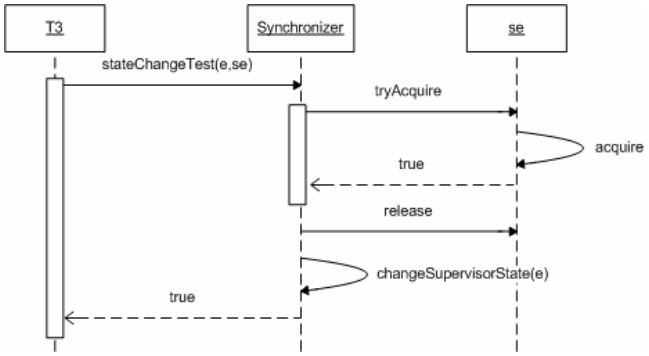

Figure 16. A successful state change for an enabled event.

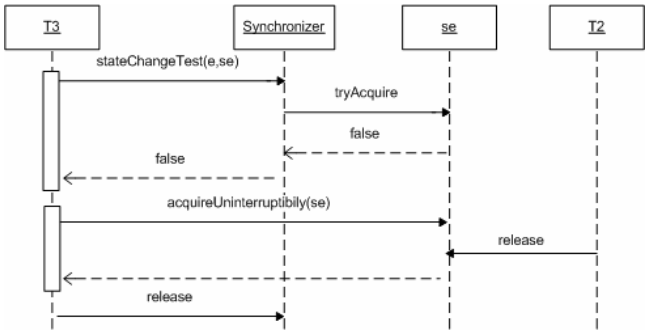

Figure 17. An unsuccessful state change for a disabled event.

The complete code, including generated concurrency controls, was verified using Java Pathfinder [20] (JPF). First, the code was instrumented using flags and assertions. Referring again to our running example, a shared flag defaulting to false was created to mark the completion of *T1*. The first statement after the inserted control code in *T2*, *T3*, and *T4* was an assertion that this flag was set to true. JPF did not discover any interleaving in which this assertion was false, thus proving that the specification was met. This was done for all specifications. In addition, JPF found the code to be free of deadlocks. Settings for JPF differed from default only by setting `search.match_depth` to true and adding `gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty` to the search properties.

4. EXTENDED EXAMPLE

4.1 Introduction

The transfer-line problem was originally outlined as a DES problem by Wonham and Ramadge in [34], and is a well-studied problem in DES literature. We cast the problem as a Java program that requires concurrency control. A depiction of how parts flow through the system is given in Fig. 18. Machine 1 (*M1*) creates parts and places them in Buffer 1 (*B1*). Machine 2 (*M2*) removes parts from *B1*, works on them, and then places them in Buffer 2

(*B2*). The test unit (*TU*) removes parts from *B2* and then tests them. If the part passes the test, it is removed from the system. If the part fails, it is uncontrollably placed in *B1*[1]. The active components *M1*, *M2*, and *TU* only work on one part at a time. Specifications are quite simple. Buffer 1 has a capacity of three, *B2* has a capacity of one, and neither buffer may overflow or underflow.

If the specifications are enforced as given, the system could deadlock. Consider path p = 11123121. Both buffers are full and *M2* is working on a part. Neither machine can proceed, as a buffer overflow could result. The test unit is prevented from taking part a part from B2, as it could uncontrollably reject the part and then overflow *B1*. The system is thus deadlocked despite all specifications being met. DES theory allows us to overcome this issue by generating a nonblocking supervisor that enforces the largest subset of controllable behaviours.

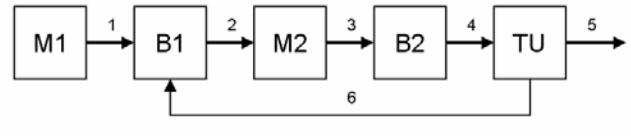

Figure 18. Flow of parts in the transfer line example.

Our Java implementation creates a thread for each of the three active components *M1*, *M2*, and *TU*. A `Part` class was defined to allow for the creation of objects to be passed through the system. Buffers are passive entities, and are used as shared resources by the active components. The main class instantiates the buffers, instantiates and starts the active components, then terminates.

Both buffers are accessed by more than one thread. To ensure data integrity of the shared resource, we include two new specifications. *B1* must be accessed in mutual exclusion, and *B2* must also be accessed in mutual exclusion. Also, each buffer access is coupled with console output describing the action taken. To ensure output matches the program state, the output statement is considered part of the buffer access and is synchronized accordingly. Finally, we assume that when an active component starts accessing a buffer (either to read or write), it finishes uncontrollably.

4.2 Defining Relevant Events

Each specification implies a set of relevant events as follows. In practice, each event is marked in the code as it is refined from the specification. In naming our relevant events, we adopt the practice of appending the label of an uncontrollable relevant event with a '*-u*'.

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

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

***B1* and *B2* must each be accessed in mutual exclusion**: Buffer access start events are already introduced. Events are needed to note the end of a buffer access. As noted in our description, all finish events are uncontrollable. The added relevant events are *M1addB1-finish-u*, *TUaddB1-finish-u*, *M2getB1-finish-u*, *M2addB2-finish-u*, and *TUgetB2-finish-u*.

The code for *M1* appears in Fig. 19, and the code for TU appears in Fig. 20, both with event annotations. The `doWork()` method in the active threads causes a brief wait to simulate work being done on a part.

```
//M1 run method
public void run() {
  while (true) {
    //wait for some random period
    doWork();
    //create a new Part
    Part newPart = new Part();
    //put it in the target buffer
    //relevant event: M1addB1-start
    System.out.println("Machine1 tries to put
                        a part in " + target);
    target.addPart(newPart);
    //relevant event: M1addB1-finish-u
  } // loop forever
}
```

Figure 19. *M1* Code with relevant events italicized.

```
//TU run method
public void run() {
  while (true) {
    //get a part from the source
    //relevant event: M2getB1-start
    System.out.println("TestUnit tries to get a
                        part from " + source);
    Part currentPart = source.removePart();
    //relevant event: M2getB1-finish-u
    //test that part for some period of time
    doWork();
    if (!testPart()) {
      //no good!  send back to rejectBin
      //relevant event: TUaddB1-start
      System.out.println("TestUnit tries to put
                  a part in " + rejectBuffer);
      rejectBuffer.addPart(currentPart);
      //relevant event: TUaddB1-finish-u
    }
    else {
      //else part is good, remove from system
      //take no action
    }
  } //loop forever
}
```

Figure 20. TU Code with relevant events italicized.

[1] This diverges slightly from the standard description, as usually events 1 and 3 are uncontrollable, and there is a controllable event to start the creation of a part in M1.

4.3 Code to DES

Using Algorithm 1 (Fig. 7), we build the DES plant for the system. In Figs. 21 and 22, we see the FSA-transformed CFG coupled with the reduced CFG for *M1* and *TU*.

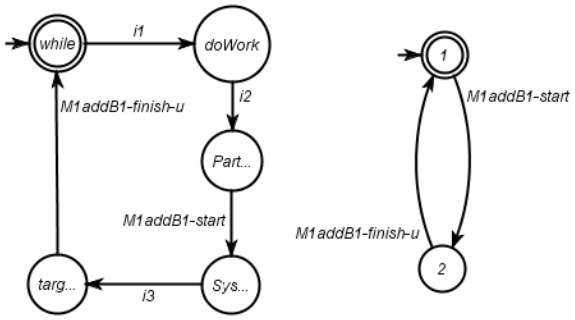

Figure 21. *M1* FSA-converted CFG and reduced FSA.

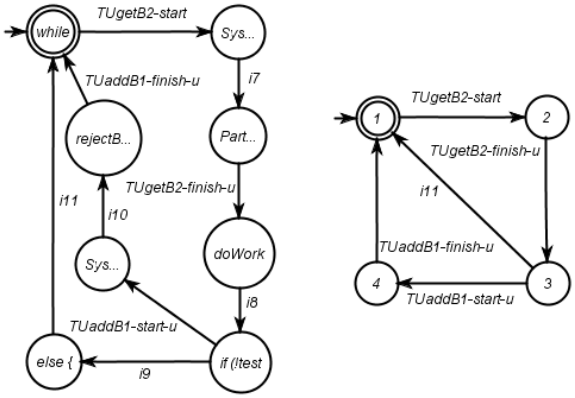

Figure 22. *TU* FSA-converted CFG and reduced FSA.

Figure 21 exhibits an irrelevant event *i11* that cannot be removed. Although it does appear in any specification, an occurrence of *i11* provides information that is important to the system. It means that events *TUaddB1-start-u* and *TUaddB1-finish-u* will not occur, and implies that a part has been removed from the system, duplicating the behaviour of event 5 from Fig. 18. In other words, an occurrence of this irrelevant event allows us to observe the non-occurrence of relevant events, which is vital information when controlling event orderings.

The plant is given by the synchronous product of each of the reduced FSAs. The resulting plant has 32 states and 104 transitions, representing all possible interleavings of the three threads being considered. A visual representation is not helpful due to the large state-space.

The next step is formalizing the specifications. Any event from the plant (including irrelevant events) that does not appear in a specification is placed in self-loop. The first specification to formalize is that *B1* may not overflow or underflow. Recall that *B1* has a capacity of three, that two events add parts (*M1addB1-start*, *TuaddB1-start-u*), and one event removes parts (*M2getB1-start*). Prevention of overflow is ensured by disallowing adding when three buffer-add actions have occurred without a corresponding remove event. To prevent underflow, a remove event is only permitted when an unmatched add event has occurred. The formal specification is given in Fig. 23, with unlabelled self-loops.

Next, the mutual exclusion specification for *B1* is formalized. There are six events that access *B1*: *TUaddB1-start*, *TUaddB1-finish*, *M1addB1-start*, *M1addB1-finish*, *M2getB1-start*, and *M2getB1-finish*. Whenever a start event begins an access to the buffer, the associated finish event must occur before any start event can occur. The FSA is given in Fig. 24. Specifications for *B2* are handled in the same manner as *B1* and are not shown here.

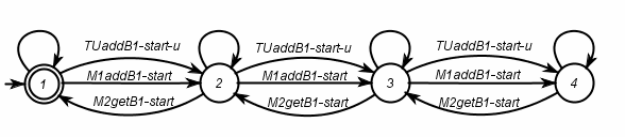

Figure 23. *B1* overflow/underflow specification.

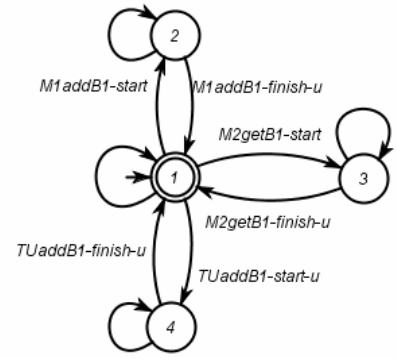

Figure 24. *B1* mutual exclusion specification.

As noted in the problem introduction, the specifications are conflicting. Thus, creating four separate modular supervisors will lead to a blocking supervisor. To avoid this, we combine the specifications into a monolithic specification using IDES. In this case, the monolithic supervisor is comprised of 52 states with 176 transitions. The event *i11* accounts for 52 of those transitions. As an irrelevant event, no specification restricts this event (except in self-loop) and therefore it appears as a self-loop at every state.

4.4 Creating the Supervisor and Code

With the plant and the monolithic supervisor in hand, a supervisor may now be synthesized. Using TCT, a nonblocking supervisor was generated. The supervisor itself contains 69 states and 114 transitions. Recall that deadlock could be reached when all buffers are simultaneously filled. In addition to enforcing the specifications as given, the control behaviour also resolves the conflict noted in the problem description. Essentially, this is done by reserving room in *B1* for *TU* to uncontrollably return a part back without overflow. The resulting behavior is a proper subset of the specified behavior. Without introducing a new specification, the conflict has been resolved and deadlock freedom has been achieved. Furthermore, the user never had to be aware of the deadlock potential, nor act to prevent it.

Using the Java implementation of Algorithm 2 (Fig. 12), code was generated to effect supervisory control. The generated code is of similar form to the code presented in §3.7, differing in event names, change map details, and the size of the change map.

Once again, the resulting code was instrumented with assertions and model-checked with Java Pathfinder. It was found to be deadlock free in all states and never in violation of the specifications.

5. RELATED WORK

Early work is based on Campbell and Habermann's "path expressions" [8], which allow the specification of the allowed sequences of operations on an object of an abstract data type. In [2], Andler extends this idea by proposing "predicate path expressions" (PPEs) and also describes an implementation scheme which is based on the translation of PPEs to finite automata. Every invocation of an operation of the data type is bracketed by a prologue and an epilogue which consults the automaton and ensures only conforming invocations can proceed. The potential for deadlock in the generated code is not discussed, but the use of formal methods is suggested.

Emerson and Clarke in [13] generate program abstractions called "synchronization skeletons", which suppress detail irrelevant to synchronization. Specifications are given in Computational Tree Logic (CTL), but can only create a synchronization skeleton if the CTL specifications are satisfiable. The program is modelled as a 'finite model', similar to an FSA. A scheme is developed that details when each thread is allowed to enter a critical section without violating the specifications. A similar method was proposed by Manna and Wolper in [24] using Linear Temporal Logic (LTL) rather than CTL. Our work mirrors this general approach by developing concurrency code separately through the use of specifications. However, the substitution of DES solves several issues, such as unsatisfiable or conflicting specifications, and provides guarantees on the quality of the solution.

The general approach from [13] has been adopted several times, differing by the usage of alternate specification formalization or a different modelling technique. In [6], specifications are given using process algebra, whereas [36] employs Bultan's Action Language. Tools allow the formal analysis of the specifications. An approach based on global invariants is discussed in [10]. The invariants specify allowed behaviour and use specific counter variables that keep track of how many processes are currently executing user-specified regions. Patterns of some common specifications are given. Support for model checking the generated code is provided to detect synthesis errors.

Alt, Sander and Wilhelm present an approach for the generation of synchronization code for parallel compilers [3]. The compiler is modularized into engines. A global dependence graph is computed from specifications of the input-output behaviour of these engines. From this graph, code controlling the invocation of engines and access to shared data structures is generated. Deadlock avoidance is guaranteed.

Matos et al. describe a technique for the automatic generation of synchronization conditions based on finite-state machine descriptions of both the components and the specifications [26]. A synchronous communication model such as Esterel, Lustre, or SMV is assumed. The resulting code needs to be checked for deadlocks, because "circular dependence between synchronized components" will cause deadlocks. The tool implementing the approach supports this by outputting an SMV representation of the system.

In [5], Autili et al. describe a tool called SYNTHESIS that produces correct and deadlock-free distributed component systems. Several examples are cited. They introduce a concept of 'last chance' states, which are the last chance to prevent a deadlock by preventing some transition. This echoes early DES work by Ramadge and Wonham, wherein last chance states correspond to the last states where control could be applied. The mirroring here highlights the relevance of introducing DES theory to concurrency control problems, as it is made clear by this paper that success can be found using DES-like techniques.

Aspect-oriented programming, introduced by Kiczales et al. in [21], is based on identification of cross-cutting concerns, coding each concern individually, and then weaving it back into the main code base. Recent work [12] deals explicitly with concurrent aspects. Advice is represented as a finite-state graph, and aspects are modelled to act in parallel. The authors note, "because of the inherent difficulty of developing correct concurrent programs, [...] a model for concurrent aspects should support the use of automatic verification techniques, such as model checking…" Once again, the value of DES is apparent, as it can not only produce control policies, but can provide formally proven guarantees on correctness and nonblocking.

Detailed information about the guarantees provided by the code generated by each of the approaches listed above was difficult to obtain. For instance, it is not always clear to what extent the generated code is nonblocking or minimally restrictive. In contrast, our approach is very clear on this point since it is directly based on a large body of research in DES control theory. A more thorough account of how the approaches differ with respect to usability, expressiveness, and guarantees is necessary and left as a topic for future work.

Also unique to this approach is the ability for DES theory to enforce conflicting or impossible specifications by instead enforcing the largest controllable sublanguage. For instance, the Dining Philosophers problem can be automatically solved without introducing special specifications to resolve the deadlock. Our extended example in section 4 demonstrated this through the use of conflicting modular specifications.

5.1 Applying DES to Concurrency

A theory of controlling discrete-event systems was detailed by Ramadge and Wonham in [29]. The authors indicate that DES theory is applicable to computing, but the idea is left unexplored. To the best of our knowledge, the application of DES to software development is left unaddressed in the DES community.

There has been work in the opposite direction, where results from model-checking have been applied to DES results. The supervisor control problem was revisited by Ziller and Schneider in [37,38], where the authors generalize the supervisor synthesis algorithm to allow for specifications given in the μ-calculus. This allows the consideration of both safety and liveness properties, including fairness properties.

Some early attempts have been made by control theorists to address the concurrency control problem. Thistle, in [32], adapted the verification framework from Manna and Pnueli [23] to solve the control problem for a small set of abstract concurrent processes. It used LTL and modelled both the system and specifications as a set of logic statements. This approach allowed

safety properties and some liveness properties, but was largely a manual process

6. CONCLUSIONS

This paper demonstrated a process to apply existing DES theory to the automatic generation of concurrency control code. The generated control scheme is guaranteed to be nonblocking and minimally restrictive while enforcing the largest controllable subset of the desired behaviour. An algorithm was provided to transform source code into FSA models through event marking. A second algorithm takes a DES supervisor and transforms it into concurrency control code.

The primary result is the formation of a link between concurrent software development and DES theory. This introduces very exciting avenues of future research: For instance, the DES theory we used has already been extended in a large variety of ways (e.g., support for liveness properties through CTL [4], CTL* [19], LTL [32], the μ-calculus [37], and real-time systems [28]). Parameterized DES [27] suggests a methodology to abstract multiple instances of the same thread. Hierarchical DES as in [39] could be applied for large programs with multiple libraries. Chances are good that at some of these results can be leveraged to extend our work.

The decision to use the FSA-based version of DES limits our process to safety properties only. Other more expressive DES theories exist, such as those based on Petri-nets [16], CTL* specifications [19], or μ-calculus [37]. The last two can express liveness properties. While recasting the presented process using a different model of DES is a non-trivial task, it is our belief that this work shows that DES is effective when applied to concurrency control problems. Thus, it is worth undertaking the technical work required to recast our process using a more expressive theory. Primarily, such a recasting would involve transforming code into the applicable model, rather than an FSA.

The primary limitations of this work involve the generality of code in an implementation. Dynamic DES needs to be introduced to properly treat dynamic thread generation. Static verification could eliminate unfeasible paths arising in FSA construction due to run-time decisions such as polymorphism. Moreover, this work does not consider efficiency; future work must also address this.

7. REFERENCES

[1] L. De Alfaro, T. Henzinger, and R. Majumdar, "From Verification to Control: Dynamic Programs for Omega-Regular Objectives" in *Proc. of the 16th Annual Symp. on Logic in Computer Science* (LICS 2001), IEEE Computer Society Press, pp. 279-290, Washington, D.C., 2001.

[2] S. Andler, "Predicate Path Expressions" in *Proc. of 6th ACM Symp. on Principles of Programming Languages* (POPL 1979), pp. 216-236, Jan. 1979.

[3] M. Alt, G. Sander and R. Wilhelm. "Generation of Synchronization Code for Parallel Compilers", in *Proc. of the 5th Intl. Symp. on Programming Language Implementation and Logic Programming*, LNCS, Vol. 714, pp. 420-421, 1993.

[4] A. Arnold, A. Vincent, and I. Walukiewicz, "Games for Synthesis of Controllers with Partial Observation", in *Theor. Comp. Sci*, Vol. 303, No. 1, pp. 7-34, Jun. 2003.

[5] M. Autili, P. Inverardi, A. Navarra, and Massimo Tivoli, "SYNTHESIS: A Tool for Automatically Assembling Correct and Distributed Component-Based Systems", in *ICSE '07: Proc. of the 29th Intl. Conf. on Software Engineering*, pp. 784-787, 2007

[6] E. Bontà, M. Bernardo, J. Magee, and J. Kramer, "Synthesizing Concurrency Control Components from Process Algebraic Specifications", in *Proc. of 8th Intl. Conf. on Coordination Models and Languages*, LNCS 4038, Bologna, Italy, Jun. 14-16, 2006.

[7] P. Brinch Hansen, "The Programming Language Concurrent Pascal", in *IEEE Trans. on Software Engineering*, 1(2). pp. 199-207, Jun. 1975.

[8] R.H. Campbell and A.N. Habermann, "The Specification of Process Synchronization by Path Expressions", in *LNCS*, Vol. 16. pp. 89-102, 1974.

[9] C. Cassandras and S Lafortune, "Introduction to Discrete-Event Systems". Kluwer, Boston, MA, 1999.

[10] X. Deng, M.B. Dwyer, J. Hatcliff and M. Mizuno. "Invariant-Based Specification, Synthesis, and Verification of Synchronization in Concurrent Programs" in *Proc. of the 24th Intl. Conf. on Software Engineering* (ICSE '02), pp. 442-452, Orlando, FL. 2002.

[11] E.W. Dijkstra. "Cooperating Sequential Processes" in *Programming Languages*, Academic Press. New York. pp. 43-112. 1965.

[12] R. Douence, D. Le Botlan, and J. Noyé, and M. Südholt, "Concurrent Aspects", in *Proc. of the 5th Intl. Conf. on Generative Programming and Component Engineering*, pp. 79-88, 2006.

[13] E.A. Emerson and E. M. Clarke, "Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons" in *Science of Computer Programming*, Vol. 3, No. 1, pp. 241-266, 1982.

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

[15] L. Grigorov and K. Rudie, "Near-Optimal Online Control of Dynamic Discrete Event Systems", in *Journal of Discrete Event Dynamic Systems: Theory and Applications*, Vol. 16, pp. 419-449, 2006.

[16] L.E. Holloway, B.H. Krogh and A. Giua, "A Survey of Petri Net Methods for Controlled Discrete Event Systems", in *Discrete Event Dynamic Systems*, Vol, 2, No. 7, pp. 151-190, April 1997.

[17] C. Hoare. "Monitors: An Operating System Structuring Concept", in *Communications of the ACM*. Vol. 17, No. 10, pp. 549-557. 1974.

[18] IDES: The Integrated Discrete-Events Systems Tool, Discrete-Event Control Systems Lab, Queen's University, http://www.ece.queensu.ca/hpages/labs/discrete/software.html, Mar. 2008.

[19] S. Jiang and R. Kumar, "Supervisory Control of Discrete Event Systems with CTL* Temporal Logic Specifications", in *Proc. of the 40th IEEE Conf. on Decision and Control*, Orlando, FL. 2001.

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

[21] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin "Aspect-Oriented Programming", in *Proc. of the European Conf. on Object-Oriented Programming*, Vol. 1241, pp. 220-242, 1997.

[22] E.A. Lee. "The Problem With Threads", in *IEEE Computer*, Vol.39, No. 5, pp. 33-42. May 2006.

[23] Z. Manna and A. Pnueli, "Verification of Concurrent Programs: A Temporal Proof System", Stanford University, Dept. of Computer Science, CS-TR-83-967, 1983.

[24] Z. Manna and P. Wolper, "Synthesis of Communicating Processes from Temporal Logic Specifications", in *ACM Trans. on Programming Languages and Systems*, Vol. 6, No. 1, pp. 68-93, Jan. 1984.

[25] R. Merritt. "Wintel Will Fund Parallel Software Lab at Berkeley". EE Times. http://www.eetimes.com/showArticle.jhtml?articleID=206503988, Feb 13, 2008.

[26] G. Matos, J. Purtilo, and E. White. "Automated Computation of Decomposable Synchronization Conditions", in *Proc. 2nd IEEE High-Assurance Systems Engineering Symp.* (HASE 97), pp. 72-77, Washington, DC, Aug. 11-12, 1997.

[27] C. Oliveira, J.E.R. Cury, C.A.A. Kaestner, "Synthesis of Supervisors for Parameterized and Non-Regular Discrete Event Systems", in *1st IFAC Workshop on Dependable Control of Discrete Systems* (DCDS 07), 2007.

[28] J.S. Ostroff and W.M. Wonham, "A Framework for Real-Time Discrete-Event Control", in *IEEE Trans. on Automatic Control*, Vol. 35, Issue 4, pp. 386-397, Apr. 1990.

[29] P.J. Ramadge and W.M. Wonham. "Supervisory Control of a Class of Discrete Event Processes", in *SIAM Journal of Control and Optimization*, Vol. 25, No. 1, pp. 206-230, 1987.

[30] Σοφία: A Java Bytecode Analysis Tool, http://sofya.unl.edu/, Mar. 2008.

[31] H. Sutter. "The Free Lunch is Over: A Fundamental Turn toward Concurrency in Software", in *Dr. Dobb's Journal*, Vol. 30, No. 3, Mar. 2005.

[32] J.G. Thistle and W.M. Wonham, "Control Problems in a Temporal Logic Framework", in *Intl. Journal of Control*, Vol. 44, No. 4, pp. 943-976, 1986.

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

[34] W. M. Wonham and P.J. Ramadge, "Modular Supervisory Control of Discrete Event Systems," in *Mathematics of Control, Signal and Systems*, pp. 13-30, 1988.

[35] P.H. Welch, G.S. Stiles, G.H. Hilderink, and A.P. Bakkers. "CSP for Java: Multithreading for All", in *Architectures, Languages and Techniques for Concurrent Systems,* Vol. 57, pp 227-299, Amsterdam, the Netherlands, April 1999.

[36] T. Yavuz-Kahveci and T. Bultan. "Specification, Verification, and Synthesis of Concurrency Control Components", in *Proc. of the 2002 ACM SIGSOFT Intl. Symp. on Software Testing and Analysis* (ISSTA 2002), pp. 169-179, Roma, Italy. 2002.

[37] R. Ziller and K. Schneider, "A μ-Calculus Approach to Supervisor Synthesis", in *Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, pp. 132–143, 2003.

[38] R. Ziller and K. Schneider, "Combining Supervisor Synthesis and Model Checking", in *ACM Trans. on Embedded Computing Systems* (TECS), Vol. 4, Issue 2, pp. 221-362, May 2005.

[39] H. Zhong, W.M. Wonham, "On the Consistency of Hierarchical Supervision in Discrete-Event Systems", in *IEEE Trans. on Automatic Control*, Vol. 35, No. 10, pp. 1125-1134, Oct. 1990