

# Discrete Control for Safe Execution of IT Automation Workflows

Yin Wang<sup>\*</sup>  
University of Michigan  
Ann Arbor, Michigan, USA  
yinw@eecs.umich.edu

Terence Kelly  
Hewlett-Packard Laboratories  
Palo Alto, California, USA  
terence.p.kelly@hp.com

Stéphane Lafortune<sup>†</sup>  
University of Michigan  
Ann Arbor, Michigan, USA  
stephane@eecs.umich.edu

## ABSTRACT

As information technology (IT) administration becomes increasingly complex, workflow technologies are gaining popularity for IT automation. Writing correct workflow programs is notoriously difficult. Although static analysis tools are available, fixing defects remains manual and error-prone. This paper applies discrete control theory to IT automation workflows. Discrete control detects flaws in workflows just as static analysis does, and more importantly it also allows safe execution of flawed workflows by dynamically avoiding run-time failures. Our approach can guarantee compliance with certain requirements and can partially decouple requirements from software, reducing the need to modify the latter if the former change. We have implemented a discrete control module for a real IT automation system. Experiments with workflows from a real production system and with randomly generated workflows show that our approach scales to workflows of practical size.

## Categories and Subject Descriptors

H.1 [Models and Principles]: Miscellaneous; H.4.1 [Information Systems Applications]: Office Automation—*Workflow management*; D.4.5 [Operating Systems]: Reliability

## General Terms

Management, Reliability

## Keywords

Workflow, Discrete Control, Deadlock Avoidance

## 1. INTRODUCTION

Information technology (IT) administration is increasingly automated. Automating routine procedures such as software deployment can increase infrastructure agility and reduce staff costs [6]. Automating extraordinary procedures such as disaster recovery can reduce time to repair [15]. Human operator error is a major cause

<sup>\*</sup>Y.W. acknowledges support from HP Labs.

<sup>†</sup>S.L. acknowledges support from NSF grant ECS-0624821.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*EuroSys'07*, March 21–23, 2007, Lisboa, Portugal.  
Copyright 2007 ACM 9781595936363/07/0003...\$5.00

of availability problems in large data centers [24], and automation can reduce such problems.

Workflows—concurrent programs written in very-high-level special languages—are an increasingly popular IT automation technology. Like conventional scripting languages, workflow languages facilitate composition of coarse-grained IT administrative actions, treated as atomic tasks. However workflow languages differ in several ways: they impose more structure, emphasize control flow rather than data manipulation in their language features, and provide far better support for concurrency. Production workflow systems include stand-alone products [13] and extensions to legacy offerings [25]. Recent research explores workflows for wide-area administration [1], storage disaster recovery [15], and testbed experiments management [10].

Like multi-threaded programming, workflow programming is notoriously difficult and error prone. Concurrency, resource contention, race conditions, and similar issues lead to subtle bugs that can survive software testing undetected. Subtly flawed disaster-recovery workflows are particularly alarming because they can exacerbate crises they were meant to solve. For example, Keeton discovered priority-inversion deadlocks in storage-recovery workflows only after scheduling their tasks [14]. Some workflow languages trade flexibility, convenience, and expressivity for safety by restricting the language in such a way that certain pitfalls, e.g., deadlock, are impossible. In most cases, however, responsibility remains with the programmer: An extensive study of commercial workflow products found that deadlocks are possible in the majority [17].

Fortunately, the restricted expressivity of workflow languages enables very powerful static analysis. Existing tools can determine if a workflow can reach user-specified undesirable states and can detect many other defects (e.g., deadlock/livelock). Such tools have found bugs in production workflows that were thought to be correct [21]. Static analysis of workflows yields far fewer spurious warnings and undetected flaws than static analysis of general programming languages via heuristic methods [2, 32, 33] or model checking [18, 37]. Static workflow analysis therefore provides a reliable off-line way to validate IT administrative actions before they are performed, complementary to dynamic validation [23] and post-mortem root cause localization [16].

Static analysis, however, merely *detects* defects; repair remains manual, time-consuming, error-prone, and costly. Manually corrected workflows are often less natural, less readable, and less efficient than the flawed originals, especially when corrections address

bizarre corner cases. Furthermore maintenance costs can be high if workflows themselves carry the full burden of compliance with requirements: Manual maintenance is necessary for workflows that were previously satisfactory but that fail to meet updated requirements. Finally, static workflow analysis is pessimistic in the sense that it assumes worst-case execution and ignores opportunities for dynamic failure avoidance.

This paper shows how *discrete control theory* can allow safe execution of unmodified flawed workflows by dynamically avoiding undesirable execution states, e.g., states that violate dependability requirements. Our approach can reduce both development and maintenance costs: By externally enforcing compliance with some requirements, it allows programmers to write straightforward workflows instead of perfect ones. By partially decoupling workflow software from requirements, it reduces the need to alter the former when the latter change. Whereas static workflow analysis assumes Murphy's Law, discrete control recognizes that anything that can be *prevented* from going wrong need not be repaired.

Classical control theory has been applied to several performance-related IT problems recently [12]. Whereas classical control considers continuous-state systems whose dynamics are described by differential equations, discrete control theory deals with *discrete event systems*, i.e., dynamic systems with discrete state spaces and event-driven dynamics [28]. Thus it is better suited to problems surrounding qualitative functional requirements, e.g., safety and dependability problems. Discrete control theory has been applied in domains ranging from manufacturing [5] to telecommunications [7]. However, to the best of our knowledge, it has never before been implemented for any IT automation or CS systems problem.

The contributions of this paper include: i) the introduction of a new body of control theory into the CS systems area; ii) a new method to address software defects and changing requirements in the workflow domain; iii) a novel architecture for incorporating discrete control with a workflow execution engine that guarantees safe execution at the workflow level of abstraction; and iv) experiments demonstrating that the discrete control logic synthesis algorithms at the core of our method are sufficiently scalable to be practical in the workflow domain.

The remainder of this paper is organized as follows: Section 2 introduces discrete control theory and its capabilities. Section 3 reviews complementary techniques for achieving similar goals, and explains how discrete control gives us greater capabilities than existing methods either individually or in combination. Section 4 describes the workflow control architecture that we are implementing and how discrete control theory operates within it. Section 5 presents examples that illustrate how discrete control dynamically avoids run-time failures in workflows for IT automation. Section 6 presents our performance evaluation demonstrating that our implementation of discrete control logic synthesis scales to workflows of practical size. Section 7 concludes with a discussion.

## 2. DISCRETE CONTROL THEORY

Over the past two decades a large body of research on the control of discrete event systems has emerged. This section outlines discrete control theory and describes the capabilities that we exploit in the present paper.

### 2.1 Framework & Capabilities

Discrete control requires a model of the system to be controlled. Several modeling formalisms are used in the literature; we use a finite state automaton  $G$  representing all workflow execution states reachable from the initial state, and we automatically generate  $G$  from a workflow. Workflow control structures and the corresponding state transitions in  $G$  are labeled as either controllable or uncontrollable; the former can be prevented or postponed at run time, but the latter cannot. Examples of controllable transitions in workflows include attempts to install software or migrate data. The times at which such attempts conclude, and whether they succeed or fail, are uncontrollable transitions.

In the most general discrete control methods, undesirable behaviors are specified as sublanguages of the regular language associated with automaton  $G$ . We expose a simpler mode of specification: The workflow programmer defines *forbidden states* representing undesirable execution states, e.g., states that violate dependability requirements. In our current implementation, the control flow of a workflow is described by a Petri net [22]. An execution state corresponds to a marking of the Petri net, and forbidden states may be defined straightforwardly and conveniently as a function of the marking. Furthermore some undesirable states, such as livelock and deadlock states, are automatically labeled forbidden during workflow-to-automaton translation, as are *terminal states* corresponding to satisfactory workflow completion.

The goal of discrete control is to ensure that the system reaches a terminal state without entering forbidden states, even if worst-case sequences of uncontrollable state transitions occur. This goal is achieved in two stages: First, an *offline control synthesis* stage uses the system model  $G$  and the specification of terminal and forbidden states to automatically synthesize a discrete controller. Then during *online dynamic control* the controller selectively disables controllable transitions based on the current execution state.

The synthesized controller should have two properties: First, it should be minimally restrictive, disabling transitions only when necessary to avoid forbidden states and livelock/deadlock. Second, it must not prevent successful termination. A controller with these properties restricts the system to its unique *maximally permissive controllable non-blocking sublanguage*, and existing methods can synthesize such a controller [28]. If no such controller exists, i.e., if it is impossible to ensure safe execution, then the system is fundamentally uncontrollable and control synthesis returns an error message. In this case, the programmer may fix the workflow, or an operator may choose to execute it anyway if she believes the probability of reaching forbidden states via uncontrollable transitions to be small. In the latter case, once the workflow enters a state where forbidden states *can* be avoided, the controller's safety guarantees are restored.

Control synthesis requires time quadratic in the size of  $G$  in the worst case. However, control synthesis is an offline operation; in the workflow domain, it does not increase execution time. Furthermore in our experience with both real and randomly-generated workflows, the time required for control synthesis is roughly *linear* in the size of  $G$ . Online dynamic control adds negligible constant-time overheads during workflow execution. Although it is possible to construct worst-case workflows whose state spaces are exponential in the number of tasks, our experience with real commercial workflows convinces us that the worst case is not typical in practice. Our discrete control synthesis implementation scales to work-

flows of practical size; Section 6 presents quantitative results on this question.

## 2.2 Extensions

We assume throughout this paper that all state changes are observable to the run-time controller, i.e., we restrict attention to *fully observable* discrete event systems. However, it is worth mentioning that extensions to the discrete control methods that we employ in this paper exist to address *partially observable* systems. In a partially observable system, transition labels in the system model  $G$  are either *observable* or *unobservable*; the former are directly and explicitly visible to the run-time controller but the latter are not. In the workflow domain, examples of observable transitions might include successful termination of tasks and exogenous inputs to the system such as request arrivals. Examples of unobservable transitions might include silent data corruption in disks and silent software failures.

Partial observability raises interesting challenges. One problem is to infer the occurrence of unobservable transitions from observable ones; this is known as the *diagnosis* problem [31] in discrete event systems. Discrete-event diagnosis methods have been applied to commercial printer/copier machines to infer failure events during system operation [30]. Diagnosis problems become more challenging in distributed environments, where the information (e.g., observable transitions) is distributed. Diagnosis of distributed systems is an active area of research in discrete control theory [35].

Another challenge raised by partial observability is to extend control synthesis algorithms to partially-observable systems [19]; the problem here is to avoid forbidden states even though we cannot observe every transition and thus are uncertain about the current system state. The solution is to build an observer automaton<sup>1</sup> that, based on observable transitions, estimates the set of states the system could possibly be in. Then, for every state in the estimate set, the controller disables transitions that can lead to forbidden states unavoidably. Similarly to the case with only uncontrollable transitions, we desire non-blocking execution, permissive control, and other properties. After building the observer, the complexity of control synthesis is polynomial to the number of observer states. With partial observability, the maximally permissive controllable non-blocking sublanguage is no longer unique. Different control actions may result in different incomparable sublanguages. Due to the need of building an observer automaton, discrete control synthesis for partially observable systems can be computationally challenging [8].

## 3. RELATED METHODS

This section surveys techniques aimed at problems similar to those that we address using discrete control theory, e.g., dynamic failure avoidance, and describes how our approach differs from them.

Rinard *et al.* have proposed “failure-oblivious computing” to improve server availability and security [29]. This approach manufactures values for invalid memory reads in C programs, potentially introducing new behaviors into the program. Our application of discrete control to workflows can only restrict the space of possible workflow execution states but cannot expand it.

<sup>1</sup>Building an observer automaton is similar to the process of building a deterministic finite-state automaton from a non-deterministic finite-state automaton.

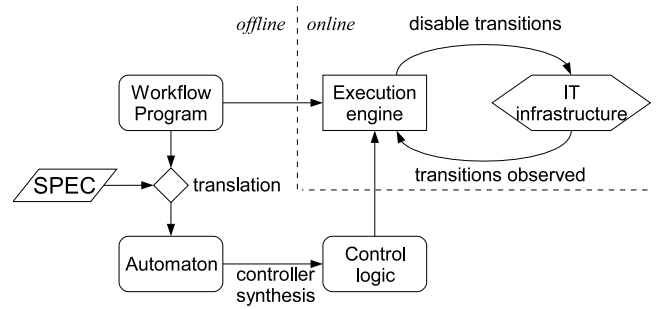


Figure 1: Workflow control architecture.

Allocating resources among concurrent computing processes can lead to deadlock, and methods such as the “banker’s algorithm” of Dijkstra [9] can dynamically avoid deadlock by postponing or denying resource requests. In contrast to the hard-coded control logic used in these methods, discrete control automatically synthesizes control logic from a system model and a behavioral specification. The banker’s algorithm addresses resource allocation problems in which all state transitions are controllable and observable. Discrete control is applicable to a wider range of problems and can cope with partial controllability and/or partial observability.

Bar-David and Taubenfeld have explored methods for automatically generating solutions to mutual exclusion problems [3]. Their approach exhaustively generates all syntactically correct algorithms up to a specified size limit and uses a model checker to eliminate incorrect ones. By contrast, the control logic synthesis methods of discrete control theory handle a far broader range of problems and do not rely on brute-force generation of candidate solutions.

Qin *et al.* have developed a software run-time control system that can survive software failures by rolling back a program to a recent checkpoint and re-executing the program in a modified environment [27]. One limitation of this approach is that not all aspects of program execution are invertible, especially in a distributed environment. In addition, as there is no system model, the re-execution must exhaustively search all possible environment modifications. Our approach builds the model *off-line* and designates portions of it unsafe. The run-time controller can then avoid unsafe states efficiently without on-line trial-and-error that risks non-invertible state transitions.

## 4. CONTROL ARCHITECTURE

Figure 1 depicts the architecture of a workflow control system. We begin with a workflow consisting of atomic tasks organized via control-flow structures. Typical structures include sequence, iteration, AND-forks to spawn parallel executions, controllable OR-forks analogous to if/else statements, uncontrollable OR-forks that model uncontrollable state transitions, and AND/OR joins that “reconnect” control flow following a fork. Some workflow languages offer extensions, e.g., BPEL includes structures to define precedence constraints among tasks, which are called *control links*.

First, a translator converts the workflow into an automaton that models its control flow and reachable state space. Transitions in the automaton represent task invocation/completion, control structure entrance/exit, and resource acquisition/release; states represent the results of these transitions. The translator identifies uncon-

trollable transitions by high-level workflow features (e.g., uncontrollable OR-forks) and can automatically detect livelock/deadlock states in the automaton and flag them as forbidden. The programmer may define additional application-specific forbidden states, e.g., via program annotations and logical predicates on execution states.

In our current implementation, the flow of control in a workflow is represented by a Petri net, and execution states in the automaton correspond to markings of the Petri net. Forbidden states may be defined by specifying a function that maps the Petri net marking vector to a designation of “forbidden” or “not forbidden.” The relationship between a high-level workflow and the Petri net that describes its control flow is straightforward, and the use of Petri nets to model the control flow of workflows is widespread [34]. It is convenient and natural to specify forbidden states in terms of Petri net markings. For instance, it is easy to forbid one of two concurrently-executing branches from completing before the other has begun; such a restriction allows us to handle the example problem described in Section 5.1.

Discrete control theory provides more general modes of specification (which we have not implemented) corresponding to more general restrictions on workflow execution: In principle, discrete control theory allows us to restrict execution to an arbitrary sub-language of the regular language associated with the automaton representing control flow in the workflow. Conceptually, such a restriction may be represented by a regular expression.

After we have obtained the annotated automaton describing reachable execution states, a control synthesis algorithm from discrete control theory uses the automaton and the associated sets of terminal and forbidden states to generate control logic that specifies which controllable transitions should be disabled as a function of current execution state. Both workflow→automaton translation and control synthesis are offline operations.

At run time, the workflow execution engine tracks execution state and refrains from executing controllable transitions that the control logic disables in the current state. The result is that the system will never enter a forbidden state, regardless of uncontrollable transitions that may occur during execution. If a workflow is fundamentally uncontrollable, i.e., if it is impossible for any controller to guarantee safe execution, we will learn this as a by-product of control logic synthesis. If an uncontrollable workflow is executed anyway and good luck leads it to a state from which safety *can* be ensured, the controller’s safety guarantees are restored.

The specific workflow execution engines that inspired our research execute workflows without human intervention. In principle, however, nothing prevents the application of our approach to situations where workflow execution is partially or completely manual. Regardless of whether the execution engine is a computer program or human operator, discrete control plays the same role: it tells the execution engine what subset of the controllable state transitions that are *possible* in the current state are *safe*.

Our workflow control architecture allows the incorporation of static analysis, dynamic validation, and post-mortem debugging tools. However discrete control offers advantages beyond what these complementary techniques can provide individually or in combination. By guiding workflows to successful conclusion without traversing forbidden states, discrete control strives to reduce the need for post-mortem debugging at the workflow level (run-time failure remains

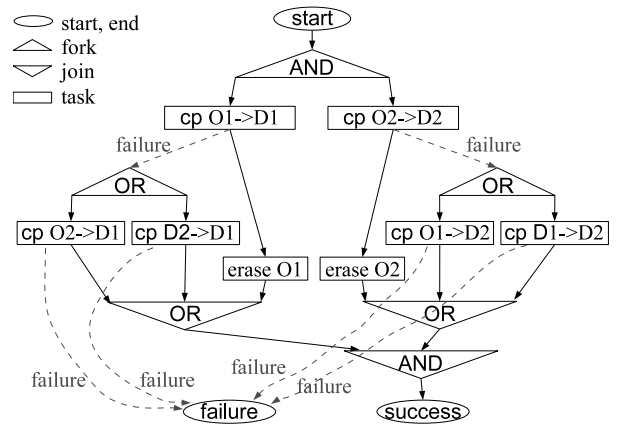


Figure 2: Data migration workflow.

possible within constituent tasks, of course). By permitting safe execution of unmodified flawed workflows, dynamic control relieves programmers of the burden of writing flawless workflows. By decoupling behavioral specifications from workflows, it reduces the need to modify workflows when requirements change.

All of the components depicted in Figure 1 are fully implemented, but not yet integrated. We hope to integrate our discrete control synthesis module into a workflow execution engine developed at HP Labs that is used in experimental IT automation projects, e.g., for the back-end resource provisioning in thin-client desktop systems [11]. We are also exploring the possibility of applying discrete control to a second workflow execution environment used for IT automation. In both cases progress has been slow largely due to organizational issues, e.g., uncertainty surrounding the project roadmaps of these workflow systems.

## 5. EXAMPLES

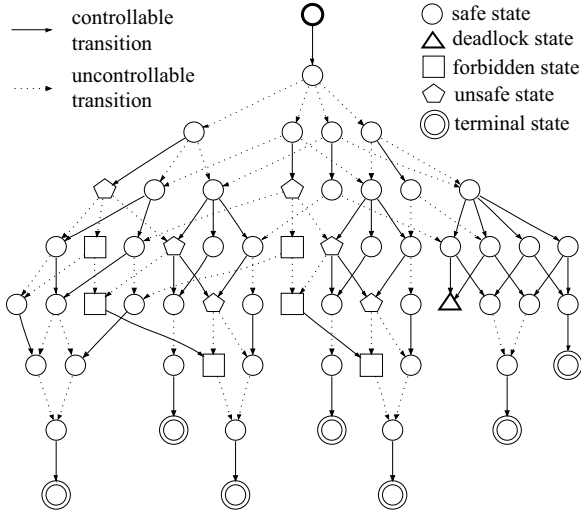
This section presents two examples that illustrate how discrete control can allow safe execution of flawed workflows and avoid the need to revise workflows when requirements change.

### 5.1 Data Migration Workflow

Figure 2 shows a simplified data migration workflow that moves two original copies of a data set, O1 and O2, to destinations D1 and D2. The two branches of the AND-fork represent concurrent copy-erase operations. Uncontrollable “failure” transitions model the possibility that copy operations may fail; other uncontrollable events include task completions. If the O1→D1 copy in the left branch fails, the workflow will retry from O2 or D2. However the workflow does not specify which; this decision is made by the execution engine, perhaps guided by performance considerations. If the second attempt to create D1 also fails, the workflow will end in global failure. The right branch, responsible for creating D2, is symmetric. Tasks require exclusive access to copies of data, e.g., the D2→D1 copy must wait until O2→D2 has finished. For readability Figure 2 omits resource management aspects of the model.

The problem with this workflow is that if both O1→D1 and O2→D2 tasks fail, and if the response to these failures are attempts to copy D2→D1 and D1→D2 respectively, then the workflow deadlocks with each branch waiting for the other to complete. Static analysis alone can detect this problem, requiring a programmer to re-

pair the flaw manually. However even for this simple bug in this small workflow, repair can be a tedious and error-prone affair if the solution must be safe (no new deadlocks), efficient (recycle storage as soon as possible), and flexible (allow several data copy sources). Discrete control allows us to safely execute the flawed workflow without modification. The controller will avoid the deadlock state by disabling either  $D2 \rightarrow D1$  or  $D1 \rightarrow D2$  if both  $O1 \rightarrow D1$  and  $O2 \rightarrow D2$  fail. Figure 3 depicts the state-space automaton for our example workflow. There is one deadlock state corresponding to the above double failure. By disabling one of the copy operations after failures, discrete control can avoid the deadlock.



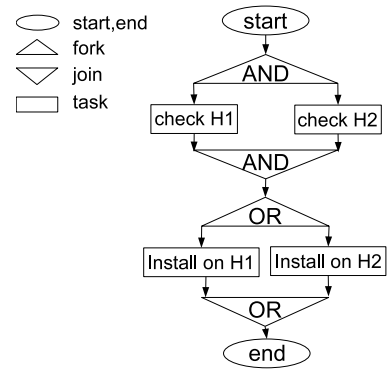
**Figure 3: State-space automaton for workflow of Figure 2. The deadlock state corresponds to double failure. Forbidden states contain neither two origin nor two destination copies. Unsafe states may reach forbidden states via sequences of uncontrollable transitions.**

Now suppose that a new requirement is imposed on the workflow: At any instant in time, either both origin or both destination copies must exist. The workflow does not satisfy this new requirement because it may erase  $O1$  before the  $O2 \rightarrow D2$  copy completes. With discrete control, the new requirement can be satisfied simply by forbidding states that violate it and then synthesizing a new controller. The controller satisfies the new requirement by appropriately postponing erase operations.

Six states in Figure 3 are forbidden because they violate the new requirement. Control synthesis identifies six additional “unsafe” states from which a sequence of uncontrollable transitions can lead to a forbidden state. For example, an unsafe state results if erase- $O2$  and  $O1 \rightarrow D1$  are in progress simultaneously, because an uncontrollable event (the completion of the former) can lead to a forbidden state. Discrete control synthesis yields a controller that avoids both unsafe and forbidden states by disabling the start of erase operations where appropriate. This scenario shows that discrete control can accommodate new requirements without workflow maintenance.

## 5.2 Software Installation Workflow in BPEL

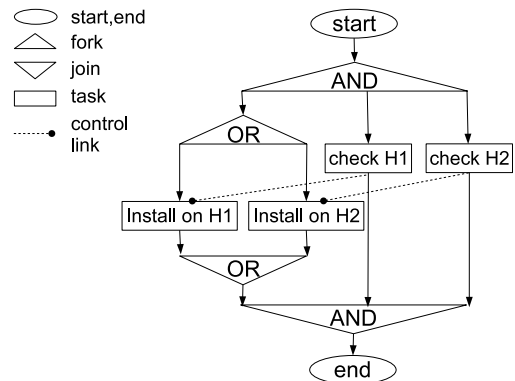
Suppose that the goal of a workflow is to install an application on one of two hosts. First we check resource availability on both hosts and pick one on which to install the application. We have no pref-



**Figure 4: IT installation workflow in BPEL.**

erence over the two hosts if both have sufficient resources, so the purpose of the workflow is to install the application as soon as one host reports availability. Figure 4 depicts one way to realize this workflow in BPEL. It is a standard BPEL architecture for selecting among multiple options. First we check availability on both machines concurrently using an AND structure, then the execution engine selects either one that is available to install the application. The AND structure requires the completion of both branches before it goes to the next step, which may delay the installation process even if one host has been checked successfully.

To achieve better performance, we redesign the above workflow as shown in Figure 5. The modified workflow increases parallelism by allowing the selection of hosts at the outset. It uses a special BPEL structure called a *control link* to guarantee that the application is installed only if the check task has completed successfully. With the new workflow design, if the availability check on the selected host succeeds, all goes well and the installation proceeds on the selected host. However if the selected host is unavailable but the other host is available, the installation task is skipped even though it would succeed on the unselected host.



**Figure 5: Modified IT installation workflow in BPEL. A “control link” is a special BPEL structure that defines dependency between two tasks. In the above example, “install on H1” must wait until “check H1” has been completed. If “check H1” is successful, “install on H1” will be executed, or skipped otherwise.**

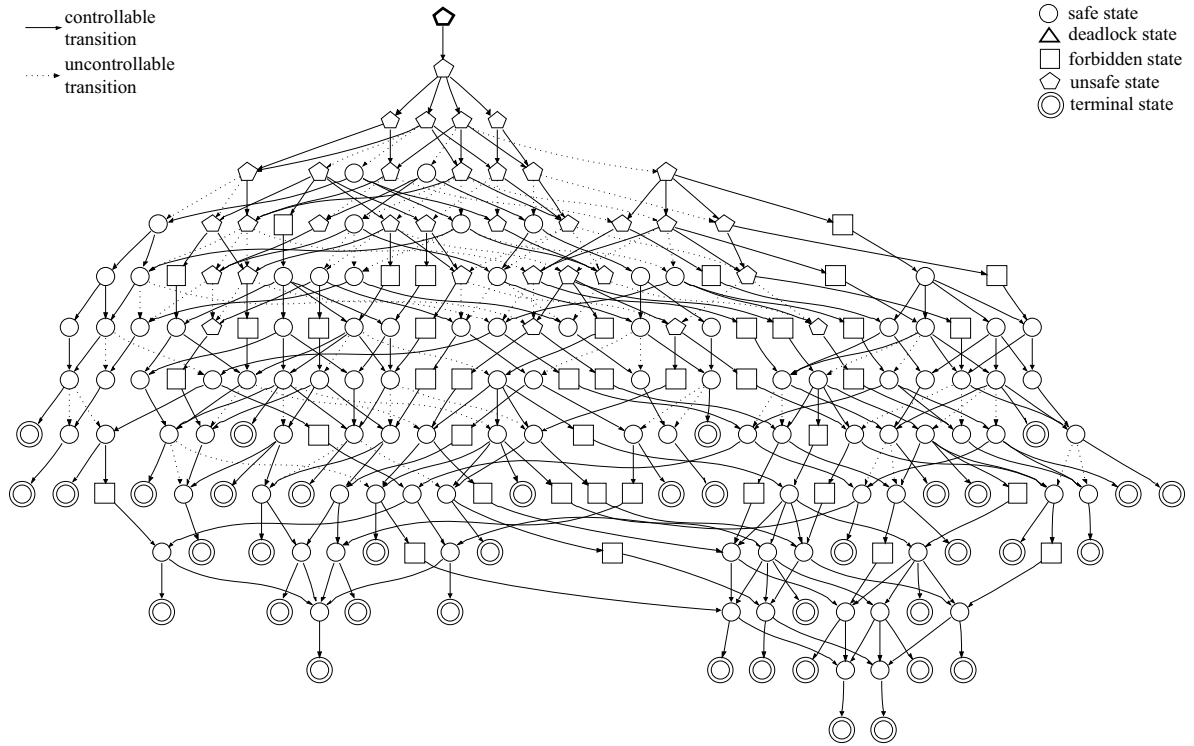


Figure 6: Automaton for workflow of Figure 5.

The problems associated with the two above designs stem from the limited task dependency allowed in the BPEL workflow language: There is a fixed partial order relationship among all tasks.

We can apply discrete control to properly execute at run-time the modified workflow in Figure 5 by specifying as forbidden those states where the installation task is skipped. The synthesized control logic is displayed in Figure 6. The controller must avoid forbidden states. As a result, both installation transitions in the OR fork are disabled from the beginning, because the controller foresees the danger of entering a forbidden state unavoidably if the transition is allowed and the selected host is unavailable. If one of the check tasks completes with a positive result, then the corresponding installation transition is allowed. By disabling potentially dangerous transitions at the appropriate time, the controller guarantees safe execution whenever possible.

Note that in this example the initial state itself is unsafe, i.e., it may lead to a forbidden state unavoidably. This is because it is possible that both hosts are unavailable and the installation simply cannot be performed. The controller, however, guarantees that installation will occur as long as one host is available. This demonstrates one advantage of online control: When it is not possible to program a workflow that always succeeds, discrete control can avoid dynamic failure where possible.

## 6. IMPLEMENTATION ISSUES

In this section we discuss implementation issues regarding workflow translation and control synthesis in the context of our workflow control architecture in Figure 1.

### 6.1 Oracle BPEL Workflows

As explained in Section 2, online control adds negligible constant overheads to workflow execution since the execution engine tracks the current state and enforces control actions by consulting a look-up table. On the other hand, the offline operations of translating workflows into automata and synthesizing the control logic are potentially expensive. No other computational obstacles surround our proposal. The only practical question is whether the state spaces of real workflows can be handled by discrete control synthesis algorithms. To understand the scalability issue we applied our control synthesizer to real BPEL workflows bundled with Oracle BPEL designer [25] and also to large randomly-generated workflows.

Most of the Oracle BPEL workflows automate IT aspects of business operations such as loan offer processing, wire transfers, and vacation request processing. Figure 7 shows an Oracle BPEL workflow implementing a loan application process. The workflow first receives an application as input, then it invokes a credit rating service to obtain the applicant's credit report. Once the credit report is ready, the process delivers the report to two loan service agencies in parallel to solicit loan offers. Finally, the workflow presents both offers to the client and completes the process after the client selects one. This type of workflow is generic to many IT automation tasks.

We used a research tool [26] to translate BPEL into Petri nets and then implemented standard reachability graph construction methods to obtain automata models of the workflows. Finally, we applied our discrete control synthesis algorithm to the automata. Of 164 Oracle workflows, five yielded malformed Petri nets due to errors in the translator. Nine others had excessively large state spaces, causing translation to automata to fail. Results of our scalability tests for the other 150 workflows are displayed in Figure 8.

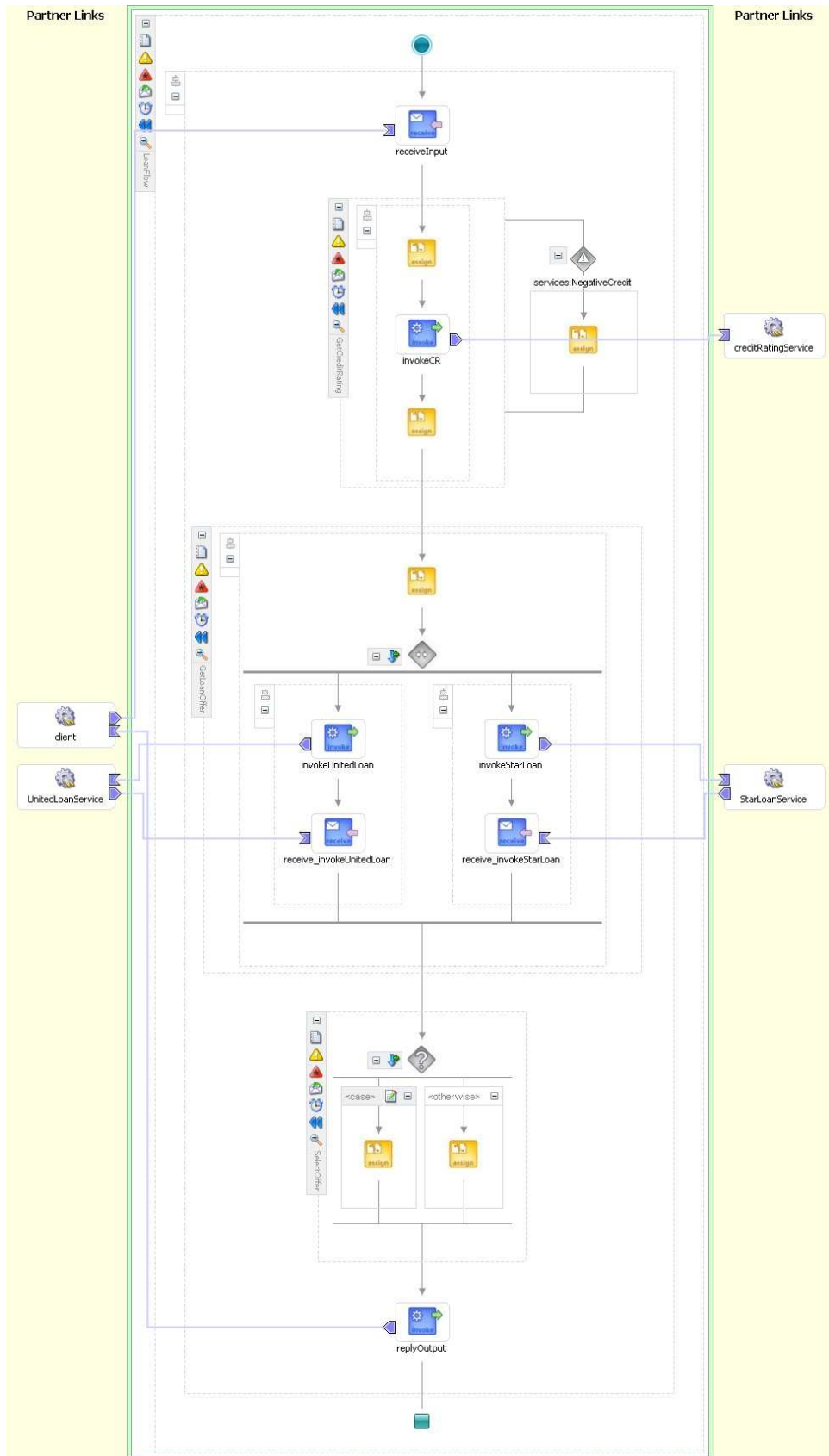
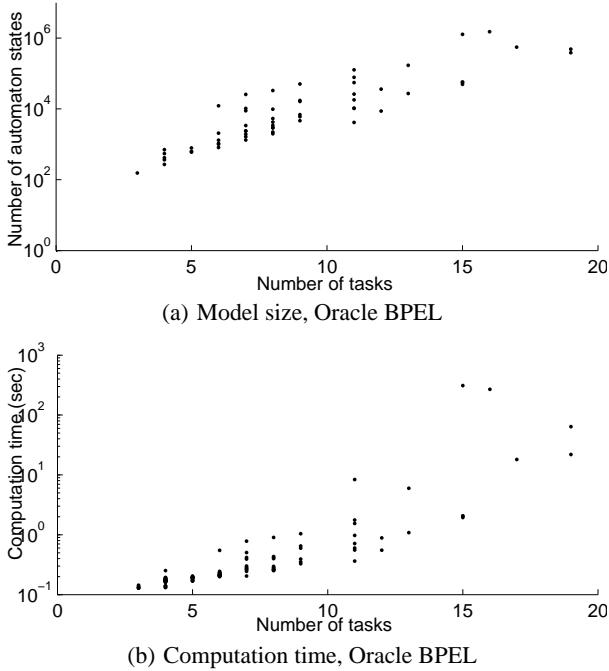


Figure 7: A BPEL workflow example consisting of 14 tasks and associated control structures.

Our implementation handles nearly all of the Oracle workflows quickly. The largest Oracle workflow has more than 20 tasks and is translated into an automaton of 1.5 million states in roughly 13 minutes on a SUN Ultra 20 (1.8 GHz processor, 2GB RAM). The computational bottleneck is translation from BPEL to automaton. We manually inspected several of the Petri nets generated by the translator we are currently using and found that unnecessarily lengthy and redundant structures are often present. We believe that direct translation from BPEL to finite automata, or translation using more concise intermediate Petri nets, is possible; this would reduce computation time.



**Figure 8: Control synthesis results for BPEL workflows.**

## 6.2 Random Workflow Experiments

The Oracle BPEL workflows are the largest collection of commercial workflows we could find, but they are somewhat limited in size and variety. Therefore we also generated random workflows using a simple probabilistic context-free grammar. The generator starts with one task in the workflow. At each iteration, it randomly picks a task and expands it using four basic structures—sequence, AND-fork, OR-fork and while loop—with equal probability. The process stops when enough tasks have been generated. The random workflows are represented in succinct Petri net format. They are translated into automata using the standard translation algorithm. We also introduced two shared resource units in the random workflows to create deadlocks and livelocks. Then we applied our control synthesis algorithm to try to find safe execution paths.

Figure 9 displays results of random workflows with 1 to 50 tasks, with 15 workflows generated for each number of tasks. Of 750 random workflows, 21 had excessively large state spaces. Figure 9(a) shows that automaton size is worst case exponential in the number of tasks, but the translated automata are typically small. To test our control synthesis algorithm, we also included one resource shared among some tasks in each workflow. As a result, on average 7% of the automata states are deadlock or livelock states. The control

synthesis algorithm then calculates the maximally permissive non-blocking controllable sublanguage associated with each automaton.

The combined computation time is displayed in Figure 9(b). We randomly picked 35 workflows with computation time raging from 2 second to 15 seconds; the detailed computation times of both translation and control synthesis are displayed in Figure 9(c). As can be seen, in our current implementation, translation to automata dominates offline control synthesis; the latter accounts for under 10% of the total offline computational cost on average.

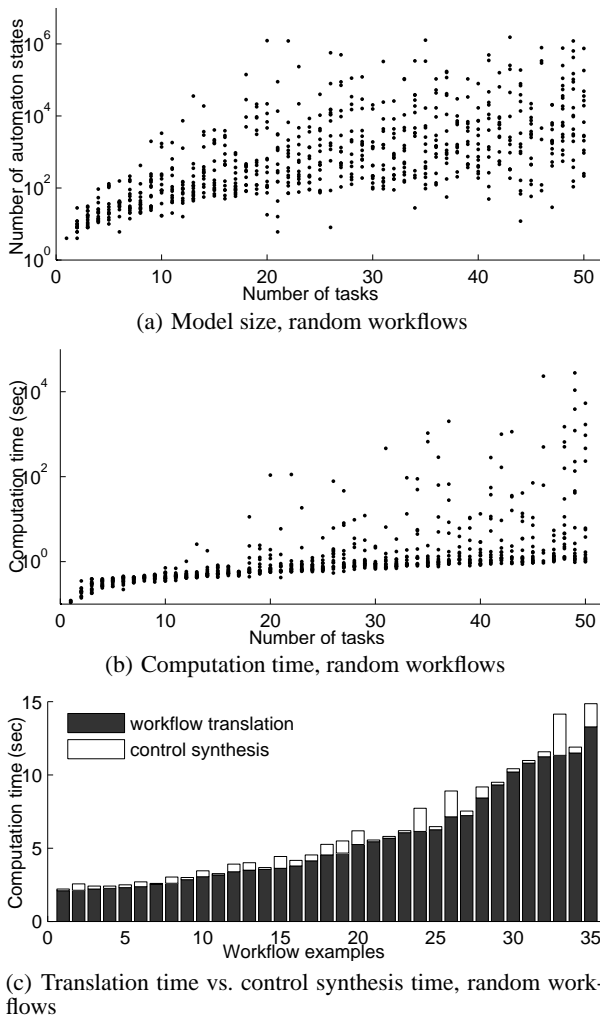
Production workflows are typically considerably smaller than our random workflows in terms of the numbers of tasks they include. A detailed analysis of over 9,000 workflows that implement real production business processes in the SAP Reference Model found that the average number of tasks is under two dozen [21]. However this is a misleading measure of workflow complexity because, as the examples of Section 5 show, concurrency can confound understanding even for very small workflows. Field experience bears this point out: The same study of SAP workflows found that *at least 5.6% of these fully-debugged, business-critical production workflows contained statically-detectable defects*. Our methods are both valuable and feasible for workflows with state spaces large enough to overwhelm human analysis yet small enough to admit control synthesis. Our personal experience and detailed investigations by other researchers convince us that the vast majority of real-world workflows fit this description.

## 7. DISCUSSION

We have argued that online dynamic control of IT automation workflows is a useful complement to existing dependability techniques. We have described how discrete control methods can synthesize controllers from workflows and declarative specifications. These controllers add negligible run-time overhead, and they prevent undesirable behavior while otherwise restricting execution as little as possible. Our approach reduces costs and increases dependability by allowing flawed workflows to be executed safely. It partially decouples workflows from requirements, reducing the need for maintenance programming when requirements change. Our ongoing work integrates our discrete control module into a real workflow execution engine.

The discrete control methods that we employ in our current work perform offline pre-computations based upon explicit representations of workflow state spaces, and scalability is a potential concern for these offline operations. Experience with real production workflows, however, convinces us that the features that elicit worst-case state spaces are contrived pathologies and are not typical of workflows in the wild. Our performance tests on a large collection of commercial workflows and on a large and diverse set of randomly-generated workflows show that our discrete control logic synthesis implementation scales to workflows of practical size. Although at present we have no pressing need to implement them, there exist extensions to discrete control techniques that could be employed to accommodate state spaces too large to be represented explicitly. These extensions include symbolic methods for state space reduction [20], decomposition [36], and limited look-ahead [4] techniques.

In our current implementation, we assume full observability, i.e., the controller knows exactly the current system state. Partial observability may exist when the system is distributed or the program simply does not retrieve enough information to identify the system



**Figure 9: Control synthesis results for randomly generated workflows.**

state. In principle, the techniques briefly described in Section 2.2 address the problem of safe execution under partial observability as well as partial controllability. However, it is an open question whether these extended techniques are sufficiently scalable to the problem of safe execution of workflows in practice.

This paper discussed the problem of avoiding forbidden states in order to achieve safety guarantees. There is a dual problem in discrete control theory of reaching a desired state in the presence of uncontrollable state transitions and optimality criteria (e.g., reaching a terminal state as quickly as possible). We leave the investigation of this dual problem for future work.

Finally, we believe that discrete control methods could be applied to a wide range of dynamic failure avoidance problems in computing systems. We started with workflow systems because of their high-level nature, simple structure, and relatively small state spaces. Encouraged by our experiences in the workflow domain, we are investigating applications of discrete control theory to a wider range of problems.

## Acknowledgements

Sharad Singhal, Sven Graupner, and Peter Chen provided useful suggestions when we started the project. We thank Arif Merchant, Kimberly Keeton and Brian Noble for comments that helped us improve the paper. We also thank the reviewers for their useful comments and suggestions.

## 8. REFERENCES

- [1] J. Albrecht, C. Tuttle, A. C. Snoeren, and A. Vahdat. Planetlab application management using push. *SIGOPS Oper. Syst. Rev.*, 40(1):33–40, Jan. 2006.
- [2] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *Proc. EuroSys*, Apr. 2006.
- [3] Y. Bar-David and G. Taubenfeld. Automatic discovery of mutual exclusion algorithms. In *Proc. 17th Int'l Sympos. Dist. Comput (LNCS 2648)*, pages 136–150, Oct. 2003.
- [4] N. Ben Hadj-Alouane, S. Lafortune, and F. Lin. Variable lookahead supervisory control with state information. *IEEE Trans. on Automatic Control*, 39(12):2398–2410, Dec. 1994.
- [5] B. A. Brandin. The real-time supervisory control of an experimental manufacturing cell. *IEEE Trans. on Robotics & Automation*, 12(1):1–14, Feb. 1996.
- [6] A. B. Brown and J. L. Hellerstein. Reducing the cost of IT operations—is automation always the answer? In *HotOS*, June 2005.
- [7] Y.-L. Chen, S. Lafortune, and F. Lin. Resolving feature interactions using modular supervisory control with priorities. In *Feature Interactions in Telecom. Networks IV*, pages 108–122. IOS Press, 1997.
- [8] R. Cieslak, C. Desclaux, A. Fawaz, and P. Varaiya. Supervisory control of discrete-event processes with partial observations. *IEEE Trans. on Automatic Control*, 33(3):249–260, Mar. 1988.
- [9] E. W. Dijkstra. *Selected Writings on Computing*, chapter The Mathematics Behind the Banker's Algorithm, pages 308–312. Springer-Verlag, 1982.
- [10] E. Eide, L. Stoller, T. Stack, J. Freire, and J. Lepreau. Integrated scientific workflow management for the emulab network testbed. In *USENIX Annual Technical Conference*, Dec. 2006.
- [11] K. Farkas, S. Iyer, V. Machiraju, J. Pruyne, and A. Sahai. Automated provisioning of shared services. In *Proceedings of the 10th IFIP/IEEE Symposium on Integrated Management*, May 2007.
- [12] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. Wiley, 2004.
- [13] jBPM. <http://www.jbpm.com/products/jbpm>.
- [14] K. Keeton. Personal communication.
- [15] K. Keeton, D. Beyer, E. Brau, A. Merchant, C. Santos, and A. Zhang. On the road to recovery: Restoring data after disasters. In *Proc. EuroSys*, Apr. 2006.
- [16] E. Kiciman and L. Subramanian. A root cause localization model for large scale systems. In *HotDep*, June 2005.
- [17] B. Kiepuszewski, A. ter Hofstede, and W. van der Aalst. Fundamentals of control flow in workflows. *Acta Informatica*, 39(3):143–209, 2003.
- [18] C. Killian, J. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems

- code. Technical report, UC San Diego, 2006.  
[http://mace.ucsd.edu/papers/MaceMC\\_TR.pdf](http://mace.ucsd.edu/papers/MaceMC_TR.pdf).
- [19] F. Lin and W. M. Wonham. On observability of discrete-event systems. *Information Sciences*, 44(3):173–198, 1988.
- [20] H. Marchand and S. Pinchinat. Supervisory control problem using symbolic bisimulation techniques. In *American Control Conference*, pages 4067–4071, June 2000.
- [21] J. Mendling, M. Moser, G. Neumann, H. Verbeek, B. van Dongen, and W. van der Aalst. A quantitative analysis of faulty EPCs in the SAP reference model. Technical Report BPM-06-08, Business Process Management Center, 2006.  
<http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/reports/2006/BPM-06-08.pdf>.
- [22] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr. 1989.
- [23] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and dealing with operator mistakes in Internet services. In *Proc. OSDI*, Dec. 2004.
- [24] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do Internet services fail, and what can be done about it? In *Proc. USITS*, Mar. 2003.
- [25] Oracle BPEL workflows. <http://www.oracle.com/technology/products/ias/bpel/>.
- [26] C. Ouyang, E. Verbeek, W. M. P. van der Aalst, S. Breutel, M. Dumas, and A. H. M. ter Hofstede. WofbpeL: A tool for automated analysis of BPEL processes. In *ICSOC*, pages 484–489, Dec. 2005.
- [27] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies—a safe method to survive software failure. In *Proc. SOSP*, Oct. 2005.
- [28] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1):206–230, 1987.
- [29] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and J. William S. Beebe. Enhancing server availability and security through failure-oblivious computing. In *Proc. OSDI*, Dec. 2004.
- [30] M. Sampath. A hybrid approach to failure diagnosis of industrial systems. In *American Control Conference*, pages 2077–2082, June 2001.
- [31] M. Sampath, R. Sengupta, K. S. S. Lafortune, and D. Teneketzis. Diagnosability of discrete event systems. *IEEE Trans. on Automatic Control*, 40(9):1555–1575, Sept. 1995.
- [32] Secure programming lint. <http://www.splint.org/>.
- [33] Sun. *WorkShop: Command-Line Utilities*, chapter 24: Using Lock Lint. Sun Press, 2006.  
<http://docs.sun.com/app/docs/doc/802-5763/>.
- [34] W. van der Aalst and A. ter Hofstede. Verification of workflow task structures: A petri-net-based approach. *Information Systems*, 25(1):43–69, 2000.
- [35] Y. Wang, T.-S. Yoo, and S. Lafortune. Diagnosis of discrete event systems using decentralized architectures. *Discrete Event Dynamic Systems*, 17(1), 2007.
- [36] W. M. Wonham and P. J. Ramadge. Modular supervisory control of discrete event systems. *Mathematics of Control of Discrete Event Systems*, 1(1):13–30, 1988.
- [37] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *Proc. OSDI*, Dec. 2004.