A control theory based approach for self-healing of un-handled runtime exceptions

Benoit Gaudin, Emil Vassev, Mike Hinchey, Paddy Nixon

A Control Theory Based Approach for Self-Healing of Un-handled Runtime Exceptions

Benoit Gaudin[1], Emil I. Vassev[1], Michael G. Hinchey[1] and Patrick Nixon[2]

1. Lero - The Irish Software Engineering Research Center, University of Limerick, Ireland
firstName.lastName@lero.ie

2. University of Tasmania, Hobart, Australia
Paddy.Nixon@utas.edu.au

ABSTRACT

This work presents an approach to self-healing that deals with un-handled exceptions within an executing program. More precisely, we propose an approach based on control theory that automatically disables system functionalities that have led to runtime exceptions. This approach requires the system to be instrumented prior to deployment so that it can later interact with a supervisor. This supervisor encodes the only sequences of actions (method calls) of the system that are permitted. We describe an implementation that automatically generates instrumentation for Java systems. We introduce an extension of Supervisory Control theory that enables automatic computation of a supervisor/controller model ensuring that an observed trace leading to an un-handled runtime exception cannot occur anymore. We demonstrate the efficacy of this approach through a comprehensive example.

Keywords

Self-Healing, Software Control, Software Maintenance, Supervisory Control Theory.

1. INTRODUCTION

This work deals with software self-healing in order to automatically generate and apply patches when facing runtime faults. More specifically, we consider legacy systems and automatically provide them with self-healing capabilities that allow for handling of runtime exceptions. We assume that the systems under consideration went through the different life cycle phases (design, implementation, testing, deployment). Typically, possible faults such as IO and NullPointer exceptions are not all detected during the testing phase, and remain in the system. These faults are usually reported by the user whenever their corresponding symptoms are observed at runtime. Faults related to exceptions may occur because of an inappropriate implementation as explained in e.g. [18]. For instance, a lack of checking in the way users may use the application can lead to such exceptions, e.g. Format Exceptions may occur from wrong user inputs into fields.

This work is part of the EU FP7 FastFIX project [1]. FastFIX involves several companies and tackles problems of industrial relevance. This project aims to develop approaches and techniques in order to identify failure symptoms, changes in user behavior as well as to perform failure replication, patch generation and patch deployment. Autonomic computing principles (see e.g. [13]) are considered in order to achieve these goals.

However, as explained in [10], although Autonomic Computing (AC) has yielded to many impressive achievement, it has not yet accomplished some of its very desired goals. Some of the AC techniques related to control theories for instance have proven to be very successful for power management but their applicability to other types of systems such as general software systems remain unclear. Moreover, introducing autonomic features to a system by implementing feedback loops requires careful design to ensure that the system does not diverge from its desired goals. In this work, we describe and implement an approach to control software system, hence providing it with autonomic features. With this capabilities the system can automatically adapt to avoid further occurrences of observed runtime exceptions. Moreover the design of such control approach is itself automated from the source code of the software system under consideration, avoiding error that could be introduced by manual design.

Other techniques than control have been considered in the past for the self-healing of software systems. As pointed out in [8], self-healing is historically very much related to fault-tolerance. Moreover, fault-tolerance approaches mostly rely on resource redundancy (see e.g. [21]) and has led to similar approaches for self-healing (hardware redundancy, software variants, etc). Other works on self-healing such as [19] rely on similarities between faults and solutions previously applied. More recently, the authors of [5] have considered the use of workarounds in order to automatically implement faulty functionality using different part of the system (e.g. changing an item in an online shopping basket is equivalent to deleting it then adding the desired one).

In this work, our healing approach does not rely on system redundancy or on previously applied solution. However it has a similar view to [5] in the sense that it prevents the execution of faulty behaviors, assuming that the intended

goal can be achieved a different way or that the system was used in an appropriate manner.

The approach that we introduce aims to automatically provide legacy systems with self-healing capabilities. In this work, Self-healing is performed through monitoring, exception catching and control.

The sequences of method calls of the system are monitored during execution and are matched to a model of the possible behaviors of the system. Runtime exceptions are also assimilated to method calls and are not taken into account by this model. When such an exception occurs for the first time, the sequence of method calls containing this exception does not match the system model and the healing process is triggered. This process consists of synthesizing a new model of the system that ensures that futur system executions cannot lead to this exception. This new model is saved into a file that can be seen as a patch and is loaded from this file when the system restarts. The system executions are then controlled according to this model so that sequences of method calls that lead to the previously detected runtime exception are prevented.

Therefore, our approach relies on method calls and exception monitoring as well as system control. In this paper we describe this approach and how Java programs can be automatically instrumented in order to provide them with monitoring and control facilities. This is achieved using the Javassist library [2]. The control mechanism relies on a model which is automatically synthesized form a previous model of the system and a previously observed sequence of method calls leading to an exception. Such synthesis is performed based on Supervisory Control theory on Discrete Event Systems [22].

Previous works on software control have been investigated such as [14] which provide general ideas on software control, in [6, 4] network communications and self-management requirements are controlled in order to ensure good performance and system policies are controlled in order to avoid conflicts (e.g. [17]). However, an important aspect of controlling systems is to ensure their stability, i.e. demonstrating that the control achieves what the designer have intended. As explained in [20], this is of great importance whenever system self-management is considered, where the system is left running without human intervention. [9] goes further and states that the lack of understanding of how automated actions affect system behavior is seen as one of the main reasons why automatic approaches are not more used.

In this paper, we consider the Supervisory Control on Discrete Event Systems. This theory considers models of the behaviors of the system and intended behaviors (called *control objective*) and to our knowledge, it has never been applied to software systems before. It applies to formal models and provides algorithms for automatic computation of the behaviors of the system under control. Moreover, the control applied do not create new behaviors but restrict the initial behaviors of the system instead, contributing to a stable solution, i.e. no undesired behaviors emerge. Finally models under consideration are represented with Finite State Machines. As our approach deals with the behaviors of the system (i.e. sequences of method calls) rather than with its variables. This makes automatic model extraction and synthesis scalable even for large application. We are currently able to extract finite state machines from the source code of the JEdit application [3] for all of its 6500 methods in about 3 minutes.

This paper contributes to several aspects related to the self-healing of legacy Java programs and supervisory control theory. It presents a control based approach for automatic introduction of self-healing features into existing Java program. It also describes an implementation of the self-healing mechanism that relies on the synthesis of a model representing correct system behaviors and that can be seen as a system patch. Finally, this paper demonstrates that this approach is theoretically sound.

In Section 2, we present some background on Autonomic Computing and Supervisory Control Theory for Discrete Event System initiated by [16] through examples. Section 3 describes our approach to automatically apply this theory to Java programs, discussing the automation of model extraction and detailing the implementation for automatic code instrumentation. In Section 4, we extend the supervisory Control theory in order to handle the case of system traces that lead to runtime exceptions. Then Finally, our approach is illustrated on a calculator example.

2. BACKGROUND

The approach described in this paper relies on self-healing principles and control theory. Some background on both these topics are presented in Sections 2.1 and 2.2.

2.1 Autonomic Computing and Self-Healing

Self-adaptive systems possess the ability to adapt to changes or situations. In particular, they can modify their behavior in order to optimize or repair themselves.

Software systems are becoming increasingly complex, thus leading to increased overhead of maintenance and support. These problems motivate the need of autonomic software paradigm with so called self-* properties such as self-healing, self-configuring, self-managing, self-optimizing and so on.

To achieve the so called self-* capability the system needs to continuously monitor its execution environment, input parameters and produced output as well as detect requirements violations.

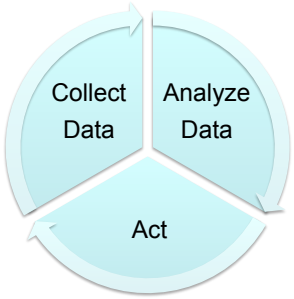


Figure 1: The autonomic feedback loop.

The different steps involved in self-* properties are usually presented as the autonomic feedback loop as presented in Figure 1. An autonomic system must be able to collect information about itself and its environment, analyse the collected data with respect to some knowledge about its behaviors. From this analysis, decisions can be made regarding whether the current observations are satisfying or whether some actions should be performed in order to ensure proper behaviors of the system.

The analysis and decision part can be implemented in separate module from the system itself, that must then be able to observe and act onto the system. The system entry points that allow for observation and action are respectively called *sensors* and *actuators*. In our approach these points are implemented through program instrumentation and the analysis is based on control theory. The *analysis* phase is actually twofold. If the current observation corresponds to some method calls, then the analysis phase is performed by a supervisor which decides whether this method should be executed in order to prevent the occurrence of previously observed un-handled runtime exceptions. If the current observation corresponds to the occurrence of an un-handled exception, then the analysis phase consists of automatically computing a new model of the supervisor that will be able to prevent the observed exception to occur again. The latter case rely on the Supervisory Control theory on Discrete Event Systems, for which basics are presented in Section2.2.

2.2 Supervisory Control of Discrete Event Systems

As systems have become more and more complex, making sure that their behaviors fulfill given requirements is an important challenge. Although testing and verification have proven to be extremely useful, some faults usually still remains in the system and are only discovered at runtime. In order to deal with this issue in the case of runtime exceptions, we propose to control the system at runtime, using a supervisor which interacts with the system in order to prevent it from executing paths leading to these exceptions. Figure 2 represents a feedback control architecture, where the system is monitored by a supervisor which can prevent some behaviors of the system from occurring.

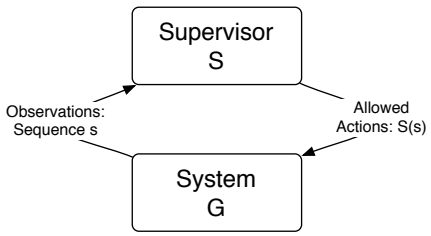


Figure 2: The Control Feedback Loop.

Moreover, considering the increasing system complexity again, implementing such mechanisms and designing a supervisor that achieve the desired objective are very challenging tasks.

Supervisory Control on Discrete Event Systems ([16]) is a formal theory that aims to automatically design a model for a supervisor ensuring some safety property. The Supervisory Control Theory defines notions and techniques that allow for existence and automatic computation of a model of the supervisor, given a model of the system as well as the property to be ensured. In this theory, models of a system G are represented by languages over alphabets of events, denoted $L(G)$. These languages correspond to sets of sequences of events, each representing a possible behavior/execution of the system.

Although not as general as languages, Finite State Machine (FSM) are used to model the possible behaviors of the system as well as the supervisor and the properties to be ensured by control. Regarding the modeling of supervisors, Figure 2 shows that they can be seen as a function that takes a given sequence s and returns to the system a set of allowed events after s. The function S representing the supervisor can be encoded by a FSM G_S such that for all $s \in L(\mathrm{S})$, $S(s)$ represents the set of events that can be triggered from the state reached in G_S after sequence s.

Supervisors ensure a given property, called *control objective*. Such a property is modeled as a FSM as well, generating a set of "safe" behaviors and meaning that the behaviors that are not encoded by this FSM are undesired.

The main goal of the Supervisory Control theory is to automatically synthesizes a model of a supervisor that ensures that the system behaviors are all included in the ones described by the control objective. The theory also considers that not every event can or should be disabled by a supervisor. Such events are said to be uncontrollable. Events corresponding to some sensor reading or the tic of a clock are typically uncontrollable.

In order to take such events into account, the alphabet of the system is assumed to be composed of a set of *controllable* events ($A_c \subseteq A$) and *uncontrollable* events ($A_u \subseteq A$). Each event of the system is either controllable or uncontrollable.

Controlling a system consists of restricting its possible behaviors taking into account the controllable nature of the system events. In order to achieve this, Ramadge and Wonham (see e.g. [22]) introduce a property called *Controllability*. A system G' whose behaviors correspond to a subset of the ones of G is controllable w.r.t A_u and G if $L(G').A_u \cap L(G) \subseteq L(G')$.

A controllable set of behaviors G' ensures that no sequence of uncontrollable events can complete a sequence of G' into a sequence of G that is no longer in G'. In other words, the controllability condition ensures the stability of G' through uncontrollable sequences. This condition must hold for the behaviors generated by any supervisor.

We now define the basic supervisory control problem, which can be stated as the following:

Basic Supervisory Control Problem (BSCP): Given a system G and a control objective K, compute the maximal controllable set of behaviors included in the ones of both G and K.

Ramadge and Wonham (see e.g. [22]) have shown that a solution to the BSCP exists if and only if the maximal controllable set of behaviors included in the ones of both G and K is not empty. They also provided an algorithm computing this FSM which encodes a most permissive supervisor ensuring the control objective (see e.g. [22]). This algorithm can be seen as a function that takes as inputs a set of uncontrollable events A_u, a FSM representing the control objective K and a FSM representing the behaviors of the system G. We denote this function SupCont and therefore $\mathrm{SupCont}(A_u, K, G)$ represents a solution of the BSCP.

The complexity of this algorithm is linear in the number of events, the number of states of the model of G and the number of states of K. We now illustrate the concepts introduced in this section with Example 1.

Example 1. *This example considers a system which can perform 2 actions: action 1 and action 2. The set of possible behaviors of the system is described in Figure 3. From its*

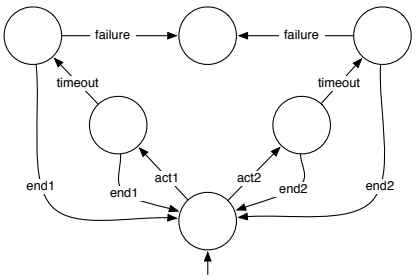


Figure 3: An FSM G modeling a system which can perform 2 actions during which failures can occur.

initial state, the system can perform either action 1 or action 2. If action 1 is performed, then the system enters a state where event end1 can be triggered. This corresponds to a proper termination of action 1, leading the system back into its initial state where a choice between executing action 1 or action 2 can be made again. In some cases, a timeout event may occur after event act1, indicating that the time usually required to complete action 1 has elapsed. This does not represent a fault but indicates that the system is not evolving in an optimal manner. If a timeout is observed, action 1 may still be ended, which is represented by the occurrence of event end1. However, after a timeout has occurred, a failure can also occur, leading the system into a deadlock state.

As this system is symmetrical, similar behaviors can be executed if action 2 is performed.

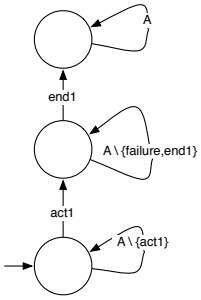


Figure 4: A control objective K stating that a failure must not occur the first time action 1 is being performed.

Figure 4 provides an example of control objective. This FSM states that once the system has started, any event can be observed from it, however, the first time act1 is observed, end1 must be observed before a failure occurs. Once end1 is observed, then any event can be observed again. Such a control objective actually aims to enforce that a failure does not happen the first time action 1 is being performed.

Given FSMs of both the system and the control objective, Supervisory Control theory provides techniques for computing a model of a supervisor that enforces the control objective, taking into account that some events are not controllable. In this example, it is assumed that events 'failure' and 'timeout' are uncontrollable. Figure 5 represents the most permissive supervisor ensuring the control objective presented in Figure 4, i.e. no larger controllable behavior can be prevented by control while still ensuring it.

This supervisor actually states that in order to prevent a failure from occurring the first time action 1 is being performed, there is no other alternative but to prevent action 1 to be performed at all. Therefore, applying SupCont in order to fulfill the control objective, shows that the system must be downgraded to only be able to perform action 2.

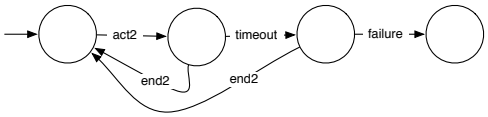


Figure 5: An FSM modeling a supervisor for system G and control objective K.

In Example 1, the model of the system encodes possible failures. However, this is unfortunately not always possible, especially for software system. In Section 4, we will show how we can use the definitions and results of this section in the case where the model of the system is not complete and the control objective is derived from a sequence whose last event corresponds to a runtime exception.

3. INTRODUCING SELF-HEALING CAPABILITIES FOR UN-HANDLED JAVA EXCEPTIONS

3.1 General Approach

In the course of this research, we apply Supervisory Control (see Section 2.2) to provide systems with self-healing capabilities. Basically, our approach aims to automatically generate a supervisor system that coexists with the original system and drives it in order to avoid critical situations. Currently, our approach works on Java applications only with full access to the source code and consists of two main stages: preparation and control.

In the preparation stage, the source code is automatically analysed and a FSM of the behaviors of the Java application is built. This FSM initially also models a supervisor behaviors and covers all the execution paths of the targeted application as long as no un-handled runtime exception has been discovered. Further, the supervisor code is automatically generated and integrated in the original code which is also instrumented to allow for the supervisory control at runtime (see Figure 6). Therefore, we deploy and execute the new instrumented application embedding a supervisor.

In the control stage, the embedded supervisor runs non-intrusively in respect to the functionality provided by the application. While running, the supervisor follows the cycle shown in Figure 1 where the three phases are as following:

- Collect Data: The supervisor monitors the method calls and exceptions occurrences.
- Analyse Data: Before the execution of each method, the collected data is analysed by considering the generated FSM. If the method call in question is not authorized by the FSM model, then the method execution is prevented. Otherwise, the method executes normally. If an un-handled runtime exception occurs during the execution of that method, then a new supervisor FSM

is synthesized in order to handle future faulty method calls.

- Act: The supervisor enters in this phase if special actions are required. Such actions are performed when an unauthorized method call is attempted or an unhandled exception has arisen. Thus, in the first case, the supervisor prevents the execution of an unauthorized method, and in the second case, supervisor FSM is synthesized to help the supervisor prevent future execution of such a sequence of method calls leading to the unhandled exception.

  In order for the system to fully benefit from this new supervisor, it usually needs to be restarted. A new supervisor can be seen as a patch for the system and the user is advised to restart the system in order to take this patch into account.

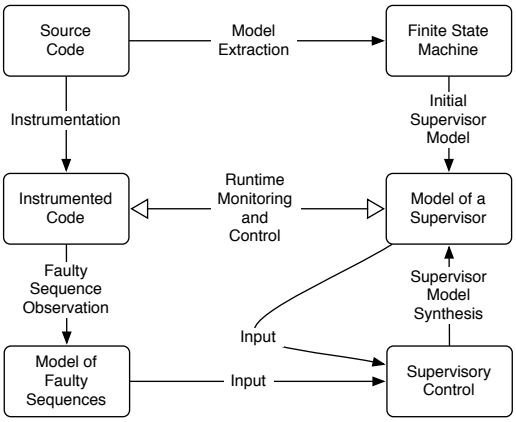


Figure 6: Approach for automatic control of software

Figure 6 depicts our approach. The application source code is used to build the Supervisory Control FSM covering all the execution paths in terms of method calls. Moreover, the source code is automatically instrumented to integrate the supervisory control features (sensors and actuators). At runtime, the supervisor embedded in the application monitors and controls the program execution by consulting the FSM. Whenever an un-handled runtime exception has arisen, it is caught through instrumentation and a model of the faulty sequence of method calls is automatically derived from it. Further, Supervisory Control Theory is applied in order to synthesize a model of a supervisor.

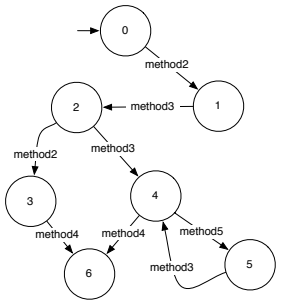
3.2 Model Extraction

In this paper, we consider Finite State Machines (FSM) to model the behavior of the system as well as the properties to be ensured by control. The model of the system is 1) automatically modified in order to ensure a given property and 2) the obtained new model represents a supervisor that is used to monitor and control the system at runtime as illustrated in Figure 2.

Considering nowadays systems complexity, manually building Finite State Machines that represent the system behavior is tedious and error-prone. Therefore, approaches for automatically extracting FSM from the system source code

```
void method1()
{
method2();
if (method3())
            {
            method2();
            }
else
            {
            while (not(method3()))
                    {
                    method5();
                    }
            }
method4();
}
```

(a) An example of method declaration.



(b) An FSM modeling the system behaviors.

Figure 7: Illustration of FSM extraction.

have been considered. For instance, Bandera (see e.g. [7]) allows for model extraction from Java programs.

These Finite State Machines are actually more complex than the ones described in Section 2 as they consider program variables. The different states of the FSM correspond to different values of the variables of the system. These FSM are therefore state oriented and their number of states depends on the possible value range of the system variables.

More recently, the authors of [12] considered an approach to extract FSM from several programming languages such as Java and C. Their approach is behavior oriented and considers method calls rather than system variables. The resulting extraction process is lightweight and the size of the extracted FSM remains reasonable for analysis. In this work , we take a similar approach, however our models need to be complete in terms of possible observations that can be made at runtime. This is due to the fact that the model of the supervisor is used for control, which requires that every possible observations of the system made at runtime are encoded in the supervisor.

In order to illustrate our approach, Figure 7 presents some code sample and the corresponding FSM extracted from this piece of code. The method calls are extracted and correspond to the edges of the generated FSM. Branching (e.g. IF, SWITCH statements, etc) and loops (e.g. FOR and WHILE statements, etc) are also taken into account.

We have implemented an Eclipse plugin for FSM extraction from Java programs. So far, our implementation follows the Java 1.6 specification regarding loops and branching. However, it does not yet take into account concurrency introduced by threading and graphical components. Although our current implementation does not provide a model for multi-threaded and graphical applications, it allows for extraction of a FSM for each of their method, assuming it is single threaded. This was applied to JEdit [3] and shows that the approach scales to application with more than 6500 methods.

Finally, although concurrency can introduce some state explosion when combining FSM representing methods running in parallel, we plan on applying supervisory control techniques developed in [11], which make it possible to avoid this issue.

3.3 Instrumentation

As illustrated in Figure 2, our approach relies on the use

of a supervisor that can observe the behavior of the system and after each observed sequence provides a set of allowed events. In this section, we illustrate how this mechanism can be implemented through the use of code instrumentation. First of all, as described in Section 2.2, the supervisor can be seen as a function that, given a sequence of events s, returns the set of allowed events after this sequence. Implementing such a method is straightforward whenever a FSM of the supervisor is available. If q represents the current state of the FSM reached after sequence s, then the set of allowed events after s corresponds to the events that can be triggered from q.

Updating the current state q of the model of the supervisor requires to monitor the application. Moreover, preventing method bodys to be executed requires to act on the system execution. Finally as described in Section 3.1, runtime exceptions must be caught and trigger the synthesis of a new model of the supervisor. Catching such exceptions can also be done through instrumentation.

Our instrumentation process relies on the Javassist library [2] and is illustrated in Figure 8.

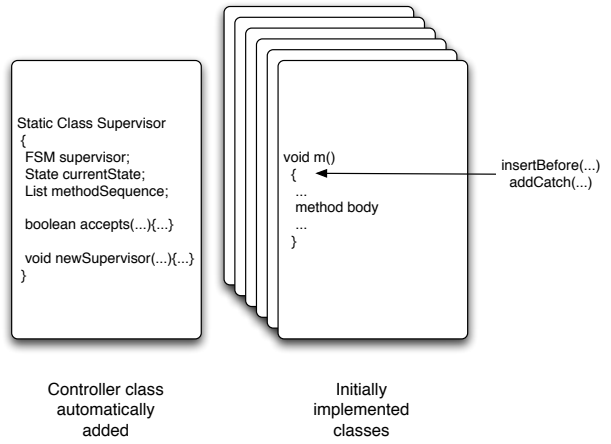


Figure 8: The instrumentation process.

Javassist offers many facilities to instrument Bytecode among which *insertBefore* and *addCatch*. Given a method m of a Java program, the *insertBefore* method allows to insert Java code that will be executed before the body of m and the *addCatch* method allows to catch runtime exceptions that are not already handled by method m.

Javassist also makes it possible to add new .class files to some existing program and to refer to the corresponding classes in the code inserted through the *insertBefore* and *addCatch* methods. In this work, we use this functionality to introduce a new Supervisor class. This class is declared as static and possess three attributes: an FSM representing the model of the supervisor, a state representing the current state of this model, and a sequence of observed method calls. This FSM is instantiated from a file containing the model of the supervisor and the current state can be instantiated as the initial state of this FSM. The Supervisor class also contains a static method called *Supervisor.accepts* which takes a string representing a method name (e.g. m) and returns a boolean. The value of this boolean is true if and only if the model of the supervisor encodes that m can be triggered from the current state of the model. Whenever the execution of m is authorized, *Supervisor .accepts* also updates the current state of the model as well as the sequence of observed method calls.

Therefore, implementing the monitoring of method calls and updating at runtime the model of the current state of the model of supervisor accordingly can simply be achieved by instrumenting each method m with:

$$insertBefore(\text{"}Supervisor.accepts(m);\text{"}) \quad (1)$$

With this approach whenever an authorized method is called at runtime, it is indicated to the *Supervisor* class which can then update the current state of the model.

The inserted code can also be augmented so that the body of unauthorized method calls are not executed. This should only be done for controllable methods while uncontrollable methods will be instrumented as described in Statement (1). In this work, we assume that only methods that do not return any value can be controllable and are instrumented using Javassist as follows:

$$insertBefore(\text{"If (!}Supervisor.accepts(m)\text{) return;"}) \quad (2)$$

Statement (2) indicates that whenever a method m is called, it is first checked if calling this method from the current state of the supervisor model is authorized. If it is the case, then the body of the method is executed normally and the current state is updated. If the method call is not authorized then the method exits before its body is executed.

Method *Supervisor.accepts* and the instrumentation presented in Statements (1) and (2) make it possible for a supervisor modeled by an FSM to control a Java program as illustrated in Figure 2.

However, more instrumentation is necessary in order for the overall system to be able to capture runtime exceptions and then trigger the healing process, i.e. synthesis of a new supervisor model.

Instrumenting Java Bytecode in order to catch runtime exceptions can also be easily be achieved through the Javassist *addCatch* method. This method takes to arguments: the type of exceptions under consideration and the code to be executed whenever such an exception is caught. In this work, we consider exceptions of type *java.lang.Exception*, which represents any type of exception. We also implemented a *newSupervisor* method in the Supervisor class. This methods takes into account an observed sequence as well as the current model of the supervisor and synthesizes a new model of the supervisor. This method is called whenever an a runtime exception occurs and the code is instrumented for this purpose the following way:

$$addCatch(\text{"}newSupervisor();\text{"}, java.lang.Exception) \quad (3)$$

Javassist makes it quite easy to automatically instrument legacy Java code in order to interact with a supervisor modeled as an FSM. The difficulty of this approach resides in the synthesis of a new supervisor that will ensure in the future that an observed exception cannot occur anymore, i.e. an algorithm for *Supervisor.newSupervisor*. Supervisory Control Theory offers results and algorithms that help answering this problem. However, the existing theory needs to

be augmented in order to fully achieve our goal. This theory indeed assumes that model of the system is complete, i.e. no event that are not part of the model can occur. We here consider incomplete models as runtime exceptions are not part of it.

Section 4 recalls some notations and results of the Supervisory Control Theory and then introduces new theoretical results from which an algorithm for the *Supervisor.newSupervisor* method can be derived.

4. SUPERVISORY CONTROL WITH INCOMPLETE MODELS

This section deals with the theoretical aspect of our approach. It provides definitions and results that allow for an automatic computation of a model of a supervisor that can prevent execution of traces leading to some runtime exception previously observed. The results provided in Section 4.2 ensures that the obtained model is correct as well as permissive, i.e. it ensures the objective while restricting as little behaviors of the system as possible.

4.1 Notations

For Discrete Event Systems (DES), languages over alphabets are often considered. Alphabets represents finite sets of events. A language L over an alphabet A represents a set of sequences of events in A. The set of all possible sequences of events in A is denoted A^*. Therefore for a language L over alphabet A, $L \subseteq A^*$. The concatenation of two sequences (resp. languages) s and s' (resp. L and L') is denoted $s.s'$ (resp. $L.L'$). Moreover, given a sequence $s \in A^*$, sequence $s' \in A^*$ is a prefix of s if it exists $s'' \in A^*$ such that $s's'' = s$. The set of all prefixes of s (resp. $L \subseteq A^*$) is denoted $\overline{s}$ (resp. $\overline{L}$).

Languages can represent infinite sets of sequences. In the case of regular languages, they can be represented using Finite State Machines[1]. An FSM is a 5-tuple (A, Q, q_0, Q_m, δ), where A is a finite alphabet (set of events), Q a finite set of states, $q_0 \in Q$ is the initial state of the FSM, Q_m is its set of marked states and $\delta : Q \times A \to Q$ is the partial transition function. Intuitively, $\delta(q, \sigma)$ is defined if and only if event σ can be triggered from state q. This can be extended to sequences $s = \sigma_1 \dots \sigma_n \in A^*$, i.e. $\delta(q, s)$ is defined whenever it exists states $q_1, \dots, q_n$ such that $\delta(q, \sigma_1)$ exists and equals q_1 and for all $i \in \{1, \dots, n-1\}$ $\delta(q_i, \sigma_{i+1})$ exists and equals q_{i+1}. Moreover, for a state q, $\delta(q)$ represents the set of events that can be triggered from state q, i.e. $\delta(q) = \{\sigma \in A |\ \delta(q, \sigma) \text{ is defined}\}$.

For a sequence of events $s \in A^*$, s is a possible behavior of the system if $\delta(q_0, s)$ is defined. If it is, then $\delta(q_0, s)$ represents the state that the system reaches after the sequence of event s occurred. $L_m(G)$ represents the marked language of the FSM, i.e. $\{s \in A^* |\ \delta(q_0, s) \in Q_m\}$. The set of behaviors of system G is the language generated by its FSM and is denoted $L(G)$. It corresponds to the language containing all the possible prefixes of sequences of $L_m(G)$, i.e. $L(G) = \overline{L_m(G)}$. A language L is said to be prefix-closed if $L = \overline{L}$.

In the following, whenever FSMs are graphically represented, its initial state is represented by a circle which is the target of a transition with no source state. Marked states are represented as double circles. However, unless stated otherwise, we assume that languages are prefix-closed (i.e. all the states are marked) and we only consider simple circles to represent the states in this case. Finally, the prefix-closure of a FSM is simply obtained by considering that all its state are marked, i.e. $Q_m = Q$.

[1]For simplicity, we will only consider regular languages in this work, although the results presented in Section 4 can be extended to any language over finite alphabets.

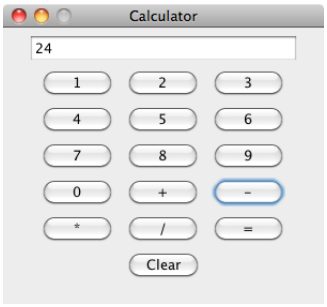


Figure 9: A basic calculator example.

4.2 Extension to Incomplete Models

In Section 2.2, the model of the system is assumed to be complete, i.e. all the possible sequences that can be observed while monitoring the system are encoded in the model. However, this requirement may not always be fulfilled. Considering the example of Figure 3 again, events act1, act2, end1, end2 could represent methods implemented in a program and events 'timeout' and 'failure' could represent some exceptions. The loop and branching of the FSM presented in Figure 3 could represent the loop and branching present in the program. Although exceptions 'timeout' and 'failure' are captured by the model, hence by the program, there may be exceptions that are not captured by the program but may occur at runtime.

In this section, we extend the formalism introduced in Section 2.2 in order to take into account the possible incompleteness of the model towards un-handled exceptions. We assume that the model is represented by a FSM on an alphabet A and more events can be observed while monitoring the system at runtime. The overall set of events is denoted A' and is a super-set of A. A may for instance correspond to all the methods and exceptions declared in the program while A' may corresponds to A to which some possible runtime exceptions that do not appear in the program source code are added. Traces of the system are represented by sequences of events in A'. The occurrence of some of these traces is not desired but can be observed at runtime. We aim to automatically compute a supervisor for the system that will prevent the occurrence of such traces.

In order to introduce and illustrate, we consider a concrete case, presented in Example 2.

EXAMPLE 2. *We consider a basic calculator with a graphical interface, presented in Figure 9.*

Figure 10 represents a Finite State Machine over an alphabet A modeling the behaviors of the calculator. A corresponds to the set of all the methods associated to events triggered when clicking the buttons represented in Figure 9, i.e. $0, \dots, 9, +, -, *, /, =$ *and 'clear'. For instance, event '0' represents the call of the method activated when button '0' is pressed. The FSM of Figure 10 encodes the possible visible*

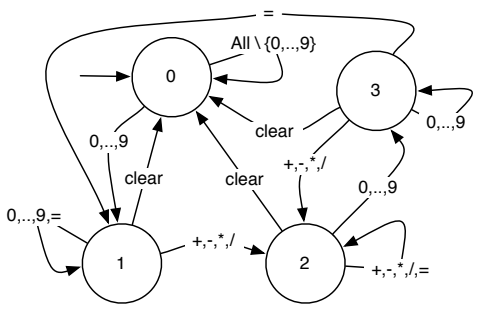


Figure 10: A model of the calculator behaviors.

behaviors of the calculator.

In this example, we assume that the exception related to the division by zero has not been handled by the programmer. In this case, a sequence such as presented in Figure 11 can be observed. Event 'exception' represents the occurrence of an exception corresponding to a division by zero that is observable at runtime. We now denote $A' = A \cup \{exception\}$ *and* $A_u = \{exception, =, +, -, *, /\}$.

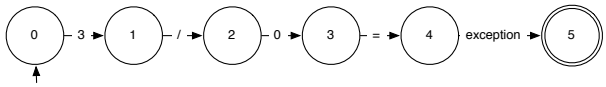


Figure 11: An undesired sequence s, exhibiting the possible occurrence of an un-handled runtime exception.

Example 2 presents a program and a possible undesired sequence of events that can be observed at runtime. It is important to note that not all the events of this sequence are responsible for the occurrence of an exception. The exception would indeed occur even if the first event of that sequence did not correspond to '3' but to any other digit. The occurrence of the exception is hence due to the division by zero and not what precedes it. Figure 12 represents some ending subsequence of the one of Figure 11 and it actually captures an undesired ending to any sequence, regardless what would precede it.

Other works such as [15] also suggest that the cause of symptoms usually occurs soon before the symptoms are observed. This entails that it is sufficient to only consider the end of an undesired sequence that was observed in order to capture what characterizes the cause of faults.

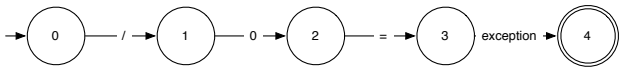


Figure 12: Ending of the undesired sequence s presented in Figure 11.

We now formalise the notions and problem presented in Example 2.

Basic Supervisory Control Problem with Incompleteness (BSCPwI): Given a system represented by an incomplete model G over an alphabet A and a sequence $s \in A^*.(A' \setminus A)$, compute the maximal controllable set of behaviors included in G preventing any behavior ending by s.

Intuitively, s represents the ending of a sequence observed during the execution of the system. However, s is not part of the system model G. Therefore, s represents new information about the possible ending sequences of the system behaviors. The BSCPwI assumes that no behavior should end by s and states that such behaviors must be prevented by control in order to remain in a subset of G.

In this section, as s represents a sequence ending a possible execution of the system, we introduce a transformation that adds prefixes to s.

Definition 1 (pre-completion). *Let A and A' be two alphabets such that $A \subseteq A'$ and let s be a sequence over A'. The pre-completion of s with respect to A is denoted by $T_A(s)$ and is defined by $T_A(s) = A^*.s$.*

Given a sequence s over alphabet A', Definition 1 defines the set of all sequences of events over A that are followed by s.

An FSM modeling the pre-completion of G with respect to A is simply obtained by adding transitions to each state of G. For each state $q \in Q \setminus Q_m$, these transitions are labeled with all events of A that cannot already be triggered from that state. These transitions

- either lead to $\delta(q_0, a)$ with label 'a', for all event 'a' such that $a \in \delta(q_0)$,
- or they lead to the initial state of the system labeled with all other events in A that cannot already be triggered from q.

Therefore, the complexity of the pre-completion algorithm is linear with respect to the number of states of the FSM.

Lemma 1. *Let A and A' be two alphabets such that $A \subseteq A'$. For any language L over alphabet A and sequence $s \in (A')^*$, we have $L \subseteq \overline{T_A(s)}$.*

Lemma 1 shows that for any undesired sequence s such as described in BSCPwI, the prefix-closure of its pre-completion $T_A(s)$ represents an over-approximation of any language G over alphabet A.

System controllability is only relevant for complete models of the system. In order to overcome this issue, an new notion relevant to the BSCPwI and related to controllability is defined in Definition 2.

Definition 2 (Partial Controllability [11]). *Let A_u, A and A' be three alphabets such that $A_u \subseteq A'$, $A \subseteq A'$ and $(A' \setminus A) \subseteq A_u$. We also consider three languages: L and L'' over alphabet A such as $L'' \subseteq L$ as well as language L' over alphabet A' such that $L \subseteq L'$. L'' is partially controllable w.r.t to $A_u \setminus A$, A_u, L and L' if*

- *L'' is controllable with respect to A_u and L.*
- *L'' is controllable with respect to $A_u \setminus A$ and L'.*

Partial controllability was introduced in [11] as a local condition allowing for global controllability for concurrent systems, i.e. systems composed of components being executed in parallel. With the notations of Definition 2, partial

controllability ensures that sequences of language L'' cannot be extended in L by a sequence over A_u that can itself be extended by a sequence of $A_u \setminus A$ in L'. In other words, if L represents the incomplete model of a system and L'' represents a set of sequences containing L and possibly ending with events in $A_u \setminus A$, then the sequence of a partially controllable cannot be completed by a sequence in A_u^* that is in L' but not in L. This notion is then very relevant to the BSCPwI.

LEMMA 2 (FROM [11]). *Given $A_u \setminus A$, A_u, L and L', it exists a unique partially controllable language w.r.t. $A_u \setminus A$, A_u, L and L'. This language is denoted $SupPC((A_u \setminus A), A_u, L, L')$ and equals $SupCont(A_u, SupCont((A_u \setminus A)L, L'), L)$.*

Lemma 2 shows the existence of a maximal partially controllable language and also provides an algorithm for it that rely on the SupCont one. We can also deduce the complexity of this algorithm from the one of SupCont: quadratic in the number of state of the FSM representing L and linear in the number of state of the FSM representing L'.

We now introduce the main theoretical result of this work. Theorem 1 shows that both pre-completion and partial controllability contribute to defining a solution to the BSCPwI.

THEOREM 1. *Let A_u, A and A' be three alphabets such that $A \subseteq A'$, $A_u \subseteq A'$ and $(A' \setminus A) \subseteq A_u$. We consider a sequence $s \in (A^*.(A' \setminus A))$ and a language L over alphabet A. If L represents an incomplete model of a system behaviors and s the ending of an undesired sequence of events, then*

$$SupPC((A_u \setminus A), A_u, L, \overline{T_A(s)})$$

is a solution to the BSCPwI.

Theorem 1 provides a solution to the BSCPwI. Intuitively, considering s represents the ending s of a sequence that lead to a runtime exception, this solution consists of transforming s into $\overline{T_A(s)}$ and considering the result of this transformation as an (over-approximated) model of the system. Then the model of the system L itself is considered as a control objective to which restrict $\overline{T_A(s)}$. This restriction is performed taking into account the controllable status of the system actions. $(A_u \setminus A)$ simply represents the possible runtime exceptions ending sequence s while A_u represents the set of actions of the system that are uncontrollable (containing runtime exceptions). Algorithm SupPC from [11] is then applied and results into a model of a supervisor preventing sequences ending by s to be executed in the future.

PROOF. In this proof, we denote

$$\text{SupPC} = \text{SupPC}((A_u \setminus A), A_u, L, \overline{T_A(s)})$$

By Definition, SupPC is the maximal sub-language of L that is partially controllable w.r.t $(A_u \setminus A)$, A_u, L and $\overline{T_A(s)}$. This entails that SupPC is included in L and is controllable w.r.t. A_u and L.

Moreover, as $s \in A^*.(A' \setminus A)$ and $L \subseteq A^*$, it means that no sequence of L ends by s. Now as SupPC $\subseteq L$, this implies that no sequence of SupPC ends by s.

Therefore it is now sufficient to show that SupPC is the largest language fulfilling the above properties. A sequence of the form $s'\sigma \in A^*$, with $s' \in$ SupPC and $s'\sigma \in L \setminus$SupPC, would only exist if one of the following held:

1. $s'\sigma \notin \overline{T_A(s)}$.
2. $s'\sigma$ either violates the controllability condition w.r.t. A_u and L or violates the controllability condition w.r.t. $A_u \setminus A$ and $\overline{T_A(s)}$.

However point 1. does not hold as $s'\sigma \in L$ and from Lemma 1, we have $L \subseteq \overline{T_A(s)}$. Moreover point 2 does not hold either as SupPC is a maximal partially controllable language w.r.t. $A_u \setminus A$, A_u, L and $\overline{T_A(s)}$. Therefore such a sequence $s'\sigma$ cannot exist and SupPC is maximal.

□

EXAMPLE 3. *In this example, we illustrate the definitions and results introduced in this section with the calculator example presented in Example 2. The problem to be solved is to control the application so that sequences ending by the sequence s of Figure 12 cannot be executed. Of course, it is desired that the possible behaviors of the application are restricted as little as possible while ensuring this goal. This problem corresponds to the BSCPwI and a solution to it is provided by Theorem 1. This solution consists of applying the SupPC algorithm which relies on SupCont and is presented in [11]. This algorithm is applied with set of uncontrollable events $A_u = \{exception, =, +, -, *, /\}$, its subset $A_u \setminus A = \{exception\}$, the control objective represented by the FSM of the calculator presented in Figure 10 and the system represented by the FSM of Figure 13.*

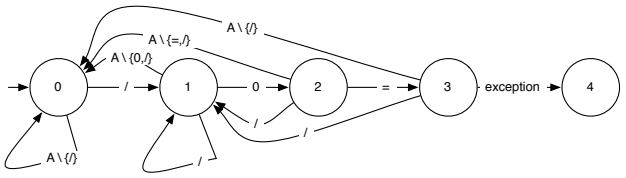


Figure 13: A FSM representing $\overline{T_A(s)}$.

Initially the application is controlled by a supervisor whose model is the one of the system, i.e. the FSM of Figure 10. During the program execution, when a sequence ends as described in Figure 12, then a new model of the supervisor is synthesized using the result of Theorem 1, which leads to the model given in Figure 14.

Comparing this model to the one of Figure 10, one can remark that from every state, clicking the '/' button is now treated differently and leads to states where the action associated to clicking button '0' is not allowed. This solution takes into account that the action associated to clicking button '=' is uncontrollable and could not be prevented by control, even after clicking button '/' followed by button '0'.

5. CONCLUSION

This work presents an approach for automatically equipping Java programs with a mechanism for self-healing of un-handle runtime exceptions. This approach relies on the Supervisory Control Theory on Discrete Event Systems and the program is automatically instrumented so that it can interact with a supervisor. This supervisor is able to prevent some method executions in order to avoid the occurrence of previously observed exceptions. Whenever a new possible exception is observed, our system automatically synthesizes a new supervisor which will prevent the execution of similar sequences of method calls that lead to this exception.

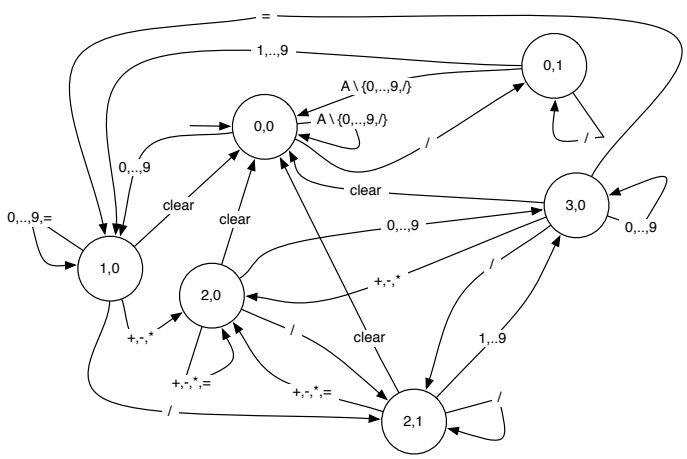


Figure 14: A supervisor preventing the occurrence of the trace in Figure 12.

Our approach relies on a Finite State Machine (FSM) modeling the possible sequences of method calls of the program. This FSM does not take into account the values of the program variables but rather its behavior. Although our current implementation does not yet allow for the automatic computation of such an FSM for the whole program, we can already extract FSMs for each of its methods. This approach has been applied to the JEdit application [3] and makes it possible to extract FSMs for all of its methods (over 6500) in about 3 minutes. While so far our implementation takes into account Java structures such as branchings and loops, our main future work in this direction consists of also taking into account the concurrency between threads as well as graphical components, hence allowing for automatic FSM extraction from industrial applications.

Also in our approach, models of a supervisor are automatically computed from a control objective that is itself derived from the ending of a sequence of method calls leading to an exception. The size of this ending sequence together with the initial model of the program impact on the relevance of the obtained solution. Our future works will consider criteria on how to set an appropriate value for the size of the ending sequence. We will also investigate validation results and techniques for the relevance of automatically computed supervisors, whenever solutions for several possible sequence sizes are considered.

Finally, in some cases such as the calculator example presented in this paper, the application does not need to be restarted after a new supervisor has been automatically computed. However, our approach usually requires such a restart. In order to overcome this issue, we are planning on integrating checkpointing techniques with Supervisory Control concepts in order to avoid application restarts whenever a new supervisor is synthesized.

6. REFERENCES

[1] Fastfix. https://services.txt.it/fastfix-project/?q=welcome.

[2] Javassit. http://www.csg.is.titech.ac.jp/ chiba/javassist/.

[3] Jedit. http://www.jedit.org/.

[4] S. Abdelwahed, N. Kandasamy, and S. Neema. A control-based framework for self-managing distributed computing systems. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 3–7, New York, NY, USA, 2004. ACM.

[5] A. Carzaniga, A. Gorla, and M. Pezzè. Healing Web applications through automatic workarounds. *International Journal on Software Tools for Technology Transfer (STTT)*, 10(6):493–502, 2008.

[6] K.-H. Cho. Robust supervisory control of communication networks. *Science and Technology, 2001. KORUS '01. Proceedings. The Fifth Russian-Korean International Symposium on*, 1:46 –47 vol.1, jun. 2001.

[7] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 439–448. IEEE, 2002.

[8] R. de Lemos. ICSE 2003 WADS Panel: Fault Tolerance and Self-Healing. 2003.

[9] Y. Diao, J. Hellerstein, S. Parekh, R. Griffith, G. Kaiser, and D. Phung. Self-managing systems: A control theory foundation. *Selected Areas in Communications, IEEE Journal on*, 23(12):2213–2222, 2005.

[10] S. Dobson, R. Sterritt, P. Nixon, and M. Hinchey. Fulfilling the vision of autonomic computing. *Computer*, 43:35–41, 2010.

[11] B. Gaudin and H. Merchand. An efficient modular method for the control of concurrent discrete event systems: A language-based approach. *Discrete Event Dyn Syst*, 17(2):179–209, Apr 2007.

[12] N. Gruska, A. Wasylkowski, and A. Zeller. Learning from 6,000 projects: lightweight cross-project anomaly detection. In *ISSTA '10: Proceedings of the 19th international symposium on Software testing and analysis*, pages 119–130, New York, NY, USA, 2010. ACM.

[13] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36:41–50, 2003.

[14] M. M. Kokar, K. Baclawski, and Y. A. Eracar. Control theory-based foundations of self-controlling software. *IEEE Intelligent Systems*, 14:37–45, 1999.

[15] K. Pan, S. Kim, and E. J. Whitehead, Jr. Toward an understanding of bug fix patterns. *Empirical Softw. Engg.*, 14(3):286–315, 2009.

[16] P. J. Ramadge and W. Wonham. Supervisory control of discrete event processes. In *Feedback Control of Linear and Nonlinear Systems*, volume 39 of *LNCIS*, pages 202–214. Springer-Verlag , Berlin, Germany, 1982.

[17] D. Reilly, A. Taleb-Bendiab, A. Laws, and N. Badr. An instrumentation and control-based approach for distributed application management and adaptation. In *Proceedings of the first workshop on Self-healing systems*, pages 61–66. ACM, 2002.

[18] H. Shah, C. Gorg, and M. Harrold. Why do developers neglect exception handling? In *Proceedings of the 4th international workshop on Exception handling*, pages 62–68. ACM, 2008.

[19] O. Shehory. A self-healing approach to designing and deploying complex, distributed and concurrent software systems. In *ProMAS'06: Proceedings of the 4th international conference on Programming multi-agent systems*, pages 3–13, Berlin, Heidelberg, 2007. Springer-Verlag.

[20] R. Taylor and C. Tofts. Self managed systems-a control theory perspective. In *Self-Star Workshop*, 2004.

[21] W. Torres-Pomales. Software fault tolerance: A tutorial. *NASA Langley Research Center, Hampton, Virginia, TM-2000-210616*, 2000.

[22] W. M. Wonham. Notes on control of discrete-event systems. Technical Report ECE 1636F/1637S, Department of Electrical and Computer EngineeringUnivertsity of Toronto, July 2003.