An Approach to Identifying and Understanding Problematic COTS Components *

Gregory M. Kapfhammer, C.C. Michael, Jennifer Haddox, Ryan Colyer †
Reliable Software Technologies Corporation
21351 Ridgetop Circle, #400, Dulles, VA 20166
http://www.rstcorp.com

August 22, 2000

Abstract

The usage of Commercial off the Shelf (COTS) components in software systems presents the possibility of temporal savings and efficiency increases. However, this temporal savings might come at the expense of system quality. When a system integrator relies upon COTS software, trust is placed in unknown, black-box components. We present a methodology that identifies problematic COTS components and then attempts to augment a system integrator's understanding of these components. Our technique uses software fault injection to expose COTS components to new failure scenarios. When these unique failure scenarios cause a COTS component to act in an unpredictable manner, our approach records the injected fault and the anomalous behavior. Next, we employ different machine learning techniques to build a representation of the anomalous behavior of the COTS component. These machine learning algorithms analyze the collected data, which describes the diverse conditions that cause a COTS component to behave unpredictably, and produce a comprehensive model of the combinations of input and component state that normally result in deviant behavior. A system integrator can inspect a graphical representation of this model in order to gain a better understanding of the anomalous COTS components. We believe our approach to isolating and understanding problematic COTS components will allow a system integrator to realize the temporal savings of reusable COTS software while also mitigating the associated risks.

1 Introduction

The usage of Commercial off the Shelf (COTS) components in software systems is normally associated with a reduction in the cost of software design and development. Since the best programmers can only produce ten lines of code per day, system integrators want to leverage the temporal and monetary savings that software COTS components could introduce [13]. Government agencies have started to mandate that software developers exploit the COTS marketplace, and commercial organizations are using COTS to increase their time-to-market. However, the current excitement about COTS components has been dampened by the realization that it is

*This research is supported by ARL Research Contract DAAD17-99-C-0052.

†Dr. Michael is the correspondence author and can be contacted at ccmich@rstcorp.com.

difficult to determine: (1) the quality of a newly-acquired COTS component and (2) the system's tolerance to the COTS components [13]. The widespread adoption of COTS components will not occur until adequate measures exist to identify and understand the problematic COTS components in a software system.

One approach to identifying and understanding problematic COTS components relies upon the existence of a black-box behavioral specification for the chosen component. A behavioral specification that treats a COTS component as a black box can be used by a system integrator to determine how the component might perform anomalously in the candidate system [1, 8]. Since this black-box specification must specify all important or relevant visible behavior of the COTS component, the cost of producing it is normally non-trivial. Moreover, most software COTS component developers never create these full specifications. Thus, it is not reasonable to use black-box behavioral specifications to determine which COTS components in a candidate system might behave anomalously.

Another approach to finding problematic COTS components and understanding their anomalous actions relies upon source code access. Software testability analysis [16] and software component dependability assessment [14] could be used to measure different aspects of COTS component quality. Software testability analysis employs a white-box, dynamic, failure-based approach to component assessment. Instead of attempting to reveal the existence of faults, testability analysis determines component source locations where faults (if they exist) are most likely to remain undetected by further testing [11]. Software component dependability assessment uses a modified testability analysis in order to provide insight into the thoroughness of testing that a specific component received [14]. However, each of the above approaches, and many others as well, require access to the source code of the COTS component. Since source code is almost never available for COTS, neither of the above techniques can be used to isolate and understand problematic COTS software.

Another approach to the examination of COTS components could utilize a simple form of behavioral specification while still relying upon execution-based evaluation. We propose a methodology that combines software fault injection at component interfaces [12] and machine learning techniques [2, 7, 9, 10] in an attempt to identify problematic COTS components and understand their anomalous behavior. After a system integrator specifies certain scenarios that COTS components should not experience, our approach uses software fault injection to introduce the system to new failure situations. The usage of fault injection at COTS component interfaces will help us to determine how the COTS components affect the system and how the system influences the COTS components. In many situations, fault injection at a chosen component interface will create a host of input and state combinations that cause other components to enter a scenario that was previously designated as problematic. Our approach uses different machine learning techniques to build a model of the anomalous behavior of a COTS component in an attempt to help the system integrator better understand the black boxes in the candidate system.

2 An Example System

In order to better explain our methodology for isolating and understanding deviant COTS components, we will provide an example system. Figure 1 offers a schematic of an Automated Teller Machine (ATM) system. In this example, a user interacts with the `ATM` unit, which could be a software-enabled embedded device. Once the user indicates the desire to interact with a specific

bank, the ATM contacts the needed Bank. If the user decides to perform any type of transaction, the Bank can utilize the TransactionAgent to perform withdrawls, deposits, and account status checks. The system that has been described could be developed as a software system that uses DCOM, CORBA, or Jini to facilitate the communication of distributed components. Alternatively, the system could be designed to run on a single machine and implemented with many different general purpose programming languages. We have implemented this example system with Java and Jini. In our system, the TransactionAgent is treated as a COTS component and the BankDatabase is a COTS database developed by outside sources. Furthermore, the TransactionAgent relies upon COTS Java Database Connectivity (JDBC) drivers to facilitate database communication.

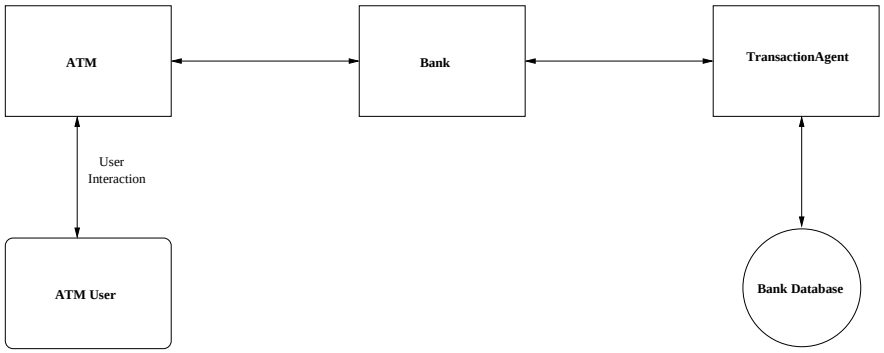

Figure 1: High-level Diagram of the ATM system.

3 Identifying and Understanding Problematic COTS

Our methodology for identifying problematic COTS components and understanding their anomalous behavior combines unique stages of fault injection, data collection, and machine learning. Figure 2 offers an overview of our approach to isolating problematic COTS components and then understanding their deviant behavior. Once an integrator has assembled a working version of a software system composed of COTS components, glue-code, and custom made components our methodology can be employed. Since the usage of COTS components accelerates system development, our methodology can be applied earlier in the software lifecycle.

In order to start using our approach, a system integrator must define the anomaly predicates that will specify inappropriate COTS component behavior. These anomaly predicates will be indicators that the current state of a component and the inputs to a chosen operation will cause it to perform in an undesired manner. Anomaly predicates can be defined in a very coarse manner, but it is expected that fine-grained predicates will help to yield a better understanding of component behavior. An example of an anomaly predicate for the `accountDeposit(int acctID, double amount)` method provided by the TransactionAgent might be,

$$PRED_{dep} = \neg isValid(acctID) \vee \neg isValid(connection)$$

This predicate indicates that the `accountDeposit()` method will perform in an anomalous manner whenever an invalid account ID is provided or the TransactionAgent has an invalid connection to the BankDatabase. Anomaly predicates can be defined in terms of the visible inputs

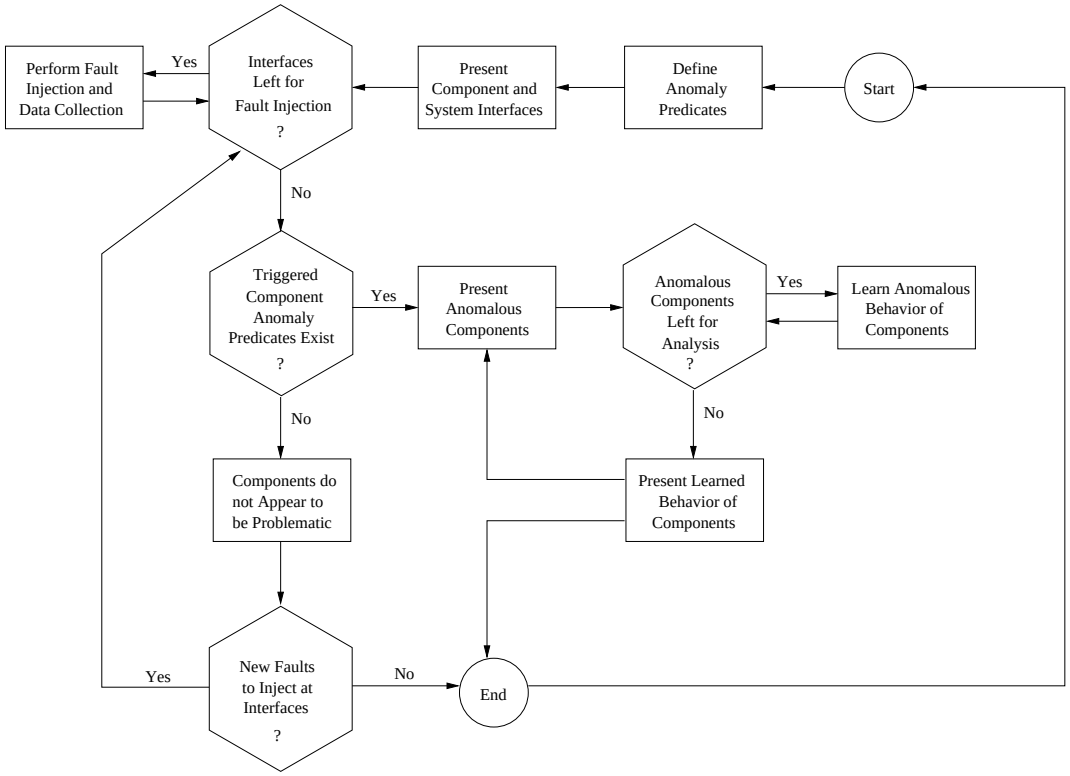

Figure 2: A Methodology for Identifying and Understanding Problematic COTS Software.

and outputs of the interface for a COTS component or any portion of the system.

Once the system integrator has defined anomaly predicates for the desired COTS components, our methodology relies upon software fault injection [15] to introduce the system to new failure states. Initially, the system integrator will be presented with a listing of all the public component and system interfaces. Fault injection will be used to perturb the inputs and outputs of the chosen interfaces in order to simulate component and system failure. A collection of pre-defined and customized perturbations will be used at each interface that is deemed to be important by the system integrator. When a perturbation causes a COTS component to trigger its anomaly predicate and to operate in an undesired manner, the type of injected fault, the component's input and output, and other pertinent data will be collected.

Suppose that we are performing fault injection on our example system. In this system, before the `Bank` can use the `TransactionAgent`, it must initialize a connection to the appropriate database. If the `TransactionAgent` provides an `initConnection(String url)` method, we could easily perform fault injection at this interface. If the `Bank` normally calls the `initConnection()` method and passes a string like `jdbc:mysql://BankName/DatabaseName`, we can perturb this input to a URL that has an illegal form. In this example, the implementation of the `initConnection()` method throws a `DatabaseConnectException` whenever it is unable to create a viable connection to the specified database. Even though the throwing of a `DatabaseConnectException` indicates that the `TransactionAgent` was unable to connect to the desired database, the naive implementation of the `Bank` catches the exception and simply prints debugging information. Since the `Bank` never recognizes that the `TransactionAgent` does not have a viable connection to the requested database, a perturbation of this nature will cause the system

to crash if the TransactionAgent is used to withdraw money from an account.

Our methodology performs fault injection at all of the appropriate component and system interfaces. While the previous example focused on performing perturbations at the interface to the TransactionAgent, it is also possible to perturb at the interface to the Bank or the BankDatabase. In a more complex, real-world system, there will be a larger number of interesting interfaces at which we could introduce input data perturbations. Fault injection at different system and component interfaces will allow our approach to learn more about how the system performs in unique failure conditions. Furthermore, fault injection at several different interfaces will allow our methodology to collect a large body of data concerning the behavior of components after faults have been injected. Refer to Section 4 for a detailed discussion of the types of faults that our methodology could inject. If our approach injected faults at the interface to the initConnection() method and the Bank later called the accountDeposit() method, the previously defined anomaly predicate for this method, denoted $PRED_{dep}$, would be triggered. When $PRED_{dep}$ was triggered, our approach would record the pertinent information surrounding the anomalous behavior of the TransactionAgent.

If the usage of fault injection has caused some of the components in the system under analysis to trigger their anomaly predicates, our approach presents the anomalous components to the system integrator. At this point, the system integrator will know that certain components in the system do not perform robustly in the presence of unique failure conditions. In an attempt to augment the knowledge of the system integrator with an understanding of the anomalous behavior of certain components, our approach uses a toolkit of machine learning algorithms to model the behavior of these components. We employ techniques that learn finite automata from the behavioral data that was collected during the fault injection process [2, 7, 9]. These learning algorithms attempt to build a finite state machine where certain FSM states represent anomalous component states and specific transitions cause the component to enter these states. Refer to Section 5 for a discussion of the application of machine learning algorithms to the problem of learning the behavior of anomalous software.

Once our approach has attempted to learn the anomalous behavior of certain components, this information is presented to the system integrator. In some situations, the system integrator might reapply our learning algorithms in an attempt to build another model of anomalous behavior or refine the model that was already produced. In other situations, the characterization of the anomalous behavior of the software components will help the system integrator make intelligent decisions concerning future software development and integration actions. For example, our methodology could develop a model of the anomalous behavior of the TransactionAgent. This model would reinforce the system integrator's understanding that the accountDeposit() method fails when the TransactionAgent is unable to obtain a connection to the desired database. Furthermore, the behavioral model might help the system integrator to realize that the failure of the accountDeposit() method could be directly related to the inappropriate manner in which the Bank handles the DatabaseConnectException that is thrown by the TransactionAgent. If the TransactionAgent was a COTS component and the Bank was a custom developed component, it would be possible to rewrite the Bank to handle the thrown exception in a more appropriate manner.

In some situations, the fault injection used by our methodology might not reveal any problematic components. When this occurs it is an indication that the current collection of perturbations was unable to introduce failure conditions that caused components to trigger their anomaly pred-

icates. At this stage, the system integrator could define new perturbations and return to the fault injection and data collection stage for any number of interfaces in the system. Since fault injection is a probabilistic process, the system integrator could also choose to restart the methodology and use the same collection of perturbations in another attempt to isolate and understand deviant COTS components. The injection of the same fault types could still cause components in the system under analysis to trigger their anomaly predicates.

Regardless of whether new perturbations or the same perturbations are used, a system integrator can continue to apply our methodology until he feels comfortable that he has successfully identified and attempted to understand the problematic COTS components in his system. It is important to note that the prototype (which is currently under development and designed to implement our methodology) allows system integrators to customize the ordering and usage of the fault injection, data collection, and machine learning stages. This will allow system integrators to tailor our approach to most effectively isolate and understand the problematic COTS components in their system.

4 Fault Injection Overview

Our approach for isolating and understanding problematic COTS components relies upon software fault injection to introduce the system to unique failure conditions. Initially, we have targeted our methodology to apply to Java-based systems that might utilize Jini to perform distributed computing. Due to this choice, we are implementing a fault-injection system for Java and Jini that relies upon the ability to wrap the interfaces where perturbations must be performed. These wrappers are able to intercept method calls and modify the data that enters and leaves chosen interfaces. In this section, we will provide a high-level overview of the techniques we use to wrap Java and Jini interfaces in order to perform fault injection. We will also examine the types of faults that our system is able to inject. For a detailed discussion of our approach to wrapping Java components and Jini services, refer to [5, 6].

Figure 3 presents the high-level usage of the wrapping system that is needed to facilitate the introduction of input and output data perturbations. In this usage example, interface perturbations are introduced at the interface to the `TransactionAgent` in the ATM system that was described in Section 2. In this situation, when the `Bank` attempts to construct an appropriate `TransactionAgent` the class loading request is intercepted by our wrapping system. Once the wrapping subsystem has gained control of the `TransactionAgent`'s loading event, it uses runtime bytecode instrumentation to change the structure of the component's methods. This instrumentation introduces bytecode that encases the `TransactionAgent` with a `FaultInjectionWrapper`. This wrapper contains hooks to the fault injection system so that it can introduce perturbations at the interface to the `TransactionAgent`. For example, the `FaultInjectionWrapper` could be used to perturb the inputs to the `initConnection(String url)` operation provided by the `TransactionAgent`.

We have currently developed and are extending a `FaultInjectionWrapper` that can encase Java components and inject a number of different types of faults. Our perturbations are based on the data type(s) that a given method accepts as input or returns as output. The faults that we inject for enumerated types include the ability to increment or decrement by some offset and the option to randomly select a member of the enumerated set. When our fault injection system deals with numeric types, the `FaultInjectionWrapper` can increment or decrement by an offset, set

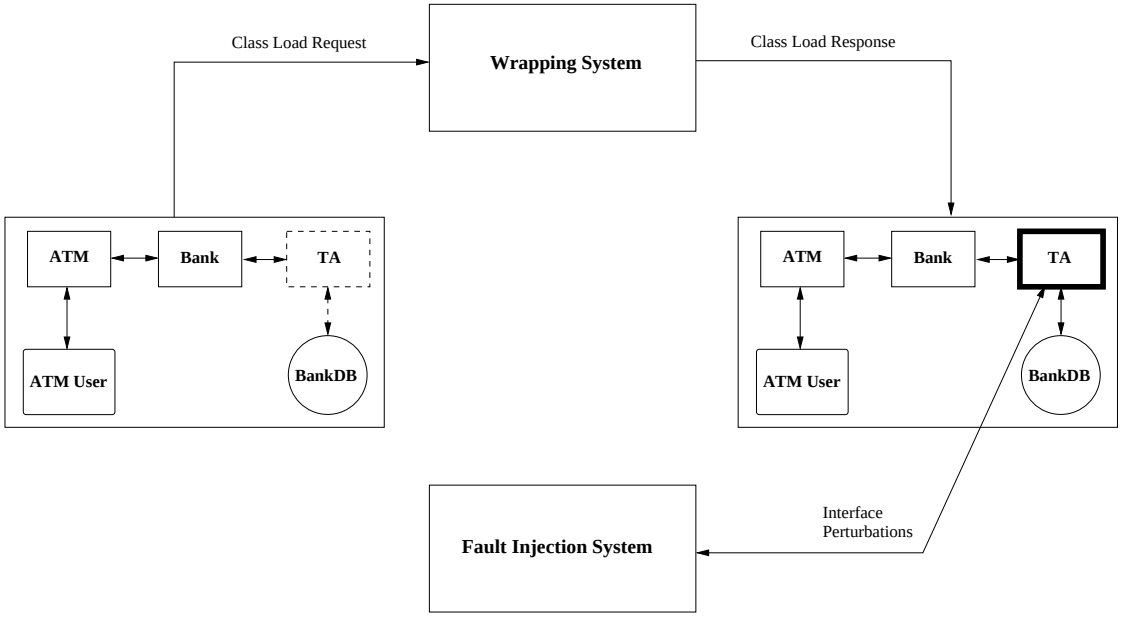

Figure 3: High-level Usage of the Wrapping Subsystem to Perform Fault Injection.

the variable to zero, invert the sign, assign the variable to the maximum positive or negative value, and select a random value from the range of possible values for the data type. If the fault injection system deals with a complex object, it currently can set the object to `null` or assign the object to a user-defined format. We are currently experimenting with mechanisms that attempt to manipulate the data members of components in a semantically correct fashion.

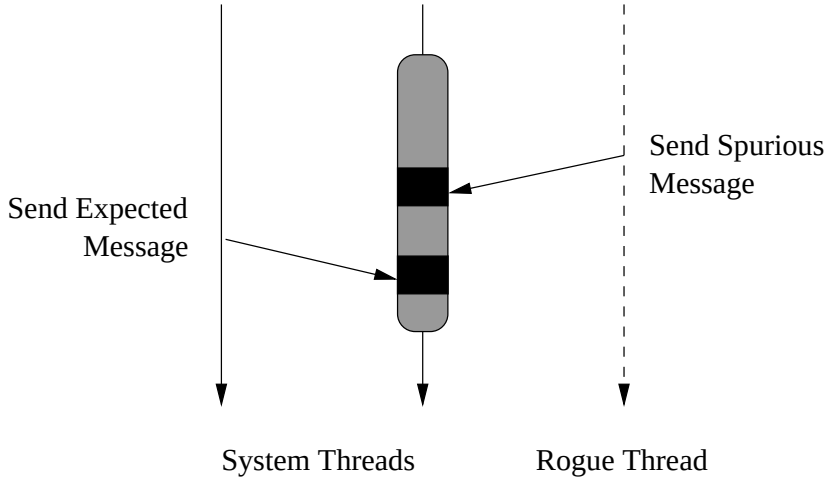

Figure 4: A Rogue Thread that Sends a Spurious Message.

Our `FaultInjectionWrapper` is also able to instruct our fault injection system to create rogue threads that help to assess how the system under test handles timing issues. Figure 4 shows an example of a rogue thread that is used to send a spurious message to an operational thread is the system. In this scenario, one system thread expects to receive a message from another system thread during a given window of time. Even though the rogue thread is not

directly aware of this time window, it can randomly send messages to other system threads in an attempt to disrupt system timing. Since the rogue thread in our example sends a message to the system thread much earlier than expected, it is possible that this unique failure condition could cause a component to trigger its anomaly predicate.

The `FaultInjectionWrappers` are also able to introduce method call collection faults. This type of fault requires that a wrapper collect a given number of calls to the method provided by a component. Once the appropriate amount of calls has been collected, the `FaultInjectionWrapper` can release them at the same time. Finally, if the system integrator needs to introduce other failure conditions into his system, he can develop a new wrapper that provides these facilities. Since the `FaultInjectionWrapper` and the associated fault injection system are Java classes that integrate into a wrapper hierarchy, the system integrator can easily define new wrappers that introduce new fault types. This is possible because our wrapping system can be configured to return components that are encased by any wrapper that implements the appropriate wrapper interfaces.

5 Machine Learning Examples

Our approach to isolating problematic COTS components uses a `FaultInjectionWrapper` to introduce the system under analysis to unique failure conditions and a `DataCollectionWrapper` to gather the circumstances surrounding anomalous component behavior. At this point in our methodology, it is possible to analyze the collected data with machine learning algorithms in an attempt to develop an understanding of a component's anomalous behavior. Before it is possible to actually use our toolkit of machine learning algorithms, it is necessary to pre-process the collected data. During the pre-processing stage, our approach generates a finite state machine that describes the different steps that lead up to the triggering of a component's anomaly predicate.

In an attempt to understand the root cause behind the triggering of an anomaly predicate, we employ state merging algorithms [2, 7, 9, 10]. The finite state machine that was generated during the pre-processing stage, normally called the acceptor tree, is manipulated by this type of learning algorithm. Learning takes place by selectively merging states in the acceptor tree. State merging algorithms vary in their choice of which states become merged and which remain separate. Suppose that the pre-processing stage develops a state machine that describes the conditions under which the `TransactionAgent` triggers its anomaly predicate. A simple state merging algorithm might choose to merge two states in the FSM if they are both the root node of the same subtree. At one extreme, state merging algorithms can simply traverse the acceptor tree and determine whether it is appropriate to merge each encountered state. At the other extreme, the learning algorithm could construct an optimality condition for the entire tree and treat the learning problem as a global optimization problem [9]. Refer to [9] for more detailed examples of state merging algorithms that are used for learning finite state machines.

We are currently involved in the refinement of our toolkit of machine learning algorithms so that we can enhance their ability to develop models of the anomalous behavior of Java components. In past research, we have tailored state merging algorithms to learn program behavior profiles that can be used in intrusion detection systems [3, 4]. Figure 5 presents an example of a finite state machine that is the product of the state merging algorithms that were employed in past intrusion detection research. This finite state machine models the behavior of the Unix `eject` program, which simply allows for the software controlled ejection of removable media. In

this example, the anomaly predicate for the component is that the usage of the program does not allow someone to gain root access. In this finite state machine, the dotted transitions represent program actions that are not associated with anomalous behavior and the bold transitions denote anomalous program actions.

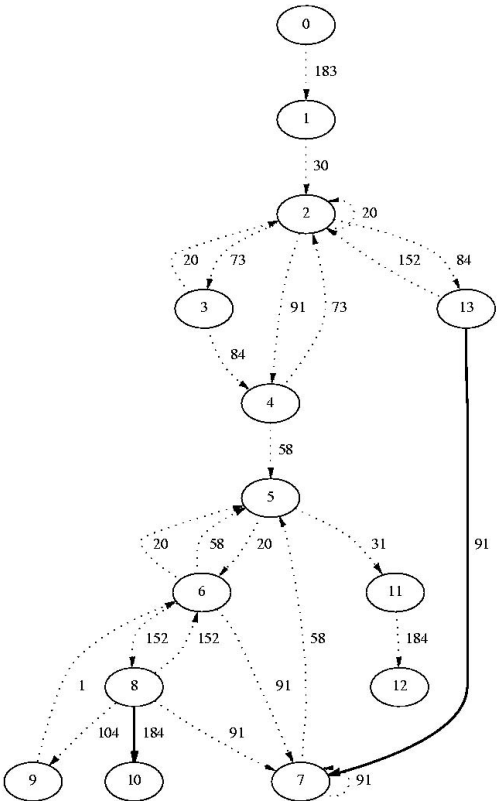

Figure 5: A Learned State Machine from the `eject` Program.

In the state machine example in Figure 5, transition 91 is labelled as anomalous. If the system integrator were to refer to the execution action that corresponded to transition 91, he could note that `eject` was performing an `open()` system call. Using this learned model of program behavior, the system integrator could delve further into the execution of `eject` to determine the name of the file or device that forces the program into an anomalous state when it is opened. The state merging algorithm has also labelled transition 184 as anomalous. This transition corresponds to the termination of the `eject` program. Since there is nothing essentially wrong with the termination of the program, the system integration can determine that it must be anomalous to terminate in state 10. Using the execution traces and the learned state machine, the system integrator could realize that `eject` terminates in a non-anomalous fashion after transitioning through state 11. The program is able to terminate in a normal fashion because the transition that leads into state 11 corresponds to a call to the `exit()` system call. Using the distilled information in the finite state machine, the system integrator can conclude that `eject` also performs in an anomalous fashion whenever it terminates before exiting properly.

6 Conclusions and Future Work

Even though the incorporation of COTS components into software products presents the possibility of temporal and monetary savings, there are inherent risks associated with the usage of black-box COTS software. Other approaches to software analysis, such as behavioral specification examination and source-based analysis, are not suitable for the the task of mitigating the risk that software COTS components introduce. In this paper, we have discussed an approach that can be used to isolate and understand problematic COTS components. In our methodology, unique stages of fault injection, data collection, and machine learning are used to discover deviant COTS components and understand their primary mode of malfunction. Our approach uses software fault injection to introduce the system under analysis to unique failure conditions. We then use data collection techniques to record the situations in which COTS components perform in an anomalous manner. Finally, we rely upon machine learning techniques to build an understanding of the root cause of a component's deviant actions. When a system integrator is armed with a knowledge of which components perform in an unpredictable manner and an understanding of their behavior, he can more intelligently handle the risks introduced by COTS software.

Our research into the usage of fault injection, data collection, and machine learning for the identification and understanding of problematic COTS components has presented a number of new research possibilities. Past results indicate that it is feasible to apply state merging techniques for the task of learning the anomalous behavior of Unix programs like `eject`. Our preliminary research indicates that it is also possible to tailor existing algorithms to learn the anomalous behavior of Java COTS components. Further refinement of machine learning algorithms will serve to strengthen our approach of identifying and understanding black-box COTS software. Highly evolved techniques for building finite state machines that describe software component behavior can also be applied in other areas. As proposed in [13], the model of a component's anomalous behavior could be used to develop a heuristic wrapper. This wrapper would be able to selectively shield the operations of a component from inputs that are predicted to cause the component to perform in an unpredictable manner. Once this technique has been fully developed, system integrators would have a complete method to mitigate the risk that these problematic COTS components introduce into a system.

References

[1] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages*, 15(1):73–132, January 1993.

[2] Yoav Freund, Michael Kearns, Dana Ron, Ronitt Rubinfeld, Robert E. Schapire, and Linda Sellie. Efficient learning of typical finite automata from random walks. *Information and Computation*, 138(1):23–48, 10 October 1997.

[3] Anup K. Ghosh, Aaron Schwartzbard, and Michael Schatz. Learning program behavior profiles for intrusion detection. In *Proceedings of the 1st USENIX Workshop on Intrusion Detection and Network Monitoring*, Santa Clara, California, April 1999.

[4] Anup K. Ghosh, Aaron Schwartzbard, and Michael Schatz. Using program behavior profiles for intrusion detection. In *Proceedings of the SANS Third Conference and Workshop on Intrusion Detection and Response*, San Diego, CA, February 1999.

[5] Gregory M. Kapfhammer. An automated wrapping system for java components and jini services. *Presentation at the 17th Int'l. Conference and Exposition on Testing Computer Software*, 2000.

[6] Gregory M. Kapfhammer, Jennifer Haddox, Michael A. Schatz, and Ryan Colyer. An approach to wrapping java components and jini services. *Submitted to the 25th Annual Software Engineering Workshop*, 2000.

[7] K. D. Lang. Evidence-driven state merging with search. NECI technical report TR98-139, 1998.

[8] Nancy G. Leveson. Using cots components in safety-critical systems. In *RTO Meeting on COTS in Defense Applications*, Brussels, Belgium, April 2000.

[9] C.C. Michael and Anup Ghosh. Using finite automata to mine execution data for intrusion detection: A preliminary report. In *Proceedings of the Third International Workshop in Recent Advances in Intrusion Detection*, Toulouse, France, October 2000.

[10] B. A. Trakhtenbrot and Ya. A. Barzdin. *Finite Automata: Behavior and Synthesis*. North-Holland, 1973.

[11] Jeffrey Voas. Pie: A dynamic failure based technique. *IEEE Transactions on Software Engineering*, 18(8):717–727, August 1992.

[12] Jeffrey Voas. An approach to certifying off-the-shelf software components. *IEEE Computer*, 31(6):53–59, June 1998.

[13] Jeffrey Voas. Defensive approaches to testing systems that contain cots and third-party functionality. In *Proc. of 15th Int'l. Conference and Exposition on Testing Computer Software*, June 1998.

[14] Jeffrey Voas and Jeffrey Payne. Dependability certification of software components. *To Appear in Journal of Systems and Software*, 2000.

[15] Jeffrey M. Voas and Gary McGraw. *Software Fault Injection: Innoculating Programs Against Errors*. John Wiley and Sons, Inc., New York, 1998.

[16] Jeffrey M. Voas, Keith W. Miller, and Jeffrey E. Payne. Pisces: A tool for predicting software testability. In *Proceedings of the Symposium on Assessment of Quality Software Development Tools*, pages 297–309, New Orleans, LA, May 1992. IEEE Computer Society.