Supervisory Control of Software Execution for Failure Avoidance: Experience from the Gadara Project ⋆

Yin Wang * **Hyoun Kyu Cho** ** **Hongwei Liao** **
Ahmed Nazeem *** **Terence P. Kelly** * **Stéphane Lafortune** **
Scott Mahlke ** **Spyros A. Reveliotis** ***

* *Hewlett-Packard Laboratories*
(e-mail: {yin.wang,terence.p.kelly}@hp.com)
** *Department of Electrical engineering and Computer Science, University of Michigan*
(e-mail: {netforce,hwliao,stephane,mahlke}@umich.edu)
*** *School of Industrial & Systems Engineering, Georgia Institute of Technology (e-mail: {anazeem@,spyros@isye.}gatech.edu)*

Abstract: We discuss our experience in the Gadara project, whose objective is to control the execution of software to avoid potential failures using discrete-event control techniques. We summarize our accomplishments so far and discuss future challenges. After initial work on safety of workflow scripts via supervisory control techniques, we have focused our efforts on deadlock avoidance in multithreaded C programs that use locking primitives to control access to shared data. We describe how we automatically construct automata models of workflows and Petri net models of concurrent programs. In the case of multithreaded C programs, the resulting models characterize a new class of resource-allocation Petri nets called Gadara nets. These nets enjoy structural properties that facilitate the synthesis of liveness-enforcing control policies that are maximally-permissive. We describe our strategy for run-time implementation of these control policies, especially by a technique known as code instrumentation. It is hoped that the lessons learned so far in the Gadara project will be useful in other application areas and will suggest avenues for future theoretical investigations.

Keywords: Discrete Event Systems, Supervisory Control, Petri Nets, Software Failures, Deadlock

1. INTRODUCTION

Recent years have witnessed two significant developments in software technology. First, multicore architectures force parallel programming upon the average programmer. Second, computing has becoming more and more distributed as Service Oriented Architectures (SOA) and Cloud Computing are increasingly prominent. Programmers have not been able to keep up with the rapid advances of these trends. As a result, there has been an explosion of research activities to address software failures in the context of these new challenges. Existing results from the Computer Science (CS) community can be largely divided into the following four categories: (i) new software architectures, programming paradigms, or programming languages; (ii) static analysis or verification of software; (iii) runtime analysis; (iv) post-mortem analysis (e.g., analyzing logs or core dumps). The idea of controlling software execution to avoid failures has occurred in the CS literature, typically without explicit models, e.g., using the *checkpoint/rollback recovery* method (Qin et al., 2005). Recently, there has been interest in applying analysis and control synthesis techniques from the field of Discrete Event Systems (DES) to software systems and embedded systems; see, e.g., (Liu et al., 2006; Dragert et al., 2008; Auer et al., 2009; Iordache and Antsaklis, 2009a,b; Gamatie et al., 2009; Delaval et al., 2010).

The problem of *deadlock* has received significant attention in the CS literature, due to its practical importance, especially in multicore architectures. Extensive studies along the above-mentioned four categories exist; the related work discussion in (Wang et al., 2008a) highlights some key references. Theoretical solutions for deadlock avoidance control were developed in the 1960s, most notably the Banker's Algorithm (Dijkstra, 1965). However, performance concerns have limited its applicability. It is fair to say that in practice, it is the programmer's responsibility to coordinate resources and make sure deadlock does not occur. On the other hand, the DES community has investigated the deadlock problem since the 1980s and many rigorous theoretical results are available in the literature. The context of these studies has mostly been the manufacturing domain, which shares certain similarities with modern concurrent software. A notable exception is

⋆ Authors from Michigan are supported in part by NSF grants CCF-0819882 and CNS-0930081, and by an award from HP Labs' Innovation Research Program. Authors from Georgia Tech are partially supported by NSF grants CMMI-0619978 and CMMI-0928231.

the work on deadlock analysis in Ada programs in (Shatz et al., 1996).

Our Gadara project [1] started in 2005 as a joint effort between the University of Michigan and HP Labs to apply control techniques from DES to repair important classes of software faults. Collaborators from Georgia Tech joined our team in 2008. Under the SOA theme, our efforts were initially focused on the control of *workflows* to ensure their correct execution (Wang et al., 2007). Workflow programs are very-high-level scripts written in restricted languages that emphasize concurrent control flow rather than data manipulation. They are used for instance in data center automation. Workflow programs must be verified for correctness with respect to specifications such as prevention of deadlock and avoidance of "forbidden states." We then moved on to the more difficult problem of deadlock avoidance [2] in general-purpose C programs. Our focus has been on "circular mutual-exclusion (mutex) wait" deadlocks in programs that use locking primitives to control access to shared data. Our results on this problem were first presented in (Wang et al., 2008b), with subsequent papers in CS venues (Wang et al., 2008a, 2009a), and control engineering venues (Wang et al., 2009b; Nazeem et al., 2010). The recent paper (Kelly et al., 2009) presents a general description of the Gadara project as of late 2009.

The goal of the present paper is two-fold: (i) to share the lessons that we have learned from our work on the Gadara project, in particular the challenges that we have faced in scalability and implementation of existing theoretical results; and (ii) to highlight theoretical issues that deserve further investigation. In order to make the paper self-contained, we will summarize key modeling, experimental, and theoretical results that have appeared in our prior publications cited above. We assume the reader is familiar with standard results on supervisory control of automata (Ramadge and Wonham, 1987), structural analysis of Petri nets (see, e.g., (Murata, 1989)), and control of Petri nets using the technique of Supervision Based on Place Invariants (Moody and Antsaklis, 1998).

This paper is organized as follows. Section 2 introduces the software control architecture we have adopted. Sections 3, 4, and 5 discuss modeling, control synthesis, and implementation issues, respectively. Section 4 also highlights a few open problems. We conclude the paper with a discussion in Section 6.

2. OVERVIEW

Figure 1 shows the typical control architecture we use for software failure avoidance. We build a model of the given software program, and then synthesize control logic in order to satisfy certain specifications, e.g., safety guarantees or deadlock avoidance. To implement the control logic in the software system, there are several design choices in modern hierarchical software architectures. For multithreaded C programs, we have tested two different implementations: at program level (program instrumentation) and at library level (library interposition). For workflows,

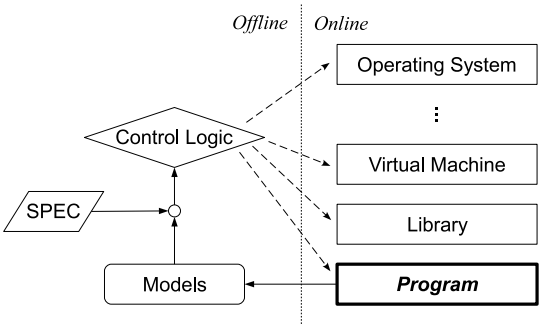


Fig. 1. Typical architecture for controlling software

it is natural to integrate the control logic within the execution engine, which is analogous to a virtual machine environment for executing workflows. People have also modified the operating system to control the execution of software (Qin et al., 2005). Performance is critical, especially in highly concurrent software. Therefore the primary objective is to move most of the computation-intensive operations offline, and make the runtime control logic as lightweight as possible.

The control synthesis loop in Fig. 1 is not to be confused with the classical feedback control loop of control engineering. Here it means that control synthesis is performed *offline*; the resulting control logic, which is embedded in the program, will affect the execution of the software at runtime.

3. MODEL CONSTRUCTION

Most control theory results start with a formal model. We know that extracting discrete-event models from real-world systems is not always easy; the right level of abstraction must be found for the problem to be analytically tractable. Computer programs are inherently discrete-event in nature; however, since computers are equivalent to Turing machines, "perfect" program verification is undecidable. Building models of the program at the right level of abstraction, tuned to the specification under consideration, is essential; we must restrict attention to a class of models for which decidability holds, and we must ensure analytical tractability for real-world programs.

Existing compilers or language parsers provide the basic analysis capabilities, so we can focus on the semantics of the model. There are often tools that exist to serve other purposes. For example, languages in SOA are almost all based on XML. Numerous libraries and open-source tools exist to parse these languages. For multithreaded software, as a byproduct of code optimization, compilers extract the basic structural information of the program. In addition to this language-level information, research tools from the field of static analysis prove to be extremely helpful by providing a high level abstraction that is closer to our needs.

3.1 Models for Workflows

Workflows are high-level scripting languages. Since they emphasize control flow over data manipulation, they are relatively easy to model. Indeed, modeling control flows of workflow languages is widespread (van der Aalst and

[1] `http://gadara.eecs.umich.edu/`

[2] Our usage of the term *deadlock avoidance* refers to the elimination of all deadlocks at runtime by control logic synthesized offline; this is consistent with the terminology used in computer science.

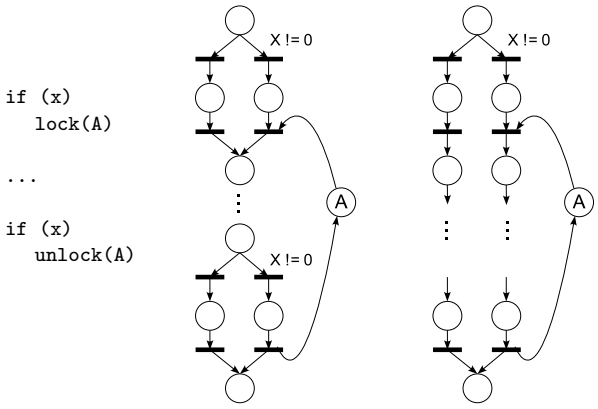
ter Hofstede, 2000). Current workflows in most applications are fairly small, with typically less than twenty *tasks* (Wang et al., 2007). The state space of the model that captures the control flow of these programs is usually manageable. In addition, these tasks are typically executed over a long time period, sometimes even with human operation involved. Control decisions need not be highly concurrent. Based on these two observations, we chose the modeling formalism of automata in our work. We note that Petri nets are also widely used to model workflows, but not to control them (van der Aalst and van Hee, 2004).

3.2 Models for Multithreaded Software

Multithreaded programs are highly concurrent. Therefore the control logic must be concurrent as well. Automata models are not practical in this case since the control decision is based on the global state, and only one transition can fire at any time. Scalability is also a major concern, as the set of reachable states of real-world programs is often unmanageable. We chose the Petri net modeling formalism for this application. In fact, modeling thread creation/termination and mutex lock/unlock operations are classical Petri net applications (cf. (Murata, 1989)). In the case of the popular Pthread library for C/C++ programs, Petri nets have been employed for modeling its programming interfaces (Kavi et al., 2002).

In practice, building Petri net models that capture the entire behavior of large programs is very challenging. In our application, the only specification is deadlock avoidance, so the model only needs to capture that aspect of the behavior of the program. Petri net models that capture deadlocks in concurrent software share similarities with those that model resource allocation systems, i.e., a set of *process subnets* interconnected by *resource places* (Reveliotis, 2005). Here resource places represent mutex locks; we call these *lock places* hereafter. Compilers provide the basic control flow information of programs in the form of *Control Flow Graphs* (CFGs), which capture the process subnets. Enhancing CFGs with lock information and acquisition/release transitions is a modeling step that is relatively easy to implement.

However, many technical difficulties arise due to C language features and programming practices. Certain dubious programming practices are impossible to model satisfactorily. For example, illegal pointer casts can obfuscate lock acquisitions. Fortunately, however, the C language standard forbids such casts, so we simply require that the programs submitted to Gadara be *legal* programs. Some legitimate practices require approximations in our models. For example, real software allocates locks dynamically, so the specific lock that is the subject of an acquisition attempt cannot always be determined during static analysis. Fortunately, real programs enclose primitive locks in a variety of different data structures, so our models can use the type of the enclosing structure—which we call the *lock type*, and which *is* known statically—to represent the lock acquired. Some need a combination of program analysis and modeling techniques: see, e.g., Fig. 2. In Fig. 2a, the variable x is *not* modified in the middle portion, so the program either acquires lock A and releases it afterwards, or does not use A at all. However, a model translated from



(a) Code (b) CFG-based model (c) Adjusted model

Fig. 2. A common pattern that violates the P-invariant

the control flow graph is depicted in Fig. 2b, which does not indicate the correlation of the two branch selections. This Petri net is not even a correct resource allocation system since there is no *P-invariant* (cf. (Murata, 1989)) associated with lock A. We devised the following fix to this problem: Once program analysis indicates the correlation, the model is adjusted as shown in Fig. 2c, where the model of the program in the middle portion is duplicated in different branches to restore the P-invariant.

Scalability is the most difficult problem. Programs are composed of functions. Building a whole program model requires inlining function calls, which results in an exponential growth of the model size. In practice, for large projects, the whole inlined program does not even fit in main memory. Since the majority of the code does not involve lock operations and therefore can be safely omitted from consideration, pruning of the model can be performed to reduce model size significantly. However, we encountered several programs whose pruned inlined models were still too large to handle. This happened for two reasons. First, real programs have numerous branches. For example, a deadlock-prone portion of code may have dozens of branches that diverge the thread away from the deadlock. We must preserve these branches in the model if the objective is to ensure maximally permissive control. Second, lock acquisition patterns are highly repetitive in real programs. For example, acquiring two locks in different orders could lead to deadlock. But each order may occur at numerous different locations that must all be included in the model. For instance, we worked with OpenLDAP, a popular open-source directory server. After pruning, its Petri net model contains 41 resource places that represent distinct lock types in the program. The `main()` function alone, without inlining any function it calls, has more than a thousand places, and the size of OpenLDAP's whole-program inlined model would exceed the memory of typical computers. With a Petri net of this size, even finding deadlocks requires innovative ideas. Fortunately, large programs are highly modular, and deadlocks typically involve only a small portion of the whole program. In view of this, our strategy was to apply a whole-program traverse procedure to identify potential deadlocks using *lock graphs*, where nodes represent locks, and a directed edge from node L_1 to node L_2 indicates that a thread may hold lock L_1 while attempting to ac-

quire L_2. Each strongly connected component (SCC) in a lock graph captures a set of correlated deadlocks, and different SCCs are independent. We then extracted the minimal inlined Petri net model for each of these SCCs, *individually* (Wang, 2009).

Lessons learned: The difficulties of modeling can never be overestimated. Modeling multithreaded programs consumed significantly more time than we originally expected. It is essential to work with domain experts to build proper models.

4. CONTROL SYNTHESIS

For workflow control, we applied standard results from supervisory control theory. Given an automaton model of a workflow and the desired specification, usually expressed in terms of forbidden states (deadlock states or otherwise), synthesis boiled down to the computation of the supremal controllable sublanguage of the specification language (Ramadge and Wonham, 1987); in this application, all events were observable but not all events were controllable.

In the remainder of this section, we focus our discussion on Petri-net-based deadlock avoidance in multithreaded C programs.

4.1 Deadlock Avoidance in Multithreaded Programs

As described in (Wang et al., 2008b), the deadlock-avoidance control synthesis algorithm in our first implementation is based on siphon analysis and the control technique of Supervision Based on Place Invariants (SBPI) (Moody and Antsaklis, 1998). It is an iterative process that finds empty siphons (specifically, those that include lock places in the model) and subsequently controls these siphons using SBPI; SBPI enforces linear inequalities, in this case, the non-emptyness of siphons, by adding new places called *monitor* or *control* places. We also apply a few heuristics to accelerate convergence, based on features of deadlocks in real programs; see (Wang et al., 2008b) for further details. This control synthesis algorithm scaled adequately for deadlocks in the open-source software we studied, including OpenLDAP, Apache, and BIND [3] (Wang, 2009). Most deadlocks we found were relatively simple, often involving only two locks; this is consistent with the observations in (Lu et al., 2008). We believe the reason is that programmers sacrifice performance for simple lock usage patterns due to the fear of deadlocks. As parallel programs become mandatory and better programming tools become available, future software may include deadlock susceptibilities that challenge the scalability of our prior approaches. For this reason, we have been working on exploiting more thoroughly the special properties of the nets that arise in this application domain.

First, we characterized a new class of Petri nets that captures the inherent features of Petri net models of multithreaded C programs; we call those *Gadara nets* (Wang et al., 2009b). We established that deadlock avoidance in these programs translates to the property of *liveness* in their corresponding Gadara net models (Wang et al., 2009a). Two notable features of Gadara nets have enabled us to develop maximally permissive control synthesis algorithms for liveness enforcement [4]. First, process subnets of a Gadara net are "state machine" Petri nets (i.e., each transition has a single incoming arc and a single outgoing arc). Second, lock places in Gadara nets have unit capacity, as we are modeling mutual exclusion locks. As a result, places in process subnets that use locks cannot contain more than one token; this results in binary markings in the places of the process subnets (excluding the idle places). The latter enables maximally permissive control using monitor places because all safe states of a Gadara net can be separated by a finite set of linear inequalities from the unsafe states (Wang et al., 2009b); in other words, a "convexity-type" property holds for the safe states. Based on the features of Gadara nets, we have recently developed in parallel two alternative liveness-enforcing control synthesis algorithms that both guarantee maximal permissiveness (Nazeem et al., 2010; Liao et al., 2010).

Lesson learned: We need to exploit to the fullest possible extent the special features of the class of models obtained in a given application domain.

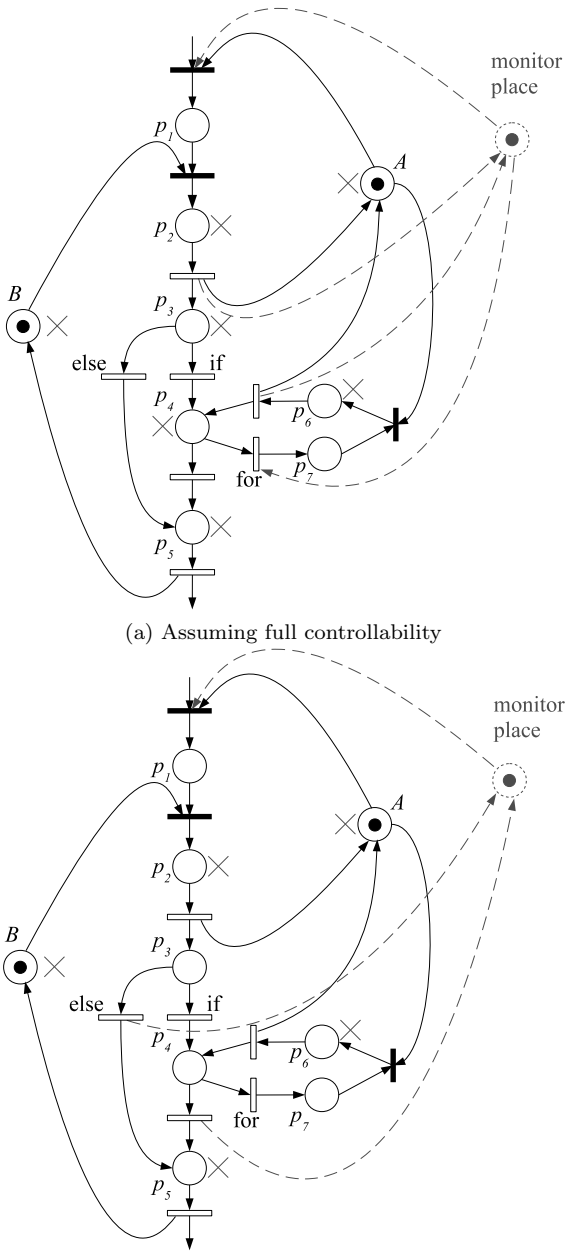
4.2 Control Problems in Practice

We discuss several interesting control problems that arise in practice. Solutions to some of these problems are under development already and other problems present opportunities for future research. By necessity, this section is more technically detailed than the rest of the paper.

Uncontrollable transitions: In practice, we can only control lock acquisition transitions. At runtime, these transitions are effectively postponed by the control logic (rather than permanently disabled), via the attached monitor place synthesized by SBPI. The issue of partial controllability occurs in practice due to numerous branches in real software. The control logic should not force one branch over another, as this would alter the behavior of the underlying program. To correctly capture this execution semantics, Gadara nets separate branching transitions from lock acquisition/release transitions (Wang et al., 2009b). Consider the simplified model of a real deadlock from OpenLDAP shown in Fig. 3, where hollow bars indicate uncontrollable transitions and the synthesized monitor place together with its arcs are shown in dashed lines. The main flow of the program (places p_{1-5}) acquires locks A and B consecutively and releases both in the same order. However, after releasing A in the main flow, there is a `for` loop (places p_{6-7}) that reacquires lock A. This creates a deadlock if meanwhile another thread has entered this section and acquired lock A in p_1. Indeed, the siphon indicated by the set of places marked by "X" captures this deadlock. The problem is that by assuming full controllability, as done in Fig. 3a, SBPI synthesizes a monitor place that can forbid the uncontrollable `for` transition leading to the loop (the dashed incoming arc in the figure).

[3] We considered the HTTP Server project from the Apache Software Foundation, and BIND is a popular domain name server program.

[4] Most existing Petri-net-based liveness enforcement algorithms are not maximally permissive since it is not always possible to represent maximally permissive control using Petri nets (Li et al., 2008).

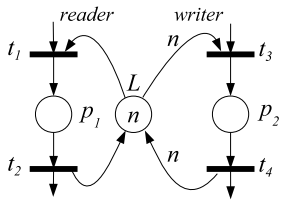


(a) Assuming full controllability

(b) With uncontrollable transitions (hollow bars)

Fig. 3. Simplified Petri net for a deadlock from OpenLDAP

Existing solutions to partial controllability in the literature usually address generic Petri nets based on *constraint transformation* (Moody and Antsaklis, 1998; Iordache and Antsaklis, 2006). As a result, they often sacrifice maximal permissiveness and can be complex. For Gadara nets, there are less complicated solutions. First, we can adjust the connecting arcs of the synthesized monitor place such that its outgoing arcs point to controllable transitions only. More specifically, observing that process subnets in Gadara nets are state machines, we can move these outgoing arcs backward in the process subnets until a lock acquisition transition is encountered. An alternative approach involves a simplified constraint transformation for Gadara nets. The key idea is to remove some places in the siphon that can lose tokens through a sequence of uncontrollable transitions and require that the rest of the siphon never becomes empty (Wang, 2009). Place p_4 is removed from the



(a) Petri net model

```
foo() {          bar() {
 ...              ...
 read(A)          read(B)
 ...              ...
 write(B)         write(A)
 ...              ...
}                }
```

(b) A RW deadlock

Fig. 4. The reader-writer lock and a deadlock example.

siphon in this case since the uncontrollable `for` transition drains its token. Subsequently, p_3 is removed because of the uncontrollable `if` transition. The state machine structure enables us to identify these token-draining places quickly. The rest of the siphon is still marked by "X". SBPI then synthesizes the monitor place with different connections, which prevent the modified siphon from becoming empty. Its outgoing arcs connect only to lock acquisition transitions, and therefore the control logic is feasible. Moreover, the control logic is maximally permissive with respect to partial controllability. The solution is shown in Fig. 3b.

Semaphores and reader/writer locks: A semaphore is essentially a resource with an arbitrary capacity. Functions `up` and `down` allocate and deallocate its units, respectively. Semaphores can be modeled by resource places with possibly more than one initial token. A reader-writer lock allows concurrent readers but the writer requires exclusive access to the resource. It is typically modeled by the Petri net in Fig. 4a, where the initial number of tokens in L and the arc weights for the writer are all n, the maximum number of concurrent readers. We may set n equal to the number of threads in the system. In practice, semaphores are relatively rare, but reader-writer locks are frequently used.

Both of these synchronization primitives result in "non-Gadara nets" because places in process subnets may have multiple tokens now. An important consequence is that the safety region in the hyperspace defined by reachable markings may be "non-convex" (i.e., it is no longer separable by linear inequalities). Therefore, SBPI cannot, in general, synthesize maximally permissive control logic in this case. For example, in Fig. 4b, if two threads are executing `foo()` and both get the reader lock for A, the state is safe. So is the case when two threads hold the reader lock for B while executing `bar()`. But if they execute the two functions separately in parallel, deadlock could occur when each holds a reader lock and is waiting for the other writer lock. The marking of the last situation is a linear combination of the first two, and therefore not linearly separable. Maximally permissive control in this case could be achieved using non-linear policy structures and representations, like the Generalized Algebraic Deadlock Avoidance Policies (GADAPs) proposed in (Reveliotis et al., 2007).

In addition, the modeling method depicted in Fig. 4a raises a scalability concern: The size of the reachable state space is a combinatorial function of the maximum number of concurrent readers, n, and the number of places in the process subnet of a reader; both can be large in practice. Intuitively, the upper limit n should not affect the control logic as long as it is large enough to capture all potential deadlocks. More specifically, we have the following remark

Thread 1	Thread 2
...	...
`read_pipe(out)`	`write_pipe(err)`
...	...
`read_pipe(err)`	`write_pipe(out)`
...	...

Fig. 5. A deadlock from Apache HTTP server software.

that alleviates the above-mentioned scalability problem: In the considered class of nets, if a state M is safe and a non-zero entry in it represents readers, by adding more readers (tokens) to the same entry, the resulting state (if reachable) is still safe. Similarly, if an unsafe state has an entry greater than one that represents multiple readers, the state is still unsafe even if we remove all surplus readers until only one is left. Therefore, instead of the binary markings in Gadara net states, we may use ternary markings with entries $\{0, 1, \omega\}$ to efficiently represent the state space. Exploring the state space in this case is analogous to the construction of the coverability graph. Our results on this approach will be presented elsewhere.

Open Problems: Existing workflow theory considers primarily the execution logic of a single process instance and tries to guarantee some behavioral properties such as proper termination (van der Aalst and van Hee, 2004). Resource sharing during the concurrent execution of more than one process instances deserves more attention. For example, a human operator should not be assigned more than a certain number of cases at the same time, or certain other resources might be exclusively allocated to cases, etc. Existing control theory for resource allocation systems could be applicable to these problems.

In multithreaded software, synchronization primitives other than locks can also lead to deadlock. We have demonstrated that siphon-based analysis applies to deadlocks caused by *condition variables* (Wang et al., 2009a). Figure 5 illustrates another type of deadlock. By POSIX standards, if a *pipe* is empty, the `read` call must wait until another thread writes to the pipe, while writing to a pipe does not block normally since the data is stored in the buffer. But if the buffer is full, the writer is blocked until another reader consumes the pipe. Therefore, Fig. 5 deadlocks if the error message to be written by Thread 2 exceeds the buffer size of pipe `err`. Under the current software architecture, only *system calls* in the operating system may block execution, and therefore possibly cause deadlock. Therefore, understanding and modeling all blocking system calls are problems worthy of investigation. Finally, there are other concurrency issues where DES control theory could be applicable, including livelock, race conditions, starvation and fairness.

Lesson learned: Once you understand the application domain thoroughly, as the saying goes, "nothing is more practical than a good theory."

5. CONTROL LOGIC IMPLEMENTATION

5.1 General Considerations

Modern software is extremely complex and performance is critical. Therefore, the implementation of the control logic should be minimally intrusive. Controlling software introduces two types of overhead: (i) control logic runtime decisions and (ii) transitions blocked as a result of the control decisions. Due to practical considerations, the average complexity of the runtime control decision should be constant time. Maximally permissive control logic minimizes the second type of overhead.

In addition, it is important to understand the software system and make judicious design choices. In the workflow domain, most workflows run over long time periods and some involve human workers executing tasks. Therefore the performance constraints on control logic overhead are relatively lenient. It is acceptable to represent the control logic using automata models, where the execution engine consults and updates the "controller automaton" at every transition. In highly concurrent software, however, introducing a global bottleneck for control decisions is unacceptable. Fortunately, SBPI allows concurrent implementation as monitor places are local and decentralized. We fully exploited this advantage in our implementation.

Concurrent programs may acquire a lock to perform a very simple task, e.g., checking or updating a single shared variable. If the lock acquisition transition is associated with many monitor places, checking and updating tokens in them could dominate the overall computation time. Therefore, it is important to *minimize* the number of monitor places. For example, places with redundant control logic should not be included. If the reachable state space of the Gadara net is available, we can use classification theory to calculate the minimum number of linear inequalities needed to separate safe and unsafe states (Nazeem et al., 2010). Then we can implement these linear inequalities as monitor places using SBPI. On the other hand, if the reachable state space is too large to be enumerated, then we must resort to structural analysis (e.g., siphon-based analysis) to achieve the same goal of minimization of monitor places; we present our current results on this problem in (Liao et al., 2010).

5.2 Implementation Details

After control synthesis, we need to implement the control logic in the software system. As stated in Section 2 and depicted in Fig. 1, the control logic can be implemented at different layers. *Source code instrumentation* modifies the program source code to include the control logic. The control logic is then compiled together with the program to obtain an instrumented binary. At runtime, the instrumented binary is executed the same way as the unmodified one; the execution is redirected to the control logic source code whenever control is needed. *Library interposition* intercepts calls to libraries of software routines, e.g., the POSIX threading library, and invokes control as needed. This method does not modify the program, and therefore is less intrusive. However, it only intercepts library calls. Other transitions in the program, e.g., branch selections, are not visible. Therefore, the control synthesis algorithm must handle partial observability, and the resulting control logic is not maximally permissive in general. At the *process virtual machine level* (e.g., Java VM), full observability is restored in the bytecode. In addition, the virtual machine may collect runtime information that can be used to refine

the model and re-synthesize better control logic. For example, we can detect if two locks will alias at runtime, and remove monitor places that control corresponding false positives. Virtual machines usually incur additional overhead, and thus may not be appropriate for performance critical applications. *Operating systems* intercept only system calls and therefore share similar limitations to the library interposition method. However, operating systems schedule threads and locks. If properties such as fairness and starvation are the control goals, this is the most appropriate place for implementing the control logic.

For deadlock avoidance in multithreaded C programs, we have experimented with the first two choices: source code instrumentation and library interposition. For the purpose of maximally permissive control, we focused on implementation through code instrumentation. Since we are going back to the software program itself (rather than to its model), the difficulties in control logic implementation are closely related with those described for the modeling phase. Specifically, we extracted only a portion of the program that pertains to deadlocks, since including irrelevant aspects of the software would bloat its model unnecessarily.

Now, the problem is how to map the execution of the *whole* program onto its *partial* model in order to invoke control properly. In some cases, our implementation adds parameters to function calls that pass necessary state information. Another problem is that when monitor places are connected to one transition, together with some lock, the control logic requires that the checking and updating of tokens in all these control places and the lock place be *atomic*. There is no direct support for this atomic operation in the existing hardware or software architectures. We implemented this using locks and condition variables in the Pthread library, as described in (Wang et al., 2009a). After careful design, we were able to reduce the runtime control overhead from negligible in most cases to about 18% in highly pessimistic and unnatural configurations; this was deemed acceptable by the operating systems community when we presented our results to them.

Lessons learned: Research in software systems is largely experimental in nature, especially when performance optimization is the goal. Control-theoretic results are often orthogonal to implementation considerations. You must tune your implementation and convince domain experts that your overhead is acceptable. Finally, it is relatively easy to experiment with computer systems. A client-server test-bed can be built at low cost with open-source software and meaningful experiments with real-world workloads can be performed on it.

6. DISCUSSION

6.1 Current Work

As services flourish on the Internet and are consolidated in "the Cloud," it becomes increasingly important to *automatically* compose these services (possibly from different vendors) and control the execution of the composite services, often represented as workflows. We hope to build upon our existing work, and integrate the workflow control engine (which will embed the discrete-event controller) into products.

We are also continuing our work on control of multithreaded programs. For the modeling phase, we are exploiting state-of-the-art compiler analysis techniques to obtain a more accurate model. As stated in Section 4.1, we have exploited the features of Gadara nets and advanced the relevant control theory. Our ongoing work is exploring, among other challenges, the research frontier discussed in Section 4.2. Finally, we are thinking about implementing the control logic at the virtual machine level. We hope to use runtime information to validate or improve our model.

6.2 Closing Remarks

The area of computer systems is full of opportunities for control theory, continuous-variable as well as discrete-event. Hellerstein et al. have applied classical control theory to quantitative aspects of software systems (Hellerstein et al., 2004), e.g., performance. We have applied discrete event techniques to specific software failure avoidance problems and demonstrated our results to the Operating Systems (Wang et al., 2008a) and Programming Languages (Wang et al., 2009a) communities. Our work has necessitated a synergy between compiler and static analysis methods in computer systems and controller synthesis techniques in discrete event systems. In both of the application domains that we have considered, deep static analysis results already existed in the literature. We benefitted significantly from studying these methods. For instance, our model decomposition technique described in Section 3.2 was inspired by the well-known static deadlock detection method in (Engler and Ashcraft, 2003). Generally speaking, static analysis helps to identify the specific instances where online (feedback) control will help. When this occurs, modeling is usually the most difficult part, as there is often existing control-theoretic results that can be exploited (or improved upon). Finally, in our case, the blend of expertise of our team, from compiler experts to operating systems specialists and discrete event systems theorists, was key to our accomplishments so far.

REFERENCES

Auer, A., Dingel, J., and Rudie, K. (2009). Concurrency control generation for dynamic threads using discrete-event systems. In *Allerton Conference on Communication, Control and Computing*.

Delaval, G., Marchand, H., and Rutten, E. (2010). Contracts for modular discrete controller synthesis. In *ACM Conference on Languages, Compilers and Tools for Embedded Systems*.

Dijkstra, E.W. (1965). Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9), 569.

Dragert, C., Dingel, J., and Rudie, K. (2008). Generation of concurrency control code using discrete-event systems theory. In *ACM International Symposium on Foundations of Software Engineering*.

Engler, D. and Ashcraft, K. (2003). RacerX : effective, static detection of race conditions and deadlocks. In *ACM Symposium on Operating Systems Principles*.

Gamatie, A., Yu, H., Delaval, G., and Rutten, E. (2009). A case study on controller synthesis for data-intensive embedded systems. In *International Conference on Embedded Software and Systems*.

Hellerstein, J.L., Diao, Y., Parekh, S., and Tilbury, D.M. (2004). *Feedback Control of Computing Systems*. Wiley.

Iordache, M.V. and Antsaklis, P.J. (2006). *Supervisory Control of Concurrent Systems: A Petri Net Structural Approach*. Birkhäuser.

Iordache, M.V. and Antsaklis, P.J. (2009a). Petri nets and programming: A survey. In *American Control Conference*.

Iordache, M.V. and Antsaklis, P.J. (2009b). Synthesis of concurrent programs based on supervisory control. Technical report, University of Notre Dame.

Kavi, K.M., Moshtaghi, A., and Chen, D.Y. (2002). Modeling multithreaded applications using Petri nets. *International Journal of Parallel Programming*, 30(5), 353–371.

Kelly, T., Wang, Y., Lafortune, S., and Mahlke, S. (2009). Eliminating concurrency bugs with control engineering. *Computer*, 42(12), 52–60.

Li, Z., Zhou, M., and Wu, N. (2008). A survey and comparison of Petri net-based deadlock prevention policies for flexible manufacturing systems. *IEEE Trans. on Systems, Man, and Cybernetics—Part C*, 38(2), 173–188.

Liao, H., Lafortune, S., Reveliotis, S., Wang, Y., and Mahlke, S. (2010). Synthesis of maximally-permissive liveness-enforcing control policies for Gadara Petri nets. Technical Report, EECS Department, University of Michigan.

Liu, C., Kondratyev, A., Watanabe, Y., Desel, J., and Sangiovanni-Vincentelli, A. (2006). Schedulability analysis of Petri nets based on structural properties. In *International Conference on Application of Concurrency to System Design*.

Lu, S., Park, S., Seo, E., and Zhou, Y. (2008). Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ACM Conference on Architectural Support for Programming Languages and Operating Systems*.

Moody, J.O. and Antsaklis, P.J. (1998). *Supervisory Control of Discrete Event Systems Using Petri Nets*. Kluwer Academic Publishers.

Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4), 541–580.

Nazeem, A., Reveliotis, S., Wang, Y., and Lafortune, S. (2010). Optimal deadlock avoidance for complex resource allocation system through classification theory. In *International Workshop on Discrete Event Systems*.

Qin, F., Tucek, J., Sundaresan, J., and Zhou, Y. (2005). Rx: Treating bugs as allergies—a safe method to survive software failure. In *ACM Symposium on Operating Systems Principles*.

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

Reveliotis, S.A., Roszkowska, E., and Choi, J.Y. (2007). Generalized algebraic deadlock avoidance policies for sequential resource allocation systems. *IEEE Trans. on Automatic Control*, 52(12), 2345–2340.

Reveliotis, S.A. (2005). *Real-Time Management of Resource Allocation Systems: A Discrete-Event Systems Approach*. Springer.

Shatz, S.M., Tu, S., Murata, T., and Duri, S. (1996). An application of Petri net reduction for Ada tasking deadlock analysis. *IEEE Trans. on Parallel and Distributed Systems*, 7(12), 1307–1322.

van der Aalst, W. and van Hee, K. (2004). *Workflow Management: Models, Methods, and Systems*. The MIT Press.

van der Aalst, W. and ter Hofstede, A. (2000). Verification of workflow task structures: A Petri-net-based approach. *Information Systems*, 25(1), 43–69.

Wang, Y. (2009). *Software Failure Avoidance Using Discrete Control Theory*. Ph.D. thesis, University of Michigan.

Wang, Y., Kelly, T., Kudlur, M., Lafortune, S., and Mahlke, S.A. (2008a). Gadara: Dynamic deadlock avoidance for multithreaded programs. In *USENIX Symposium on Operating Systems Design and Implementation*.

Wang, Y., Kelly, T., Kudlur, M., Mahlke, S., and Lafortune, S. (2008b). The application of supervisory control to deadlock avoidance in concurrent software. In *International Workshop on Discrete Event Systems*.

Wang, Y., Kelly, T., and Lafortune, S. (2007). Discrete control for safe execution of IT automation workflows. In *ACM EuroSys Conference*.

Wang, Y., Lafortune, S., Kelly, T., Kudlur, M., and Mahlke, S. (2009a). The theory of deadlock avoidance via discrete control. In *ACM Symposium on Principles of Programming Languages*.

Wang, Y., Liao, H., Reveliotis, S., Kelly, T., Mahlke, S., and Lafortune, S. (2009b). Gadara nets: Modeling and analyzing lock allocation for deadlock avoidance in multithreaded software. In *IEEE Conference on Decision and Control*.