Policy Specification for Non-Local Fault Tolerance in Large Distributed Information Systems

A Thesis

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Master of Science

by

Philip E. Varner

May 2003

© Copyright May 2003

Philip E. Varner

All rights reserved

Approval Page

This thesis is submitted in partial fulfillment of the requirements for the degree of

Master of Science

Philip E. Varner

Approved:

John C. Knight (Advisor)

David Evans (Committee Chair)

Anita K. Jones

Accepted by the School of Engineering and Applied Science:

Richard W. Miksad (Dean)

Dean, School of Engineering and

Applied Science

May 2003

Abstract

The services provided by critical infrastructure systems are essential to the operation of modern society. These systems include the financial payments system, transportation systems, military command and control systems, the electric power grid, and telecommunications systems including the Internet. Widespread failure of any of these system might result in severe financial loss or perhaps human injury. Critical infrastructure systems rely heavily on distributed information systems for operation. These information systems must therefore be *dependable*; that is, they must "deliver service that can justifiably be trusted."

Traditional dependability alone does not provide a rich enough model to deal with the faults in large, critical distributed systems operating in hostile environments. These systems require not simply dependability but instead require survivability. Informally, survivability is when a system has "the ability to continue to provide service (possibly degraded or different) in a given environment when various events cause major damage to the system or its operating environment.".

One means of achieving survivability is non-local fault tolerance, where faults that affect significant portions of the network must be detected and handled in a coordinated fashion. Our approach to doing this is with a survivability control system. This control system takes network sensor events as input, uses these to detect faults, and responds with application reconfiguration. This thesis presents TEDL, the Time-based Event Detection Language, for formal specification of the reactive policy of this control system. A translator is used to synthesize an executable implementation from this specification. The results from using TEDL to describe and execute several attack and failure scenarios for a simplified financial payments system are presented.

Acknowledgments

First of all, I would like to thank my Mom and Dad, Bettie Sue and Jerry Varner. Without their continual support, I would not be where I am today. Thanks to my fiancee Christy Pagels for putting up with me while I was working on this. Thanks to my advisor John Knight for getting me into this mess. Thanks to Jonathan Hill for being a great research role model. Thanks to the Legion Research Group for access to the Centurion cluster.

This work was supported in part by the Defense Advanced Research Projects Agency under grant N66001- 00-8945 (SPAWAR) and the Air Force Research Laboratory under grant F30602-01-1-0503. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, the Air Force, or the U.S. Government.

Contents

1 Introduction **1**

2 Survivability and the Willow Architecture **4**

2.1 Survivability 4

2.2 Survivability Control Systems 9

2.3 The Willow Architecture 9

3 Language Role and Requirements **14**

3.1 Scalability 15

3.2 Expressivity 16

3.3 Ease of Use 17

3.4 Implementability 17

3.5 Analyzability 17

3.6 Summary 18

4 Language Design **19**

4.1 Declarative Object-oriented Constructs 20

4.2 A Model of Time 21

4.3 Event-Driven Architecture 23

4.4 Inherited Hierarchy of Event Types 24

4.5 Dynamic Sets of Events 25

4.6 Finite State Machines for Detection 27

4.7 Hierarchy of Finite State Machines 29
4.8 Reaction 31
4.9 Machine Event Generation 33
4.10 Assessment of success and subsequent recovery 33
4.11 Exception handling 34
4.12 Transforms 34
4.13 Summary 35

5 Implementation **36**

6 Experimentation and Assessment **39**
6.1 Evaluation Challenges 39
6.2 Financial Payments System Scenarios 40
6.3 Summary 47

7 Language Evaluation **48**
7.1 Scalability 48
7.2 Expressivity 50
7.3 Ease of Use 52
7.4 Implementability 52
7.5 Analyzability 53
7.6 Additional Limitations 53

8 Related Work **54**
8.1 Fault-tolerant Distributed Systems 54
8.2 Languages 56
8.3 Event Correlation 58

9 Conclusion **61**
9.1 Conclusion 61

9.2 Future Work 62

A Language Grammar **63**

B TEDL Source Files **68**

B.1 File siena.tedl 68

B.2 Money Center Bank Failure Scenario 68

B.3 Coordinated Attack Scenario 69

B.4 Regional-level Power Failure Scenario 71

Bibliography **73**

List of Figures

2.1 Willow Architecture 10

4.1 Example Finite State Machine Detector 20
4.2 Event Inheritance Hierarchy 25
4.3 Example Finite State Machine Hierarchy 30

5.1 SPARTAN Implementation Architecture 37

6.1 Example Financial Payments System Diagram 41
6.2 MCBFailureDetector State Diagram 41
6.3 CoordinatedAttackDetector State Diagram 42
6.4 RegionalFailureDetector State Diagram 46
6.5 NationalFailureDetector State Diagram 46

List of Tables

6.1 Results for Money Center Bank Failure Scenario (times in seconds) 42

6.2 Results for Coordinated Attack Scenario (times in seconds) 43

6.3 Command Response Times for Coordinated Attack Scenario (times in seconds) . . . 44

6.4 Response times for Selective Notification (times in seconds) 45

Chapter 1

Introduction

> *Electric power is the critical utility. After more than about three days (of failure) everything just folds up. Trains, heat, refrigeration, water supplies all go. We'd be straight back to 18th and 19th century, and it would take 20 years to regain the lost economic capability. – Ross Anderson*

Our lives depend on distributed information systems. Widespread failure of any critical infrastructure system would result in high financial loss and possibly human injury. These systems include financial payments systems, transportation systems, military command and control systems, the electric power grid, and telecommunications systems including the Internet. These critical infrastructures rely heavily on distributed information systems for operation. Failure of the underlying information systems will usually result in failure of the infrastructure system. The information systems must therefore be *dependable*; that is, they must "deliver service that can justifiably be trusted" [74]. This thesis presents TEDL, the Time-base Event Detection Language. TEDL is a language for specifying the detection of errors and repair of faults in Internet-scale distributed systems. It has been designed to be scalable, expressive, easy to use, implementable, and analyzable. These properties allow manageable description of complex fault scenarios in large information systems.

Great care is taken to avoid introducing design faults and minimizing degradation faults in critical distributed systems. However, faults will always be a part of such systems. Therefore, they

must be handled through a combination of fault avoidance, fault elimination, and fault tolerance. That is, faults must be avoided through rigorous system development, eliminated when they are discovered during development, and their effects tolerated during service [30]. Component- and node-level faults can be handled with techniques such as N-modular redundancy. These faults are referred to as *local* because they only affect individual components or nodes and, therefore, can be handled at a local level. The faults we are interested in are those that either cause or create potential for affecting significant portions of the network. We call these *non-local* faults. Tolerating non-local faults requires coordinated non-local action as they cannot be masked on an individual basis. They also cannot usually be fully masked and require that the system be placed in an alternate service state.

With this in mind, *survivability* has emerged as a concept that provides applications with the necessary framework to tolerate non-local faults. Informally, survivability is when a system has "the ability to continue to provide service (possibly degraded or different) in a given environment when various events cause major damage to the system or its operating environment" [31]. Survivability is more rigorously defined elsewhere [31]. Non-local fault tolerance is one mechanism for achieving system survivability.

Implementation of non-local fault tolerance requires that errors first be detected and then treated through reconfiguration. Our approach to doing this is through a *survivability architecture* or *information survivability control system* [66]. Such a system senses network state, analyzes this state, and then actuates the network appropriately to enforce non-local fault tolerance policy. This approach to survivability is embodied in the *Willow architecture* [30].

Willow is a comprehensive survivability architecture for providing fault avoidance, fault elimination, and fault tolerance. Willow is based on the general notion of *reconfigurable applications*. These applications can be postured at runtime to meet the demands of the environment. Sensor events are gathered from both the application and other monitors such as intrusion detection systems. These events are input into the SPARTAN distributed analysis system and used to detect non-local faults. When a fault of interest is detected, SPARTAN enforces the survivability control policy by reconfiguring the network using the ANDREA system. ANDREA allows for intentional

command, scalable assessment of command success, and command conflict resolution.

In many fault-tolerant systems, the policy of the system is directly embedded in the mechanism at a design/implementation level. Fault-tolerance algorithms are usually described informally using natural language or pseudocode and then hand-coded into an executable implementation. This approach makes the fault-tolerance algorithms difficult to understand, analyze, and modify. For a typical large-scale survivable application there are thousands of complex non-local faults and reconfigurations. An ad hoc approach to this definition would quickly make modification of the policy difficult; therefore a different approach must be taken.

This different approach is *synthesis from formal specification* [29]. The policy of the survivability control system, i.e., which faults require which reactions, is first specified in a formal policy specification language. This specification is then translated mechanically into an executable implementation. This allows us to "describe what the system must do without saying how it is to be done" [64], thereby cleanly decoupling policy and mechanism. The policy can then easily be understood and modified apart from its implementation. This thesis is about a language, TEDL, that has been designed for the specification of survivability control system policy.

The remainder of this thesis is organized in the following manner. Chapter 2 presents a more complete description of survivability and detailed discussion of the Willow architecture as it relates to survivability. Chapter 3 describes the requirements for a language to specify survivability control system policy for non-local fault-tolerance. Chapter 4 details the syntax and semantics of TEDL. Chapter 5 then argues how TEDL meets the requirements set forth in Chapter 3. Chapter 6 describes our experimentation of using TEDL to describe non-local fault scenarios in a real distributed application. Chapter 7 presents work related to this thesis. Chapter 8 concludes with a summary of the work presented and proposed future work.

Chapter 2

Survivability and the Willow Architecture

Oh no not I! I will survive! Oh, as long as I know how to love, I know I'll stay alive. I've got all my life to live, And I've got all my love to give, I'll survive, I will survive! Hey hey... – Gloria Gaynor

2.1 Survivability

Survivability, like love, is serious business. The systems that require the property of survivability prevent modern society from falling into chaos. They run the electric power grid, control military defense systems, support the transport of food from producer to consumer, and prevent aircraft collisions. The loss of the service provided by any one of these can have financial consequences and possibly endanger human life. People are generally unaware of many of the systems they rely on because they rarely must do without them. For instance, if the national rail transportation system were made unavailable, food could not be transported efficiently and coal would not be available to power plants. If service were out for more than a few days, the economic impact would be immense and possibly irreparable. These events could all stem from the failure of a critical infrastructure system of which many people are completely oblivious.

That said, widespread catastrophic failures have a low chance of occurrence. A nationwide power outage has never occurred and only a few regional outages have occurred. Network worms have attacked the Internet many times over the past several years, but none has caused major dam-

age. However, the potential for catastrophic failure exists and therefore necessitates careful failure recovery planning. Survivability gives us a framework for describing and providing for the dependability requirements of these critical infrastructure systems.

2.1.1 Description of Survivability

Within the context of dependable systems in dynamic environments, survivability has emerged as a concept that provides a structure for describing the necessary system service requirements. Dependability is not a rich enough model for defining necessary properties of these systems because there is no explicit concept of degraded or different service. Under dependability, service is treated as a binary concept – either service is provided or it is not. Survivability, instead, deals explicitly with varied service in response to varied environment. For instance, we may want to vary the security and functionality of a system. When the system is not under attack, we want to provide full service. If the system comes under attack, the operating mode must be changed such that resources are redirected from providing service to defense. Another example is major damage to a command and control system where the non-damaged resources must be dynamically reconfigured to provide some specified level of service.

Great care is usually taken to avoid introducing design faults and minimizing degradation faults in critical distributed systems. Design faults are carefully prevented using a variety of rigorous system and software development techniques. Techniques such as formal methods are used to minimize the introduction of faults in the specification phase. Depending on their purpose, systems are implemented in safety-oriented languages such as Ada or security-oriented languages such as Java. Static analysis can be performed to find faults before execution. Systems are extensively tested and verified using formal proof techniques. Hardware degradation faults are prevented with rigorous manufacturing techniques and extensive testing.

Even with these techniques, deployed systems will still have faults. These faults must be handled through a combination of fault avoidance, fault elimination, and fault tolerance. Faults would ideally be eliminated when they are discovered by updating the running system. Some cannot be eliminated due to runtime constraints, so these faults must be avoided by reconfiguring the system

such that the fault is never activated.

Some faults inherently cannot be eliminated or avoided, so they must instead be tolerated [30]. Fault tolerance is a mechanism by which the property of survivability can be ensured. For instance, network worm attacks are an inherent consequence of operating in an open environment. The threat of these worms cannot be avoided or eliminated completely, but they can be tolerated using emerging techniques [72] that dynamically modify the environment. This modification not only results in defense against the attack, but also changes the service provided to users. This type of response is the fundamental concept behind survivability.

Informally, survivability is a system property in which [30]:

1. the system must provide complete service a fraction of the time
2. the system must provide reduced or different service if it cannot provide complete service due to failure or attack
3. the system must have several sets of service modes that correspond directly to the character of the failure or attack

Survivability is not just "graceful degradation." This would imply that the *only* course of action is to simply reduce service. Reduction of service is one possible response, but survivability places explicit importance upon the provision of *alternate* service. The survivability control system can be programmed to reconfigure the application to provide this alternate service. These alternate service modes are precisely defined by the user of the system such that they can be automatically effected under specified circumstances.

Component- and node-level faults are referred to as *local* because they only affect individual components or nodes, and can therefore be handled at a local level. A survivability control system is only concerned with faults that affect significant portions of the network. These are known as *non-local* faults. Non-local fault tolerance is much more difficult than local fault tolerance because it requires coordinated non-local action. These faults cannot be masked on an individual basis because they only exhibit non-local effects. In addition, they cannot usually be masked fully and

instead require that the system be placed in an alternate service state. Examples of non-local faults include extensive physical damage, widespread power failure, common-mode software defects, and coordinated, multiple-target security attacks.

Three critical issues in non-local fault tolerance are the abilities to detect [30]:

- Fault sequences - multiple faults that arise in some grouping. One fault may occur while another is being handled, or two faults may indicate a different fault condition than either fault individually.
- Fault hierarchies - a refinement of detection where a more generic fault condition is initially detected and then additional information leads to a subsequent refinement of the detection to a more specific fault.
- Interdependent application faults - multiple separate applications may interact in a manner that requires that events from both are used to detect a single non-local fault. One example is the dependence of telecommunications networks on the electric power grid. Failure of the power grid may cause telecommunications network failure even though there is no fault directly with the telecommunications network itself.

These fault types constitute a majority of non-local fault of interest and are difficult to detect effectively.

2.1.2 Motivating Applications

Critical infrastructure applications have many properties that are difficult to manage effectively. The most relevant of these are scale, complexity, criticality, availability, reliability, and security.

Scale The applications that run critical infrastructure system are very large. They typically contain between 10,000 and 100,000 individual nodes. For instance, the U.S. financial payments system has more than 19,000 member banks and 12 Federal Reserve banks. In addition, a next-generation military global command and control system could conceivably have an

application node for every one of the 3.4 million active duty, reserve, and civilian personnel and the many thousands of computers and weaponry.

Complexity The applications that run critical infrastructure systems are very complex. They typically consist of millions of lines of code. Individual software nodes usually must interact to perform a necessary function. For instance, the U.S. financial payments system operates in a hierarchic manner that requires multiple nodes to communicate in order to route a single transaction. These applications typically perform complex processing at each node that is difficult to even program functionally correct. These nodes are not only functionally complex, but they often also exhibit emergent behavior that cannot be simulated and is only apparent when the application is deployed. This complexity results in a large variety of possible non-local faults.

Criticality These applications are essential to the continued operation of critical infrastructure systems. The infrastructures usually cannot function at all without the services of the information systems. These information systems are usually the weakest point of critical infrastructures because of their relative fragility compared to the electro-mechanical components. It is essential for these information systems to meet their service requirements, as even brief outages can result in large losses.

Availability At any given time, there must be a high probability that the services provided by the information system are available. As long as the the application is usually available, short interruptions in service might be acceptable. For instance, the telephone network is usually available, i.e., when you pick up the phone there is a dial tone, even though it experiences a high number of very short service interruptions. The financial payments system can handle periods of non-availability by buffering and retrying transactions, but the service must eventually become available again. However, even short periods of unavailability for the power grid are unacceptable. Exact availability requirements vary among systems, but all have relatively high availability requirements. This implies that a survivability control system must be able to detect and repair failures quickly to meet system availability requirements.

Reliability Reliability measures the probability that a service will be continuously provided over time. One example of this is the power grid, where power must be continuously provided or the service is not acceptable. Many critical infrastructures rely on continuous service by information systems; therefore, interruptions can cause severe consequences. Reliability implies that the survivability architecture must be able to reposture the application to prevent impending failure and repair faults before they affect service.

Security The information systems behind critical infrastructure are a high profile target for attack, generally because they are more fragile than the electro-mechanical systems that rely on them. They are also usually more accessible because they allow for remote, non-physical access. In order ensure correct operation, the confidentiality and integrity of the information systems must be maintained at all times.

2.2 Survivability Control Systems

Survivability control systems, or survivability architectures, are an approach to providing non-local fault tolerance in large distributed applications. They are characterized by a sense/analyze/respond control loop model [66]. The control system is discrete state because the nature of the controlled information system is discrete. Control is divided into multiple, parallel control loops, possibly with overlap in their sensing and actuation domains. The controlled application, the operating environment, and external "real-world" state are monitored. Events from these three sources are used for detection of specified, abstract non-local events. The control system then appropriately reconfigures the application according to the indicated fault condition.

2.3 The Willow Architecture

Willow is a comprehensive survivability architecture for providing fault avoidance, elimination, and tolerance. Willow was briefly described in Chapter 1. The individual components, as shown in Figure 2.1, are now described in more detail.

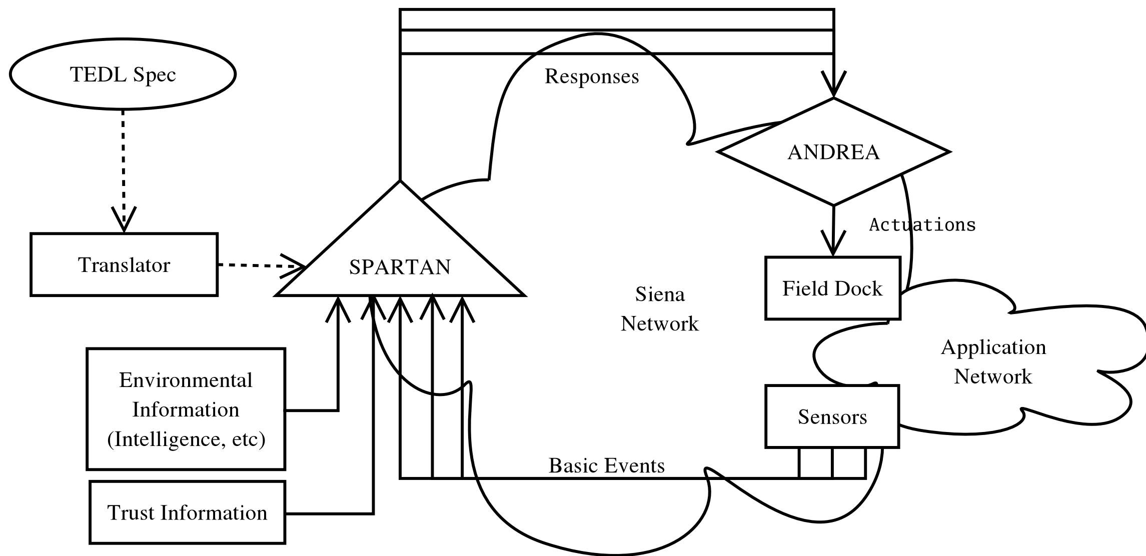

Figure 2.1: Willow Architecture

2.3.1 Reconfigurable Applications

Willow is based on the general notion of *reconfigurable applications*. Many current software applications are designed to be run as immutable, monolithic entities that may have some configurable parameters when executed. However, they are typically not designed to be reconfigured while running; this makes response changes in the operating environment difficult. Current research is focusing on frameworks for reconfigurable applications that will allow for scalable, consistent, and manageable reconfiguration.

2.3.2 Sensors

Sensor events are gathered from the application, network monitors such as intrusion detection systems, and offline intelligence. These events are then used to detect errors that indicate faults. The development of sensing technology is outside of the scope of the Willow project. It is assumed that sensors are provided by others and can be used for input into the control system.

2.3.3 Event Transmission

Sensors and detectors are decoupled, so sensor events must be transmitted from the sensors to the detectors. This must be done scalably and and independent of location. This is accomplished in Willow with the Siena content-based networking infrastructure [7]. Siena provides a publish-subscribe interface for sending and receiving messages. This interface allows sensors to transparently send their events to the appropriate detectors, without requiring that the sensors know where they are sending the events. Detectors can then subscribe to receive events based on the properties of the sensors and events without needing to explicitly name sensors from which to receive. In addition, this decoupling allows for transparent replication of detectors and transparent migration of detectors and sensors.

2.3.4 Error Detection Mechanism

Events emitted by sensors are then used for the detection of higher-level abstract events. Detection is modeled as a group of finite state machines. Events cause transitions in the finite state machines and detect specified network states of interest, i.e., non-local faults. The finite state machines are composed into a hierarchy where increasing higher-level abstract events are detected.

The original error detection component in Willow was RAPTOR. SPARTAN is the successor to RAPTOR and was used for the implementation of TEDL. SPARTAN is discussed further in Chapter 5

2.3.5 Error Recovery Mechanism

When non-local faults are detected, they require response. These responses detail the commands that must be executed to handle the indicated fault state. In RAPTOR, commands were sent directly to all nodes affected by the fault as listed in the centralized state database. SPARTAN instead enacts responses using an intentional command paradigm. In this paradigm, commands are sent to nodes based on the runtime properties of those nodes using selective notification. The ANDREA system is used to implement intentional command.

2.3.6 Synthesis of Error Detection and Recovery

The complexity and dynamic nature of the faults and attacks necessitates that the detection and recovery cannot be coded directly. Informal specification and hand-coding is slow and error-prone. Detection and recovery must instead be formally specified and then mechanically synthesized into an executable implementation. Such a specification language is the focus of this thesis.

In RAPTOR, detection Error Detection Specification (EDS) defines the finite state detection machines and the Error Recovery Specification (ERS). These both use the Z formal specification language. TEDL was designed to take the place of the EDS and ERS; TEDL specifications are used to synthesize executable instances of SPARTAN. Recovery commands are not yet a part of the TEDL language and are currently coded directly using the ANDREA API. They will be integrated into the language directly when a better understanding of the intentional command paradigm is gained.

2.3.7 Notification Mechanism

Non-local fault recovery is performed by reconfiguring the application using the services of ANDREA. ANDREA uses selective notification to target commands to only these nodes who have runtime state relevant to the command. There is no central database of nodes and node state, but instead, node state is completely distributed throughout the network. This provides for scalable command.

2.3.8 Actuators

Specific actuation will vary widely between applications. The general framework for actuation is provided by the Field Dock software [21]. Reconfigurations are received by the Field Dock associated with each application node and then enact local system state changes. Field Dock provides a uniform interface for universal node-level actuation.

2.3.9 Conflict resolution

The control model of Willow is a set of independent parallel control loops. This asynchronous control makes it highly likely that control loops will attempt conflicting actions. For instance, a widespread failure may occur while a routine software upgrade is occurring. The low priority upgrade must be suspended and the high priority failure recovery enacted. These types of conflicts are resolved where they conflict by ANDREA resource manager/priority enforcer [24]. ANDREA uses a prioritized reconfiguration list and dynamic resource management in a distributed workflow model to resolve conflicting requests.

2.3.10 Security

Control of applications networks allows for survivability, but it also presents a ripe target for complete application intrusion. Therefore, all components of the architecture must be protected. Security mechanisms for the publish/subscribe infrastructure are currently being researched. At the sensor and actuator level, techniques such as code obfuscation and randomization of behavior can be used to secure nodes [70, 30]. Trustworthiness of sensor event sources is assessed using a Trust Mediator architecture.

Chapter 3

Language Role and Requirements

> *A final hint: listen carefully to what language users say they really want, until you have an understanding of what they really want. Then find some way of achieving the latter at a small fraction of the cost of the former. – C.A.R. Hoare*

The traditional approach to fault tolerance has been to directly embed the fault-tolerant behavior policy of a system directly into its implementation. This severely limits the flexibility of the system in programming new or different behaviors in response to faults. The goals of the Willow project include the notion of using a formal language to define the survivability specification of a distributed application and using this to mechanically synthesize an executable control system.

In previous work on RAPTOR, the Z formal specification language was used for policy declaration. Z is a general-purpose specification language and does not map well to the sort of structures that must be described. Therefore, TEDL was designed as a customized language that provides, precisely and efficiently, the necessary language constructs for survivability control system specification.

Language design is complex and should not be undertaken lightly nor without careful thought and development of the language requirements. In this chapter, the requirements for the language and the motivation for those requirements are documented. The requirements can be grouped into five categories: scalability; expressivity; ease of use; implementability; and analyzability. Each of these categories is discussed in turn.

3.1 Scalability

Scalability in this context refers to the ability of a language to efficiently handle scale in all the dimensions of interest. In this case, the dimensions of scale are:

1. Number of nodes: Typical large-scale critical infrastructure information systems range in size from several thousand to a million nodes. Approaches based on global shared state cannot scale to these dimensions, so another approach is necessary.

2. Number of different node types: It is likely that any significant distributed application will have many different types of node. These different types of node must be handled efficiently with good abstraction of detail. The number of node types might reasonably be expected to range from several dozen to several hundred.

3. Number of elements involved in a non-local fault: The faults with which Willow is expected to deal will usually involve a large number of system elements of many different types. Thus, for example, a fault might involve processing, storage, communications, or software components. Scalability in this case is concerned with the number of components and the types of these components. For the non-local faults of interest here, the number of components might range from a few tens to a few hundreds, and the component types might range from one to tens.

4. Number of programs in application of interest: Distributed systems that provide essential services to critical infrastructure systems create those services by composing functionality from a variety of interacting programs. In major financial systems, for example, it is common for several thousand programs to coexist and interact during routine operation. The number of programs in the applications of interest might reasonably be expected to range between a few hundred and several thousand.

5. Amount of data communicated and the amount of computation consumed: Even with very low sensor event rates, a million components will generate an enormous number of events. These events must be handled in an efficient and scalable manner. The language must have

mechanisms to provide decentralized detection such that the communication of events and computation of detection are efficiently distributed.

3.2 Expressivity

The language must be sufficiently expressive to allow all non-local faults of interest to be specified. At the highest level, the critical criterion is the ability to express the complete details of all non-local faults of interest, together with the entire range of desired responses. It must be possible to define these specifications precisely, completely, succinctly, and unambiguously.

The expressivity requirements are broken down into the following five categories:

1. Basic semantics: There are a number of basic semantics implied by the functional requirements that must be included in the language. These basic semantics include the description of system states of interest, the changes in system state over time with real-time as well as temporal descriptions, and arbitrary combinations of state elements.

2. Category of non-local fault: Fault sequences, fault hierarchies, and interdependent applications must be handled. These fault types were described in Section 2.1.1. These are three fundamentally different types of complex, non-local fault and it must be possible to specify each type.

3. Scale: All of the dimensions of scale discussed in the previous section must be definable throughout their entire range. The language must allow for compact and efficient specification of the survivability of large systems.

4. Specification size: It should be possible to specify the policy of a given system succinctly and in a way that makes it easy for a reader to grasp. It is usually possible to construct the same specification in a variety of languages. However, the resulting specification must be a reasonable size and clearly understandable. Otherwise, the notation cannot be viewed as having appropriate expressivity. This requirement may conflict with other elements of expressibility, but it is essential for producing manageable specifications.

3.3 Ease of Use

The language must be easy to use. Therefore, the language syntax must be simple and easy to read. The semantics of the language must foster a clear mental model of the specified control system action. There should be a clear mapping between the syntax and the semantics. The language must be flexible enough to allow simple scenarios to be easily described. Complex scenarios must not require obtuse syntax crafting or an exponential increase in programming work. The language should allow for modularity and encapsulation so the specification can be modified without extensive rewriting. Specifications must be concise to allow for easier comprehension and modification. The language must be declarative rather than imperative, such that there is decoupling between what should be detected and how it should be detected.

3.4 Implementability

Clearly, any language design that is to be useful must be implementable. Implementability for the language being discussed here is significantly different from implementability of a conventional programming language. The object language in this case is a high-level programming language, and there must be a clear translation path from a specification to an executable implementation.

In addition, it is important to keep in mind the requirement for real-time semantics, the evaluation of expressions describing system state, and the efficient implementation of whatever data structures are used to describe system state in a specification.

3.5 Analyzability

Since the language under discussion is formal, it is amenable to analysis as well as synthesis. Given that opportunity, it is important that the language design include consideration of the potential for analysis that will allow useful properties of specifications to be established.

The types of analysis that might be undertaken include:

1. A set of type analyses based on the inclusion of a suitable type structure in the language definition.

2. Checking completeness of the use of various types, such as node types.

3. Reachability analysis of the various state descriptions and state change descriptions that constitute the various fault definitions.

4. Analysis of oscillatory behavior and terminal states.

3.6 Summary

There are few languages that have even attempted to fulfill the requirements set forth in this chapter. Many languages related to policy specification focus heavily on precise expressivity and have no allowance for scalability. In the next chapter, we present the syntax and semantics of TEDL, the Time-based Event Detection Language, designed to fulfill all of the necessary requirements for a survivability control system specification language.

Chapter 4

Language Design

> *An alternative approach lies in the creation of extremely simple languages, languages with very little built in. It is hoped that the simplicity of the tool will contribute more to the reliability of the product than would the omitted features. – D.L. Parnas [50]*

The syntax and semantic model of TEDL allow for manageable construction and modification of complex survivability control system specifications. The TEDL model of non-local fault detection is that of a group of finite state machines. Sets of temporally associated events are used to predicate transitions in the detection model. Each finite state has an associated response that enforces the appropriate element of survivability policy.

The most important aspects of the TEDL language are:

- Event-driven architecture instead of a complete system state model.
- Modular specification construction using declarative object-oriented classes.
- Finite state machine model of detection (Figure 4.1).
- Hierarchy of finite state machines for scalable detection.
- Action predicated on dynamic sets of sensor events rather than single events.
- Response based on intentional command paradigm for scalability.

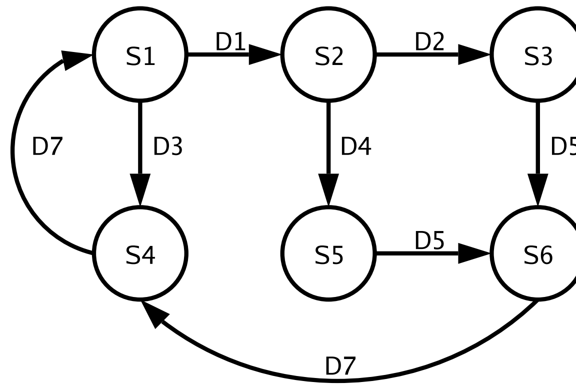

Figure 4.1: Example Finite State Machine Detector

4.1 Declarative Object-oriented Constructs

Many existing policy languages have syntax that is difficult to write, read, understand, and modify. There is little or no mechanism for abstraction and the syntax is not conducive for understanding the semantic meaning of a specification [38]. Declaration of policy elements is not modularized, so multiple, different semantic levels must be understood simultaneously [13]. The ability to abstract away detail and focus on single conceptual levels is essential for a maintainable and scalable language.

Many general purpose formal specification languages have syntax based on discrete mathematical constructs. These concepts are incredibly powerful but not intuitive for a majority of users. The syntax of TEDL is designed to have a similar look and feel to the Java object-oriented language. The declarative object-oriented syntax and semantics of TEDL, as opposed to a more mathematical notation, makes comprehension easier for many people.

TEDL consists of a set of class primitives that are declared as named objects with specific attributes. These objects are then composed to form control system specifications. Naming these objects and referencing them during composition allows abstraction of details; this allows a specifier to focus on fewer semantic levels in each part of a specification. Inheritance of declared objects is allowed for several classes to decrease redundancy. For other classes, inheritance only obfuscates their behavior and is, therefore, disallowed.

An example of the object-oriented syntax is seen in the **predicate** class. This declaration is presented only to give an idea of the syntax. All classes in TEDL have similar syntax to the predicate class. A predicate object is declared as:

```
predicate MyPredicate {
    SetOne > SetTwo ;
}
```

The meaning of this class will explained in detail later. This example declares a predicate named MyPredicate that is true if the cardinality of SetOne is greater than the cardinality of SetTwo.

The syntax for object inheritance is similar to Java and uses the **extends** keyword:

```
set ParentSet { set description }
set ChildSet extends ParentSet{ more set description }
```

In this declaration, ParentSet is declared with some properties, and then ChildSet inherits those attributes and then adds properties.

4.2 A Model of Time

Our specification approach includes time as an integrated notion. Unfortunately, many formal specification languages, such as Z, have no inherent notion of time. Timing information must be added as a postcondition on operations based on a special variable that is assumed to be a clock [65]. In RAPTOR, time is expressed this way and was difficult to use effectively [14]. The event-condition-action language PDL [38] takes the approach of treating time as simply another attribute of events.

Time is so important to event detection that it is a carefully considered notion in TEDL. Time is a formal, precise entity in TEDL, that is based on a simple approach to the problem of synchronization and time measurement. The crucial problems in dealing with time as an entity in specification for a distributed system are: (1) that the different components in the system will have clocks that can never be synchronized precisely thereby making the notions of simultaneity and time difference difficult to deal with; and (2) that delays in processing and transmission always make time associated with events at one location difficult to determine at another thereby making the problem of non-synchronized clocks much worse. To solve the first problem, it would be necessary to operate a

distributed system either with a single clock or with very tightly synchronized clocks, and to solve the second, it would be necessary to determine transmission delays very accurately. Neither of these approaches is possible for the types of system of interest.

The basic approach in the TEDL model of time is *to assume that these problems do not exist*. In other words, to assume that all clocks in all network components are fully synchronized and that transmission delays are zero. Clearly, this assumption is not valid. However, it is a clear and easy-to-understand assumption that TEDL programmers can *build on* in the preparation of TEDL specifications. The reason that the assumption is made in TEDL is because we are not aware of any other reasonable assumption that could be made which would allow the user of the language to control the problems that time raises. With this assumption, the specifier is both aware of the assumption because it is defined in the TEDL semantics and aware that, for his/her system, the assumption is invalid. Thus, when constructing a specification, the specifier can take into account the degree of drift in system clocks and the delays in transmission for his/her specific system in the specification by adjusting specific time values to meet the known system characteristics. In this way, any system timing characteristics can be accommodated because there is no attempt within TEDL itself to provide a "one-size-fits-all" model of reality. Specifiers are given an ideal model and they can adjust the meaning of TEDL statements to reflect the reality of their own systems by adjusting any time references suitably.

As an example, consider a system in which a specifier is interested in a series of related events that he/she expects to occur within some set of network nodes over a period of five minutes if the network is under a particular form of attack. The specifier knows: (1) that the events will be timestamped using component clocks that are skewed; (2) that the event notifications will arrive with realistic delays; and (3) that the interpretation of all timestamps by TEDL will be using the ideal time model. Thus, to specify such an attack, the specifier will have to use time intervals longer than five minutes to accommodate the known errors, and will have to understand that decisions made in the SPARTAN detector (the TEDL output) will have all of the uncertainty that is inherent in real timing situations. The great benefit that the TEDL model gives is that the specifier has as much control of the problem as can be provided, and he/she has a precise model of exactly what the

SPARTAN detector will do since it implements the TEDL semantics.

In our experience, the event patterns used for non-local fault detection are usually of the form "10 A events occurred in the last 20 minutes and 5 B events occurred in the last 10 minutes." These patterns are of thresholds of the number of events that must have occurred in a certain time period relative to when the pattern is evaluated. In TEDL, these time periods are referred to as *time windows*. Time windows are declared as an amount of time, e.g., 10 minutes, and a description of when the window ends as an offset from when the time window is evaluated, i.e., the present. As the time windows move ahead with the real progression of time, the windows are evaluated against independently timestamped events to determine what events are in what windows. Inclusion is determined based on the ideal time model that TEDL uses and so, in general, will not be accurate with respect to the clock of the entity evaluating the time window. As noted above, the burden of dealing with the difference between ideal and actual time semantics is placed on the specifier. The specifier must correct for the fact that evaluation of the windows is not instantaneous and takes computational time proportional to the size of the window and the number of candidate events.

This model of time provides a convenient model and is the only viable alternative to infeasible real-time semantics. It is also not a significant restriction because time windows will be large relative to clock skew, and non-local faults will usually go far above minimum detection thresholds.

4.3 Event-Driven Architecture

The approach to fault detection in previous work on RAPTOR [14] was to hold complete network state information in a centralized database. Queries were performed on this database for non-local fault detection. This approach does not scale to large systems because the communication and computation requirements are too great and the state model will always be inconsistent with actual system state. The SITAR system [71] for intrusion tolerant applications requires consistent global shared state as provided by reliable atomic multicast, distributed shared memory, or Javaspaces. However, these approaches to global state also will not scale to large distributed systems. Therefore, a different approach must be taken.

The approach behind TEDL is to rely on sensor events to provide small subsets of the actual complete state of the network. Only state relevant to detection and synthesis of response at a detection node is maintained at that node. This minimizes the amount of information that must be contained in any single node of the detection network. Events also hide specific properties of the monitored nodes not relevant to detection. Non-local fault detection is thereby mapped to detection of event sequences instead of detection of specific configurations of perceived system state.

The three required attributes of an event instance are a *type*, a *source*, and a *timestamp*. Event types may be declared with arbitrary attributes describing additional information. Instances of these event types are assigned values by the generating entity.

For example, the following event type would be assigned a source, timestamp, region, and severity when an instance of it is generated.

```
event IDSAlarmEvent {
    region ;
    severity ;
}
```

An instance event of this type could then be:

```
{ type="IDSAlarmEvent" source="node180" timestamp="1052590255"
    region="NorthWest" severity="8" }
```

4.4 Inherited Hierarchy of Event Types

A hierarchy of inherited event types provides a convenient mechanism for handling diverse types of events [48]. This hierarchy has the same semantics as a programming language class hierarchy through is-a relationships. For example, we could have an inheritance hierarchy as in Figure 4.2 where an DoSAttackEvent is-a AttackEvent and both LandDoSAttackEvent and SmurfDoSAttack-Event are-a DoSAttackEvent. The purpose of this hierarchy is to allow generic description of attacks or failures that are composed of certain classes of events, regardless of the specific event type. This hierarchy does not indicate any semantic relationship between events, as described by the detection hierarchy (Section 4.7).

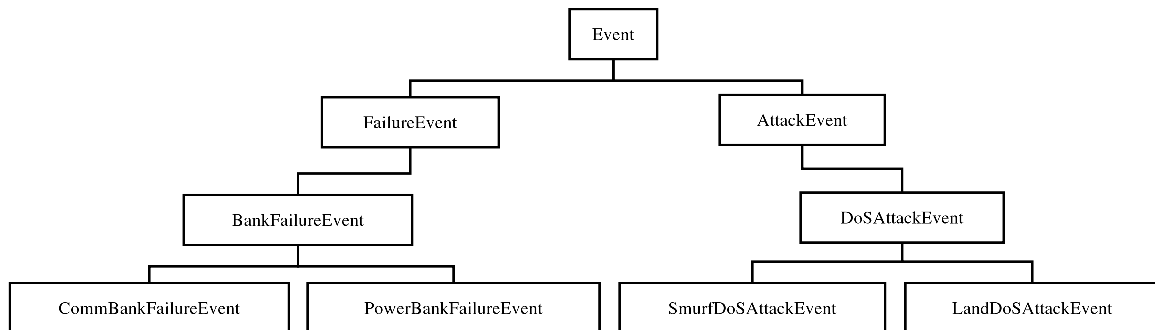

Figure 4.2: Event Inheritance Hierarchy

TEDL provides a mechanism for describing these inherited hierarchies of event types. The syntax for a declaring the event hierarchy uses the **extends** syntax similar to Java. The following code declares the previously described hierarchy.

```
event AttackEvent { }
event DoSAttackEvent extends AttackEvent { }
event SmurfDoSAttackEvent extends DoSAttackEvent { }
event LandDoSAttackEvent extends DoSAttackEvent { }
```

4.5 Dynamic Sets of Events

Many event-condition-action policy languages are only designed for few-to-one mappings between events and the associated required responsive action. Ponder [40] obligations allow multiple events to trigger action, but there is no easy way to describe the time relationship between these events. Describing time relationships between single events in PDL [38] is obtuse, and describing relationships between groups of events is even more difficult. Detection of non-local faults will always involve multiple events over time, so clear language mechanisms for this are essential.

Detection in TEDL is based not on single events but, rather, on logical sets of temporally qualified events. These sets specify membership qualification declaratively. The most important of these qualifications is time. All sets have a specific time window (as described previously) in which all of their events must have occurred. This semantic provides a clear and concise mechanism for describing sets of events over time. Sets significantly decrease the complexity of event sequence specification and maintenance. They are a key element for allowing scalable specification. The

notion of time is contained within the set concept and, therefore, preserves the untimed semantics of the finite-state-machine detection model, as described in the next section. The use of sets as the basis of detection reduces the precision of specific event sequences that can be described, but it is essential for scalable event specification.

The **set** class is used for set declaration. The following code declares an example set of events:

```
set SetOfEventA {
    EventA in 20 min offset 0 min;
}
```

The first item is the name of the set, `SetOfEventA`. The type of events in the set is `EventA`. Sets are restricted to having a single event type for simplicity. However, any event that is a subtype of the specified event type in the event inheritance hierarchy will match the type specification. This restriction is not overly limiting because sets can be composed together using the predicate mechanism, described later.

The value specified after the **in** keyword is the size of the time window with a time unit. Valid time units are msec, sec, min, hour, and day. The **offset** is how much time the leading edge of the time window is offset from the present. A zero offset means the window ends at the time the set is evaluated, i.e., the present. Offsets are useful for describing changes in event rates over time.

As real time progresses, the sets are evaluated as often as possible for membership and events are added or removed according to set membership requirements. The implications of the model of time used in these sets was discussed previously. The rate at which the sets are updated depends on the computational resources available, the complexity of the set specification, the number of candidate events, and the implementation. Note that even this computation delay is accommodated by the TEDL model of time. The specifier has to understand the impact of realistic calculation times for set expression.

Sensor events may be emitted from anywhere in the application network and any element can emit any number of events pertaining to the same attack. It is therefore necessary to have a mechanism to distinguish between these events. This is done with the **unique** modifier. A set specified with the unique modifier makes multiple events with the same *source* attribute only be counted once in a set.

```
set DoSEventsIn10min {
    unique DoSEvent in 10 min offset 0 min;
}
```

This declares a set *DoSEventsIn10min* that is the set containing all events *DoSEvent* from unique sources that have been received in the previous ten minutes.

Additional constraints can also be specified on event inclusion. These are done with the **qual** keyword.

```
set DoSEventsIn10min {
    unique DoSEvent in 10 min offset 0 min qual OS = Linux ;
}
```

This set is the same as the previous one with the additional constraint that the "OS" attribute of the DoSEvents must be equal to the string "Linux".

Sets can inherit other sets and change their qualifications using *extends* and *super*. The declaration of an inherited set uses the syntax of "extends parentname" after the set name, and then contains the keyword "super" in the declaration body. This is useful for encapsulating common properties of related sets into a single place. The common specific properties are then included in the inherited sets.

```
set DoSEventsIn10min extends DoSEventsWindow2 {
    super offset 30min ;
}

set DoSEventsOnImmunixBoxes extends DoSEventsIn10min {
    super qual LinuxDistro = Immunix;
}
```

4.6 Finite State Machines for Detection

The semantic model of detection is a collection of finite state machines that use dynamic sets of events to predicate state transitions. An example of this can be seen in Figure 4.1. Each state in this figure is a specific abstract state of the application network, and transitions between states are based on perceived faults. In this example, a transition to state S2 from state S1 is based on the satisfaction of the predicate D1, where this predicate indicates some specified damage has occurred.

Each finite state machine detects specific abstract events and then takes action, either by emitting an abstract event or initiating application reconfiguration. This is different than the approach of most event-condition-action policy languages that predicate action only on sequences of events and have no notion of state. PDL represents state by translating it into an event and including this event in detection sequences. This is a confusing semantic for specification of state. While at some level these approaches are algebraically equivalent, the finite-state-machine model for detection is much easier to comprehend, especially with visual depiction.

To transition in a TEDL finite state machine, a predicate must be satisfied. These predicates are based on the event sets. The semantics of the sets and predicates are abstracted away from the finite state machine. The machine is only aware of the condition (satisfied or not) of a predicate at a given time and knows none of the internal structure of that predicate. This abstraction allows us to easily integrate the notion of events that are relevant over time without affecting the untimed-semantics of traditional finite state machines. It also allows additional set or predicate description mechanisms to be added to the language without affecting the syntax or semantics of other language elements.

Predicates are either simple or compound. Simple predicates define either a comparison of the cardinalities of two sets or the cardinality of a set and a constant value. The relative operations $=$, $!=$, $>$, $<$, $>=$, and $<=$ are supported for comparison. Using the Boolean operators *and*, *or*, and *not*, these simple predicates can then be composed into compound predicates. This forces abstraction and encapsulation of predicates and makes them much easier to understand when reading a specification.

A simple predicate is declared:

```
predicate AttackOccuring {
    IDSAlarmsSet > IDSAlarmThreshold ;
}
```

A compound predicate is declared:

```
predicate LockdownTrigger {
    AttackOccuring and UnderOrangeAlert ;
}
```

Sets can also be negated when being used in compound predicates:

```
predicate WidespreadFailureNotRecovered {
    WidespreadFailure and not RecoveredFromWidespreadFailure ;
}
```

These predicate declarations are used to build finite state machine declarations with the **machine** class. The following declaration defines a simple machine that has a “normal” state which indicates full operation with no attacks or failures and a failure state when failure is detected. When the failure has been recovered, it transitions back to the normal state. States are described in greater detail later.

```
machine DetectWidespreadFailure {
    Normal + WidespreadFailureDetected -> WidespreadFailure ;
    WidespreadFailure + WidespreadFailureRecovered -> Normal ;
}
```

Each state in the machine corresponds to an abstract network state that requires detection and response. States and responses are described in Section 4.8.

The finite state model is convenient for describing survivability policy. Our goal is to vary application service in response to the network environment [31]. For instance, finite resources can be redirected from providing service to providing defense, depending on the level of attack the system is under. When not under attack, all services are available. Under localized attack, only essential services are available, and under intense widespread coordinated attack the system is shutdown completely to maintain integrity. As the attack worsens over time, we vary our response to it. An example of this is our Coordinated Attack Scenario in Section 6.2.2.

The finite state model is convenient for conducting formal analysis. Model-checking can be used to prove properties of the specification. For instance, it would be useful to be able to prove that all states in finite state machine description are reachable.

4.7 Hierarchy of Finite State Machines

Our finite state machines are not intended to be completely independent of one another. Instead, they are intended to be composed into a detection hierarchy (Figure 4.3). Machines in the hierarchy detect increasingly abstract events from events that lower-level detectors output. Individual

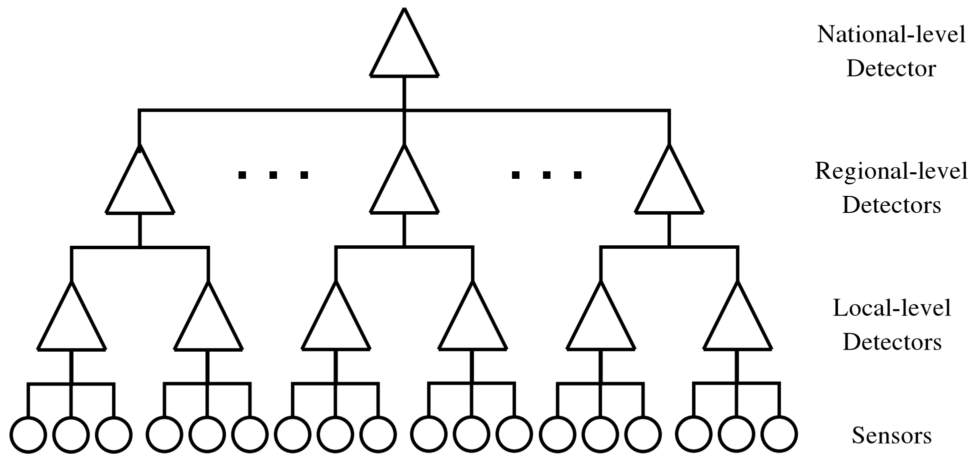

Figure 4.3: Example Finite State Machine Hierarchy

machines in the hierarchy take multiple lower-level events as input, use these to predicate their state change, and then output higher-level events or reconfigurations in response to state change. This decomposition allows us to easily describe the path from low-level network sensor events to high-level abstract event detection and reaction.

The hierarchy also provides for decentralization of detection. This is essential for scalability. Any centralized entity, i.e. a central state database, will quickly be overwhelmed by both the number of events it has to process and the amount of state it has to hold. In our distributed model, only state necessary for detection and response at any particular detection node is maintained at that node.

An example of the use of this hierarchy is the detection of geographic or topological non-local failures. In one machine, a local attack is detected, with one detector attached to each local area. These attack detectors then send their detected local-level attack events to their appropriate regional-level attack detector. All regional-level attack detectors then send their events to a national-level detector. This decentralizes detection to provide for scalability. At each level of detection we can vary the response to the perceived threat level. In response to a local attack, the local detector can make a relatively minor adjustment, such as shutting off a few services. When a nation-wide attack occurs, drastic action must be taken to prevent data loss and ensure future operational potential.

4.8 Reaction

When detection is successful, the system must react. This reaction is an application reconfiguration. Reconfiguration in TEDL is based on an intentional command paradigm. In our implementation, the services of ANDREA are used to provide an efficient interface to intentional command [24].

Our approach to intentional command is based on *selective notification* for sending commands only to those nodes whose runtime state is relevant to the command. Selective notification is explained in greater depth elsewhere [26]. The basic premise behind selective notification is that messages are targeted to receivers based on the runtime properties of the message sender, messages, and possible receivers. In this case, these messages are commands that select only nodes with relevant runtime state. Receiver nodes expose an “antigen” of relevant state, and then commands “bind” at runtime to only those node antigens that meet the command’s selection criteria. A simple example of this is that a command could be sent to perform a software update only to hosts that are running a particular defective version of that software, without requiring that the sender have a list of such nodes beforehand. Just as the event-based detection allows us to collect only relevant network state, intentional command enables the same for response. A complete state database is not necessary to target commands to relevant nodes, but rather commands are dynamically targeted to relevant nodes by the underlying communications infrastructure.

Responses to detection are associated with entrance to a state. The state class declares a state name and possible response. For example, a state “UnderAttackState” is declared:

```
state UnderAttackState {
    do RespondToAttack ;
}
```

The **do** element declares the name of the response that must be enacted upon entering the state. Responses then describe the event or actions that occur as a response. This provides another abstraction and allows multiple states to reuse the same responses.

The mechanism of action responses is implementation dependent. A formal description of responses was not added to to the language because the details of the intentional addressing paradigm

were not well modeled. When a better model is developed, a formal description of responses will be added. Responses are currently written directly in Java and compiled.

Action responses are declared as:

```
response RepairApplicationResponse {
    action RepairApplication ;
    SecurityLevel := 5 ;
}
```

In our implementation of TEDL, the action “RepairApplication” refers to an executable Java object. Runtime parameters are set in this object by declaring their name and value, as the case with “SecurityLevel”.

Responses can also be inherited. For instance, there may need to be multiple machines that perform similar actions, but have different specific parameters for their responses. This could be done as:

```
response RecoverFromWormAttack {
    action WormDefense ;
    sienaURI := "tcp:sirius.cs.virginia.edu:23456" ;
}
response WebserverRecoverFromWormAttack extends RecoverFromWormAttack {
    target_application := "WebServer" ;
}
response FileserverRecoverFromWormAttack extends RecoverFromWormAttack {
    target_application := "NFS" ;
}
```

This syntax is used in place of a function call-like mechanism to maintain orthogonality with other language elements. If the syntax were similar to that of a function call, the declarative nature may be obfuscated.

4.9 Machine Event Generation

The second type of response a machine can enact is an event response. This is the emittance of an event from a finite state machine that is used as input to a higher-level event detector.

Event responses are declared similarly to action responses. Instead of a **do** directive, the **output** directive sets the type of the event to output. Additional string value/attribute pairs can set attributes in the event instance.

```
response rEmitAttackEvent {
    output AttackEvent ;
    operating_system := "Immunix" ;
    region := "NorthWest" ;
}
```

4.10 Assessment of success and subsequent recovery

Assessment of the success or failure of a response is important. With communications mechanisms such as selective notification [24], one can send a message to an unknown and arbitrarily large number of hosts based on their properties. Individual replies are generally not required, and would quickly flood the system. This problem is solved with the concept of harvesting [25]. Harvesting provides a scalable, decoupled mechanism for receiving replies to selective notifications. The reply can be in the form of either individual responses, of which there may be millions, or a summary

histogram of the responses. Assessment of success is difficult because, in large-scale networks, the variability of reply time may be arbitrarily large. One must therefore describe a complex expected response model and then base assessment on the comparison of this expected model to the actual responses.

The normal finite-state mechanism is relied upon for assessment instead of a specialized language construct. Response states have transitions from them that indicate either success or failure. Events are fed back into the control system from the response and these events are used to transition into failure or success states. The primary difficultly with this is that expected responses from responding nodes must be predicted and then a response to unfavorable ones must be formulated. Additionally, a reconfiguration could possibly cause even greater damage to the system when attempting to fix it. This is an especially hard problem because one can get into a continuous cycle of attempts and failures.

4.11 Exception handling

When attempting a reconfiguration, actuations may fail or additional damage could occur while reconfiguring, prompting the need for an alternate plan. Exception handling in this case is similar, but not identical, to assessment of success and subsequent recovery. With exception handling, it is not that the reconfiguration did not work, but that something went wrong with the reconfiguration mechanism itself.

The low-level command execution of responses is currently implemented in Java code. All exception handling is done within this code. Exceptions can be handled by retrying the command or emitting a failure event back into the finite-state model. Future work will focus on evaluation of the intentional command paradigm for providing a useful exception handling model.

4.12 Transforms

Transforms allow arithmetic computations to be performed on the cardinality of sets. This is useful for comparing two sets that do not necessarily need to be equal, but must be within some tolerance

of each other.

Transforms are declared as:

```
transform t90Percent{ * 0.9 }
```

This would multiply the value of the cardinality of the transformed set by 0.9. Transforms are used by placing them before the set to be transformed:

```
predicate WidespreadFailure {
    PrevMinHeartbeats < (t90Percent) PrevMinHeartbeatsOffsetOneMin ;
}
```

This would say "the predicate WidespreadFailure is true when the cardinality of the set PrevMinHeartbeats is less than 0.9 times the cardinality of the set PrevMinHeartbeatsOffsetOneMin." Because set cardinalities are integer values, the type representation of the transformed set will also be an integer.

4.13 Summary

Together, these language elements provided a clear and concise language that efficiently fulfills the requirements for a control system policy language. In the next chapter, we present the results of using TEDL to specify example non-local fault-tolerance scenarios in an example financial payments system.

Chapter 5

Implementation

Implementation is the sincerest form of flattery – L. Peter Deutsch

In previous research with RAPTOR, the system was constructed as a simulation to demonstrate the feasibility of a survivability architecture. SPARTAN was, instead, implemented as a real system to further demonstrate feasibility. SPARTAN is a simple object-oriented framework for detection that is used by the translator to compose executable detectors. Close integration with TEDL makes synthesis from formal specification a mostly simple transformation. The object-oriented SableCC compiler framework [15] was used to construct the translator. The use of object-oriented paradigms in the syntax of TEDL, the SableCC compiler framework, and the SPARTAN implementation framework expedited system development.

Each node in our hierarchy of detectors corresponds to a single SPARTAN agent. These distributed mobile agents communicate events via a publish-subscribe network, in this case Siena [7]. In contrast to traditional point-to-point communication, publish-subscribe messages are not sent to a single host based on name, but rather to a group of interested hosts based on attributes of the message. This allows transparent flexibility in detector location, easy replication of detectors, and simple construction of the agent hierarchy.

The SPARTAN Java class design can be seen in Figure 5.1. The EventHandler thread communicates with the Siena network via the ThinClient interface. Events are received and put into the EventDatabase component, encapsulating a timestamp-ordered linked list of all received events. The EventHandler then signals the SpartanSetHolder that the EventDatabase has changed. The

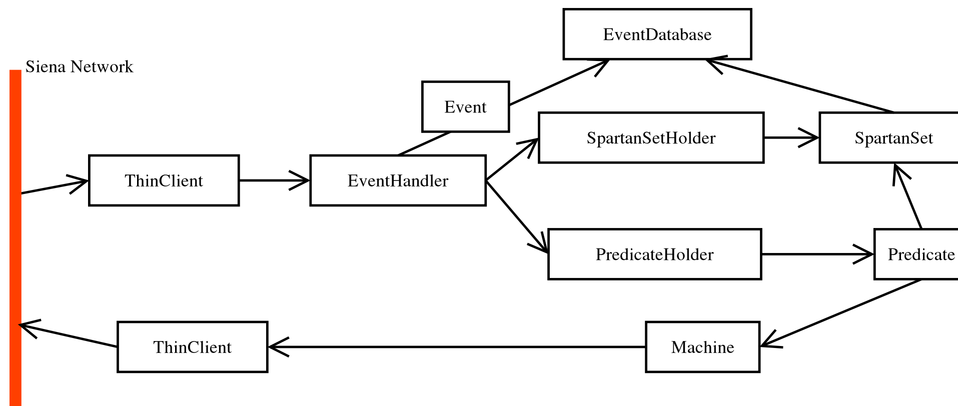

Figure 5.1: SPARTAN Implementation Architecture

SpartanSetHolder has references to all of the sets in the detector and updates them based on the current time and updated EventDatabase. The SpartanSetHolder then signals the PredicateHolder to update all predicates based on the updated sets.

Each SpartanSet represents a logical set declaration. Events must match this declaration to be placed in the set. Each Predicate then examines the cardinality of the SpartanSets it depends upon to determine if it is true or false. If it transitions from false to true, it notifies the Machine that uses it and then that Machine updates its state and takes appropriate action based on the Predicate. This action can either be emitting an event or the execution of Java code. This architecture is simple and presents us an easy to use framework for translation.

Currently, all sets and predicates are updated for every event. However, high event rates would quickly overwhelm this approach. A better approach would be to have a maximum and minimum rate at which the sets were updated, and any rate between these bounds the sets are updated for each event.

Reconfigurations rely on the ANDREA system to provide several key services. ANDREA is a set of services for highly scalable management of large scale application networks [24]. The first service SPARTAN uses is selective notification, a novel communications paradigm where receptor nodes expose an “antigen” of their state and messages “bind” to the antigen based on a selection function. With this, messages are routed only to receptors that match their selector. This service is used for reconfiguration in an intentional command paradigm. Instead of commanding nodes by

name, we command only those with relevant runtime state. A simple example of this is that we could send a command to only to hosts that are running a vulnerable piece of software to shutdown the software, upgrade it, and then restart the service. The second ANDREA service used is harvest. Harvesting is a scalable and decoupled reply mechanism for selective notifications. Using harvest, a selective notification can be sent to an unknown and arbitrarily large number of hosts and only receive a single reply containing a histogram of all receiver responses. This allows us to scalably assess the success or failure of a response. The final ANDREA service that SPARTAN uses is intention council conflict resolution. Each reconfiguration command is accompanied by an intent and a reason. The intent describes what the reconfiguration is designed to do, and the reason describes why it is doing it. Multiple commands with conflicting intents and reasons are arbitrated before affecting the application. This aspect of ANDREA is currently in development and is therefore only used nominally.

In the next chapter, we present the use of this implementation for executing several non-local fault-tolerance scenarios for a simplified financial payments system.

Chapter 6

Experimentation and Assessment

Language designers are not intellectuals. They're not as interested in thinking as you might hope. They just want to get a language done and start using it. – Dave Moon

6.1 Evaluation Challenges

The evaluation of a language is difficult. While one may make arguments on how a language fulfills desired goals, one cannot prove that a language fulfills them. One cannot run experiments directly on a language and gather quantitative data. However, we can use the language and qualitatively assess its usefulness. In this chapter, we present the results from our use of TEDL for describing several non-local fault-tolerance scenarios in an example distributed application. We then use the results of these experiments in the next chapter as part of our language evaluation.

With our current resources, we are limited to a network of several thousand application nodes running on approximately 100 physical machines that are part of the Centurion cluster. This application is nearly two orders of magnitude smaller than our theoretical target, but it is large enough to demonstrate that our approach exhibits at least minimal scalability.

Several scenarios have been devised to demonstrate that TEDL easily and efficiently describes real-world, non-local fault scenarios. These specifications have been constructed for scenarios involving a prototype financial payments system. The scenarios involve non-local fault detection and recovery that use the intentional command paradigm provided by ANDREA.

6.2 Financial Payments System Scenarios

Throughout this thesis we have argued how TEDL provides the properties necessary for a useful survivability control system specification language. However, the only true evaluation of a language is whether or not it allows users to easily solve real problems in their domains. Towards this, we have used TEDL to specify several example attack and failure scenarios. These scenarios are representative of the many non-local faults that can occur in large distributed systems. The results from the execution of each scenario are presented to show the correct function of the implementation and give the reader a general idea of the timing of detection and response.

These scenarios are for a simple financial payments system based on the United States financial payments system, the Federal Reserve's Fedwire System. Our simplified example system routes electronic money transfers between separate banks. The system is hierarchical and consists of three levels, as seen in Figure 6.1. At the top of the hierarchy are multiple Federal Reserve banks. Each of these has some number of Money Center Banks (MCBs) that use its services. Each MCB in turn has child Branch Banks. In our examples, the size of the network varies between scenarios because of resource limitations. More information about operational financial payments systems can be found in Elder [14].

In our system, the individual bank application nodes have been made reconfigurable such that they can be modified in response to attack and failure. These actuations range from practical and useful ones, such as the ability to change the bank's parent node and start and stop services, to more abstract actuations, such as specifying a "security level" to operate under.

The testbed for running these experiments was a set of 100 dual Intel Pentium II 400mHz machines with 512 MB of memory running Red Hat Linux 6.2. The machines are connected with switched 100BaseT Ethernet. The Sun JDK 1.4.1_02 was used to execute the Java bytecode.

In the remainder of this section, these three non-local fault scenarios are described. The TEDL specification for each scenario is presented in Appendix B.

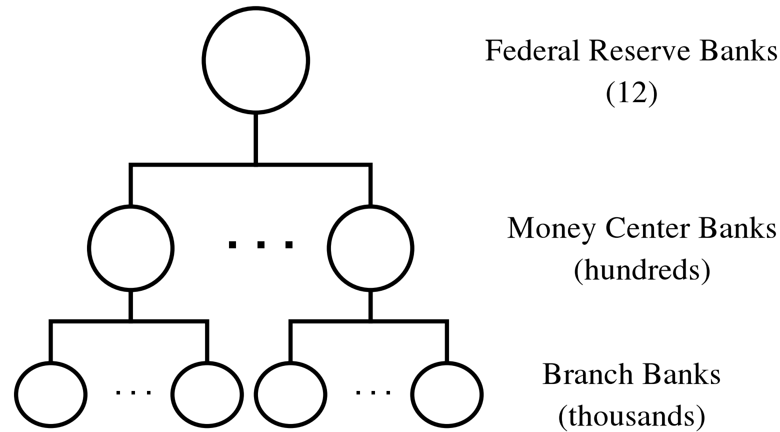

Figure 6.1: Example Financial Payments System Diagram

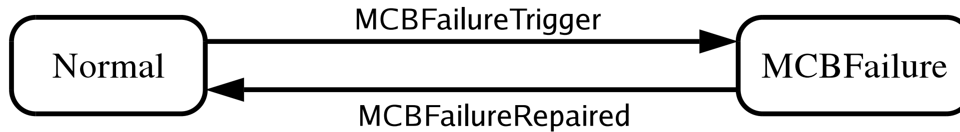

Figure 6.2: MCBFailureDetector State Diagram

6.2.1 Money Center Bank Failure

In this scenario, a Money Center Bank (MCB) fails and this failure is detected by transaction faults reported by its child Branch Banks, as specified by the MCBFailureTrigger. The state diagram for this is shown in Figure 6.2. The TEDL source is listed in Section B.2. All bank nodes are reporting their sensor events to a single SPARTAN detector node. After one of the MCBs (Richmond_MCB_000) is stopped to simulate failure, the child Branch Bank nodes of this MCB cannot transact with it. In response, each affected node emits a sensor event indicating this condition. After the SPARTAN detector receives 10 of these events, the predicate MCBFailureTrigger is satisfied and the detector transitions to the state MCBFailure. A reconfiguration is emitted that instructs all Branch Banks with one or more transaction faults to change their parent Money Center Bank to a different parent (Richmond_MCB_001) to repair the application network. In a more complex example, the response would attempt to find a MCB that has not failed and switch to that one, but we have simply devised a static response for simplicity.

In our tests for this scenario, there were 6,700 banks in the network. The results for five trials

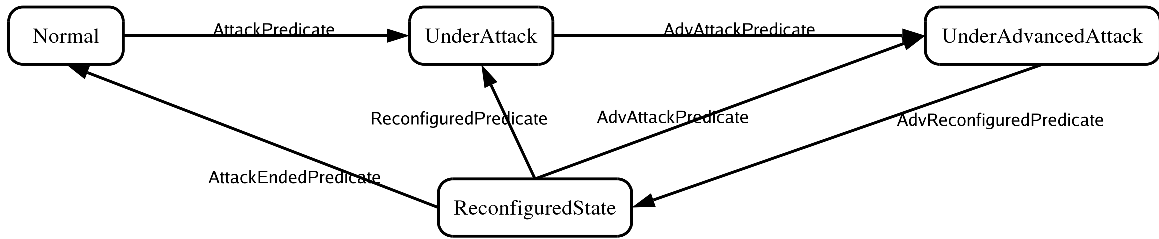

Figure 6.3: CoordinatedAttackDetector State Diagram

are shown in Table 6.1. The first column shows the amount of time from when the failure was caused until the first error event was received. The bank nodes attempt transactions once every three minutes, so this amount of time varies depending on where in that cycle the banks are. The second column shows the time at which the non-local fault had been detected and a reconfiguration emitted. The next column shows the time at which all of the reconfigured banks had responded to the reconfiguration, and the last column shows the total elapsed time from the first sensor event to successful reconfiguration. In this scenario, all 25 branch banks that should receive the command do and the network is therefore successfully repaired.

6.2.2 Coordinated Attack

This scenario demonstrates a coordinated attack where multiple bank nodes are attacked simultaneously. We assume that host or network-based attack sensors, like intrusion detection systems, exist to detect attacks at each node. The system correlates the attacks happening at multiple sites and then takes action that is different from the action that would would be taken with only local attack knowledge. In our scenario, the response is increasing an abstract "security level" at each bank and

Table 6.1: Results for Money Center Bank Failure Scenario (times in seconds)

	First Input Event	Reconfiguration emitted	All nodes reply	Elapsed Time
Trial 1	0.55s	0.57s	2.07s	1.12s
Trial 2	0.08s	0.14s	1.16s	1.08s
Trial 3	2.07s	2.10s	3.17s	1.10s
Trial 4	2.44s	2.48s	3.55s	1.11s
Trial 5	0.54s	0.59s	1.57s	1.03s
Average	–	–	–	1.07s

Table 6.2: Results for Coordinated Attack Scenario (times in seconds)

Event	Total Elapsed Time	Elapsed Time since last event
First event	0.0	0.0
Last event	13.7	13.7
Attack detected	13.8	0.1
Response complete	194.0	180.2
Adv. Attack first event	253.7	59.7
Adv. Attack last event	282.3	28.6
Adv. Attack detected	282.4	0.1
Response complete	462.6	180.2
Transition back to Normal	762.6	300

disabling all non-essential services at each bank to redirect as many resources as possible to defend against the attack. When the attack worsens, the security level is increased further and all services are disabled. This phased response allows varied service in response to escalating attack. The state diagram for this scenario can be seen in Figure 6.3 and the TEDL source is listed in Section B.3. While shutting down all the services is essentially a self-denial-of-service, we assume that this is the only way to defend against the attack adequately. This is assumed to be a major attack and drastic action is necessary to preserve essential service, prevent worse damage, and ensure quick recovery after the attack.

In our experiment, synthetic sensor events (DoSSecurityEvent and PortscanSecurityEvents) were injected into the SPARTAN detector using our Syringe publish-subscribe event injection tool. These two event types are representative of the myriad of possible events from real intrusion sensors. 100 DoSSecurityEvents and 100 PortscanSecurityEvents were injected to trigger a control system response. The control system then attempted to reconfigure the network to defend against this level of attack by setting an artificial "security level" higher and shutting down non-essential services. 200 more events were injected to trigger a more drastic response in the network. This response sets the security level even higher and shuts down all services to prevent further damage.

The results of a typical execution of this scenario are seen in Table 6.2. This table shows the amount of time that elapses between different actions in the scenario. The first two entries are the times of the first and last sensor event that is injected into the detector. The next entry is when these

Table 6.3: Command Response Times for Coordinated Attack Scenario (times in seconds)

Time from reconfig start (sec)	responders out of 3444
Attack State EssentialServicesOnly()	
0	0
15	0
30	1154
45	3435
60	3435
75	3435
90	3435
Attack State SetSecurityLevel(1)	
0	0
15	0
30	1154
45	3435
60	3435
75	3435
90	3435
Advanced Attack State ShutdownServices()	
0	0
15	0
30	1154
45	3435
60	3435
75	3435
90	3435
Advanced Attack State SetSecurityLevel(0)	
0	0
15	0
30	2763
45	3435
60	3435
75	3435
90	3435

Table 6.4: Response times for Selective Notification (times in seconds)

Time (s)	Total Responders	Remaining Responders	New Responders
0	0	3444	0
15	0	3444	0
30	856	2588	856
45	3041	403	2185
60	3435	9	394
75	3435	9	0
90	3435	9	0
105	3435	9	0
120	3437	7	2
135	3440	4	3
150	3441	3	1
165	3442	2	1
180	3444	0	2

events have been successfully used to detect the attack and initiate response. The next entry is time of the completion of the response. The next four entries describe the times of the second phase of the attack. The final entry in the table is the time at which the detector determines that the attack has ended and transitions back to the normal state.

Table 6.3 shows the response times to the four selective notification commands that are emitted. Each entry shows the amount of time elapsed since the command was emitted and the number of nodes that had responded. There were 3,444 nodes in this example and all should have responded to all commands. However, a few did not because all of the available hardware was saturated and they could not respond in time. This behavior mirrors the behavior seen in a typical large distributed application. After the execution of the scenario had ended, “Ping” bank command that was manually injected. The command was targeted to all nodes and simply required that the nodes respond. Table 6.4 shows the response times for this command. For intervals of 15 seconds, the table shows the total responders, the remaining responders, and the number of new responders since the last update period. The command was waited upon until all receivers had responded instead of stopping after after a specified amount of time. This pattern of response matches that from all other commands.

Figure 6.4: RegionalFailureDetector State Diagram

6.2.3 Regional-level Power Failures

The following scenario demonstrates the utility of the system for detecting topological or geographical failures. For instance, a power failure causing widespread outages in the application network requires detection and response. It is assumed that human intervention is required in these scenarios, so the results of detection are sent to an operator.

Here we have the nodes of the network divided into regional geographic areas. The nodes are partitioned according to their Federal Reserve Bank into the following categories:

- NorthEast – Boston, New York, Cleveland
- SouthEast – Philadelphia, Richmond, Atlanta
- NorthWest – Chicago, St. Louis, Minneapolis
- SouthWest – Kansas City, Dallas, San Francisco

A hierarchy of detectors is used in this scenario. For each region, we have a SPARTAN node that monitors power failure events from that region. When a certain number of events (in this case 250) are received, the node triggers a response that notifies a human operator that a widespread regional failure has occurred and sends a regional power failure event up the detector hierarchy to a national level SPARTAN node monitoring all the regional detectors. When this national detector receives two regional failure events, it then notifies a human operator that a national failure event has occurred. In this way, multiple local failures signal a regional failure, and multiple regional failures signify a larger national-level failure. As in the last scenario, synthetic events were injected using

Figure 6.5: NationalFailureDetector State Diagram

Syringe to simulate sensors events. The state diagrams for this scenario can be seen in Figures 6.4 and 6.5. The TEDL source is listed in Section B.4.

The execution of this scenario was successful. The timing information is not of interest, so it is not presented. Both regional failures were successfully detected, as was the national failure.

6.3 Summary

In this chapter, we have presented the results of our use of TEDL in specifying three non-local fault scenarios for a simplified financial payments systems. These scenarios are far from conclusive proof of the usefulness of TEDL, but they do give indication of such usefulness. Future work with TEDL will involve specification of larger and more complex scenarios to further evaluate its utility.

Chapter 7

Language Evaluation

The proof of a system's value is its existence. – Alan J. Perlis

This chapter evaluates the language with respect to the requirements put forth in Chapter 3. This section presents these requirements along with the language elements intended to meet them.

7.1 Scalability

Scalability is the most important aspect of TEDL. While the other requirements are important, scalability is the only one that can completely nullify the usefulness of the others. The elements of TEDL that provide for scalability are now discussed.

1. The number of nodes in the distributed application: Large numbers of application nodes are managed by having a hierarchical event-based architecture. A central database of network state would be unscalable, but the event-based architecture abstracts away the number of nodes in the network. Imperative command prevents the need to have a complete state database with all node names and state. Detection and response are therefore scalable to an arbitrarily large network.

2. The number of different node types in the distributed application: The event-based architecture abstracts away any specific properties of nodes. This allows an arbitrarily large number of node types without requiring individual specification of the details of these types. For

instance, in the regional failure scenario, arbitrary bank types could emit the power failure events without requiring that the control system know about each of these node types.

3. The number of elements that could be involved in a non-local fault: The event-based architecture and intentional command paradigm provide for this requirement. This requirement is difficult to fulfill, as when a fault is detected, we must repair exactly those nodes affected. A centralized state database would always be inconsistent with the actual state of the network and, therefore, nodes that change state after the commands have been emitted may either receive unnecessary commands or not receive necessary commands. However, an intentional command will always go to only those nodes whose actual runtime state meets the command requirements, so command targeting is always consistent with actual network state. With the event-based architecture, a reconfiguration can be enacted when the number of some type of event goes above a certain threshold, but the commands will still be targeted to any nodes that fail after the reconfiguration has started. In the Coordinated Attack scenario, an arbitrarily large number of nodes could have been under attack, as long as that number were above a certain threshold. The actual number involved does not matter, as the intentional command determines this at runtime. Alternatively, this command could have been targeted to only those nodes that were under attack at the time.

4. The number of programs that constitute the distributed applications of interest: Multiple different programs are abstracted away by the event model. Relevant information from each program is normalized into an event such that all data may be handled the same way for detection.

5. The amount of data communicated and the amount of computation consumed: By having a detection hierarchy, the communication and computation of detection can be distributed throughout a network. Only that state relevant to detection and response at any node in hierarchy is stored at that node, significantly decreasing the amount of data contained at any one node. Without this distribution, it a centralized structure would soon be overwhelmed with communication, storage, and processing. In the Regional Failure scenario, the hierarchy

could be arbitrarily large and the specification of the detection nodes would be exactly the same.

7.2 Expressivity

TEDL is designed to only support a small class of detections, but to do so easily and understandably. This expressivity limitation is one of the core design decisions in TEDL. With regards to these detections, the expressivity of TEDL is now discussed.

1. Basic semantics: System states of interest are described with the finite state machine model. Detection is not simply based on event-action sequences but, instead, on the more meaningful conceptual model of abstract application system state. All elements of the detection are continuously evaluated over time, so the abstract system state model is maintained to match the real system as closely as possible. The relevancy of events over time, as described by the set class, is essential because events have time relevancy in the real world. For example, local failures will usually be occurring in the network, but only groups of these occurring in a defined time span constitute a non-local failure. The event set and state elements of the language allow for arbitrary event sequences to be linked together without regard for the specific types of nodes affected or types of events involved.

2. Category of non-local fault: Fault sequences and fault hierarchies are easily described using the state model. Sequences can be composed using Boolean expressions to indicate state transition or can be used in a branching chain of state transitions. Fault hierarchies can be described easily with a sequence of transitions. Interdependent application faults can be described because any specific properties of the applications can either be encapsulated by events or abstracted away with events. These events can then be used in arbitrary ways to describe the interaction of the application faults.

3. Scale: The use of sets of events instead of individual events for detection allows scale to be expressed easily. If individual events were the basis, the specification would soon grow

unmanageably large and would require an exponential increase in programming work to describe them.

4. Specification size: The modularity of the language allows the specification size to be decreased. Elements can be specified once and then used multiple times. All detection elements are divided into small classes, so simple detections may be longer than some non-modular equivalent language. However, the modularity allows for many elements, such as machine objects, to be much smaller when describing complex detections. This leads to increased comprehensibility. Each individual class has a clear and concise syntax, so elements can be read and understood easily.

   It is difficult to compare the size of a TEDL specification to the size of the equivalent RAPTOR specification. RAPTOR specifications are more monolithic, in that all detection and responses are declared in the same file. The financial payments system scenarios presented by Elder [14] are about 40 pages of Z. Specification of only the Money Center Bank failure scenario would require about five of these pages, as compared with the one page of TEDL for the same scenario (Appendix B).

There are several limitations to the expressivity. Event rate changes over time are not easily expressible. If an attack or failure rate fluctuates over time, only the resultant events are evident and not the peak rates. It would be possible to declare a number of small, successive time-windowed sets, and then compare the cardinalities of these, but this approach is difficult because the size of the fluctuating windows would not be possible to determine before execution.

Another difficulty is the "summarization" of detection information in the hierarchy. Detectors in the hierarchy must emit discrete events, so there is loss of information going up the hierarchy. Summary information on the sets of events resulting in a higher-level event could be put in the higher-level events, for instance, the average value of some attribute. It is desirable to keep away from single event values, i.e., minimum and maximum values, as these may often be statistic outliers.

The lack of a full collection of set operators, such as union and intersect, is a limitation. The set cardinality is currently the only information about a set available to the predicates. Additional set operations such as intersect and union would be useful. Summary information on the sets, such as average values of attributes and histograms of attribute values, will also be useful for some detections.

7.3 Ease of Use

Ease of use is a difficult requirement to assess and is largely personal preference. This language was designed for use by people coming from an object-oriented programming background, whereas many formal specification languages are based on discrete logic and many policy languages have syntax similar to imperative languages. TEDL is both object-oriented and declarative, so it is easily understandable yet still provides decoupling between policy and mechanism.

The syntax was designed to be clear and aesthetically pleasing. The modularity of the language makes reading specifications easier than if they were large monolithic structures. The semantics of various elements are encapsulated within those elements, so changes can be easily made to one object while preserving its use by other objects. For instance, the notion of time is encapsulated within the set class, and is therefore abstracted away from all other elements. The sets are encapsulated within the predicate class, so new types of predicates can easily be added without affecting the machine declaration syntax or semantics.

The finite state model of detection fosters a clear mental model of detection. This explicit model allows for an easier to understand model than other approaches based on simple event-action policies and optional state. Non-local faults usually involve sequences of events, so the finite state model matches this well.

7.4 Implementability

A translator has been implemented to synthesize executable SPARTAN nodes from TEDL specifications. Even without any experience with compiler construction, the author was able to quickly

develop a translator and implementation framework. The event-driven, object-oriented language semantics are easily mapped to an event-driven, object-oriented implementation. The modularity of TEDL allowed the compiler to be constructed incrementally and made language modifications simple.

7.5 Analyzability

Current research is focusing on language analysis. The declarative nature of the language makes this possible. The finite state machine model makes model-checking a good prospect for analysis. The modularity of the language makes symbolic representation more clear. Future work will explore these aspects of analysis.

7.6 Additional Limitations

The lack of a formal response notation is a significant problem for the language. Even in the current implementation, the notion that a response takes time to complete and that the finite state machine may transition while a response is occurring. This indicates that the formal response language elements must allow for this. One approach is to put a minimum time that the machine must stay in a state, corresponding the the expected execution time of the response. In the implementation, responses are currently spawned as separate threads and the machine continues normal operation. These approaches will be evaluated further in the future.

Chapter 8

Related Work

... knowing about too many well-researched details can often confuse the real direction we have to take. I like writing new code, and I prefer to approach things from the physics side: take a few elementary rules and build up the 'one correct' solution, no compromises. This might not be as effective as first reading all the available material and then cherry-picking a few ideas and thinking up the remaining things, but it sure gives me lots of fun – Ingo Molnar

8.1 Fault-tolerant Distributed Systems

Cristian's Advanced Automation System (AAS) replacement for the Air Traffic Control (ATC) system presents a useful instantiation of his fault-tolerance ideas and architecture for distributed systems [8]. His approach was to mask multiple independent concurrent low-level component failures through reconfiguration of redundant components. This low-level masking prevented the failures from being apparent to higher levels of abstraction.

Papadopalos presents an event based system for online monitoring [49]. This work is primarily concerned with state changes in complex systems that change fault propagation paths. He asserts that representing these state-fault dependencies is crucial to developing accurate system models. He models these state dependencies using a dynamic model of hierarchical finite state machines where each machine is a component and its children are subcomponents. This model is automatically translated from graphical fault tree. Low-level failures can propagate up the hierarchy to cause

failure of components. At each level recovery measures can be enacted and conditions checked to verify success or failure. The systems he is primarily concerned with are continuous, so he introduces the concept of normal transient behavior. Inputs to the system sometimes fluctuate beyond "normal" limits, for instance during startup. In this model, conditions must remain true over some time period to affect the monitoring model. The system he built consists of three main components: an event monitor, a diagnostic engine, and an event processor. The event monitor detects symptoms of failure. It uses a trinary logic of 1, 0, and unknown to account for circumstances of incomplete information. The diagnostic engine then traverses a fault tree where symptoms are top level events to return a set of possible root causes of the symptoms. The event processor examines the impact of events on the state machines in order to keep track of current state and take corrective action. The target systems for this research are mostly small physical systems rather than large distributed applications. It assumes a small number of components that can be precisely monitored in real-time. This approach does not appear to be scalable to large distributed information systems.

Chameleon is a "software infrastructure for adaptive fault tolerance" [28]. The system utilizes ARMORs to control the software elements of a system in order to provide multiple software fault tolerance strategies. The intended target appears to be fault-tolerant program execution on small-scale distributed systems experiencing random failures, although no specific mention is made. There is little mention of the scalability and no mention of widespread failures or security attacks.

Other fault-tolerant systems are WAFT [1], Coyote [5], and ISIS/Horus [6]. While these systems are designed for distributed systems, they are not designed explicitly to be scalable nor to tolerate the complex non-local faults apparent in critical infrastructure systems.

RAPTOR [14] is the system from which this work directly descends. The goals of RAPTOR are the same as those for TEDL and SPARTAN. RAPTOR uses three specifications for their system. One describes the fault-tolerant system, one the detectable errors, and one the recovery responses. An object-oriented database is used to store descriptive information about the system. The formal specification notation Z is used to define the error detection and error recovery specifications. The system was implemented as a distributed simulation with positive results. However, the centralized nature of detection significantly decreased the scalability. The Z notation made description difficult

and was the primary impetus for developing a specialized notation.

8.2 Languages

8.2.1 Policy Languages

The Policy Description Language (PDL) [38] is an event-condition-action (ECA) language from Bell Labs designed to describe policies in network switches. There is little inherent modularity in the language, which makes complex rules difficult to write and understand. Their basic construct is the "event causes action if condition" statement. Groups of these statements are composed to define policies. The main focus of the work on PDL is the formal semantics of the language, so the usability of the semantic model and language for complex policies is not a primary concern.

Ponder [40] is a general, declarative, object-oriented policy language. The main focus is role-based access control, but it has an ECA construct known as an "obligation." The main target of Ponder is small scale network management. The language is tailored to small sequences of events, so describing large complex sequences of events is difficult. There is little structure for modularity or abstraction, as all elements of an obligation must be contained within a single declaration.

8.2.2 IDS Languages

Much work has been done in the area of specification languages that are specifically for intrusion detection systems. These languages mostly fall into the categories of event, response, reporting, correlation, exploit, and detection [13]. Event languages describe how data should be formatted and are generally used for application logging. Examples are Sun Microsystems' SunSHIELD Basic Security Module (BSM) audit records [45], syslog messages [3], shorewall firewall log messages [61], xinetd messages, tcpdump packets [42], and the Normalized Audit Data Format (NADF) [46]. Response languages describe action that should be taken in response to attacks. There are currently no examples of these languages. Most systems that allow response require that this action be coded directly in a programming language.

Reporting languages describe a common format in which to encode IDS alerts. Examples are the Common Intrusion Specification language (CISL) [18], the Intrusion Detection Message Ex-

change Format (IDMEF) [11], and the Snort alert format [62, 56]. Exploit languages attempt to encode the steps an attacker must take to exploit a vulnerability. These include the Custom Attack Simulation Language (CASL) [47] and Nessus Attack Specification Language (NASL) [12]. Exploit languages are generally focused on specific, single-point types of attacks, i.e., buffer-overflow attacks, and are not general purpose attack description tools.

The main body of research related to TEDL is in the area of correlation and detection languages. Detection languages are designed to detect certain events, usually from network streams, that identify an attack. These include N-code used in Network Flight Recorder [55], P-BEST used in SRI's EMERALD [36], RUSSEL used in ASAX [20], SNP-L [68], GASSATA [43], the language used in IDIOT [9, 34, 33], the language used in Bro [51], the language used in Snort [56, 62], parallel environment grammars [32], JIGSAW [67], REE [59, 60], and ASL [58].

Correlation languages describe the relations among separate events, possibly detected by a detection language, and attempt to reason abstract meaningful events from them. Examples of correlation languages are Honeywell's ARGUS [2], SRI's eBayes [69], STATL [13], SRI's P-BEST [36], MuSigs [35], Roger and Goubault-Larreq's linear time temporal logic [57], Uppuluri and Sekar's REE [59, 60], Sutekh from Pouzol and Ducasse [53, 54], Gerard's LaDAA language for generating ASAX rules [17], LAMBDA from Cuppens and Ortalo [10], and ADeLe from Michel and Me [44].

STATL [13] is a detection language developed for attack specification in the STAT family of intrusion detection systems, including USTAT, NetSTAT, and WinSTAT. STATL models attacks as a set of states, beginning with an initial secure state and moving through predicate transitions until a final compromised state is reached. The primary focus is network intrusion detection so the language is targeted to a packet or connection level event detection. The language itself has a very similar look to C. Detection units are called scenarios and consist of a set of states and transitions that occur between the states. Transitions in the language are either consuming, nonconsuming, or unwinding. Consuming transitions act like a deterministic finite automaton, such that the machine transitions to only one state upon input. Non-consuming transitions allow a non-deterministic approach such that after a transition event, further events can still cause transitions from the previous state. Unwinding transitions allow for "rollback" of state transitions whose predicates are no longer

be satisfied or after certain amount of time transpires before a new event causes another transition. The timing element of the system is instantiated using the concept of countdown timers that than can be started upon entering a state. Unwinding transitions can occur if the the timer goes off before another transition occurs. One of the limitations in the language is that all elements of a scenario must be contained within the scenario. This does not allow for reuse of any of the specified elements in other scenarios.

Sutekh from Pouzol and Ducasse [53, 54] is a declarative language for intrusion detection. The construct their system from events consisting of attribute/value pairs, sets of constraints called filters on these events, totally ordered sequences of these events called trails, and signatures consisting of combinations of filters. They present an approach of how this declarative language would be translated into an algorithmic description for execution.

ADeLe is IDS exploit, detection, correlation, and response language from Michel and Me [44]. It was developed in parallel with the LAMBDA language [10] in the Mirador project. LAMBDA's approach is more declarative, while ADeLe's approach is more procedural. The syntax for ADeLe is a hybrid of XML and C. ADeLe is designed mainly for detection at a local level using network traffic.

SEL is an event correlation language used in the SHAMAN network management scripting Framework [73]. ODE [16] and CEDAR [22] are languages designed for specifying active database triggers. EBBM [4] is designed to create meaningful abstractions from system events for debugging purposes. These languages are designed with precise expressivity as their main criterion and are generally not scalable.

8.3 Event Correlation

A significant amount of work in the area of event correlation applies to this research. The primary focus of most event correlation systems is on simple management of relatively small networks suffering from random failures or on possibly large telecommunications networks suffering from a small number of different types of failures. In general, their syntax is not modular or clearly readable. Many of the research systems available are described in more detail in Henderson [23].

The Generalized Event Monitor (GEM) was developed [41] for network management event correlation. It has a powerful expression language with syntax similar to C and introduces effective algorithms for enacting this language on events. RAPIDE [39] uses partially ordered sets to represent detected event patterns and then generates higher-level events. One advantage of RAPIDE is that new rules can be dynamically modified while the program is running.

The Software Monitoring System (SoMoS) was developed by the Computing Services Support Solutions [63]. SoMoS is programmed in the Formal Language for Expressing Assumptions (FLEA) which is based on Common Lisp. Unfortunately, SoMoS was not originally intended to be an event correlation system and, therefore, does not have several of the necessary characteristics for high-volume event correlation.

JECTOR [37] is an event correlator based on the JEM event detector. Their primary focus is on condition detection in active databases. They introduce the concept of composite events that are determined from basic events in the network. The system relies on a centralized event database. The system was tested in an anecdotal way with real event data from HP OpenView. This data was mostly from misconfigured machines and did not consider intrusion detection events or complex failures.

Henderson's framework is useful in designing an event correlation system [23]. For the description language, he uses a combination of XML and programming language code to make many scenarios easy to specify and difficult scenarios possible to specify. The system uses a centralized structure for correlation and uses a publish/subscribe network for communication with basic event sensors.

The system most applicable to SPARTAN is the Stanford University Complex Event Processor (CEP) [52]. Their primary focus is "cyber battlefield awareness" so that humans can get insight into events occurring in an application network. They introduce the concept of event abstraction hierarchies for creating different levels of semantic human interface. However, this hierarchy is fixed at four levels with each level performing a specific duty. They support causal relationships, dynamic configuration with a pattern language, and decentralized processing networks, although no mention is made of logical connection mechanism or organization. They present a pattern algebra

that is useful in rigorously specifying matching semantics.

Gruschke proposes the use of dependency graphs for specifying event correlation [19]. Dependency graphs are built to show the functional dependencies between components, such that if A depends on B, then a failure in B will elicit symptoms from A. However, their approach is mostly as an interface to existing event correlation systems and is generally not suitable to high volume event systems with large numbers of complex interactions.

Commercial systems doing event correlation include IMPACT, NetFACT, EXCpert, InCharge, Hewlett-Packard OpenView Event Correlation Service, NerveCenter, and the experimental GRACE [27] system from GTE. In general, they are specifically focused on small local area networks experiencing misconfiguration or random rare failures and therefore have little use for our purpose. For example, HP OpenView's Event Correlation Description Language is very expressive, but is tailored specifically to SNMP and CMIP events.

Chapter 9

Conclusion

Don't worry about what anybody else is going to do. The best way to predict the future is to invent it. – Alan Kay

9.1 Conclusion

The importance of application networks used to run critical infrastructure system necessitates that they be survivable. Non-local fault tolerance is one mechanism for survivability and can be implemented using a survivability control system. This thesis has presented the TEDL language for specifying policy in such a control system.

The syntactic and semantic elements of TEDL combine to form an effective language for the specification of survivability control system policy. The language has been shown to meet the goals of scalability, expressivity, ease of use, implementability, and analyzability. The event-driven architecture provides for scalable detection, and the intentional command mechanism provides for scalable response. The declarative, object-oriented classes allow for modular specification with clear abstraction and clean encapsulation. The finite state machine model provides a useful semantic for detection. Composing these machines into a hierarchy allows for the decentralization of the communication and computation required for scalable detection. By basing detection on dynamic sets of sensor events rather than single events, detections involving large numbers of different events become manageable.

Through experimentation and argument, the TEDL language has been shown to be useful for the specification of non-local fault detection and response in large-scale applications.

9.2 Future Work

We see the following as the future direction of this work:

- Formal specification of response commands: TEDL currently requires that response commands be coded in Java. A language mechanism for declaring the response action will be integrated into the language.

- Robust exception handling model: Since responses are not integrated into the language, we have not developed a robust exception handling model. Exception handling is done in an informal way using the harvesting results of the intentional commands. As a better understanding is obtained of the exceptions that can occur in the intentional command paradigm, appropriate language mechanisms will be developed.

- Graphical language: The finite state machine model of TEDL makes graphical representation natural. We plan on developing a graphical equivalent of the language, similar to Statecharts, that can be used to specify control systems graphically.

- Formal analysis: TEDL is a formal language and is, therefore, amenable to formal analysis. We plan to research how formal methods, such as model checking, can be used to analyze TEDL specifications for desired properties.

Appendix A

Language Grammar

This is the SableCC grammar for the TEDL language. The notation is nearly the same as Backus-Nahr Form, with the replacement of the := production notation with = and the addition of a braced alternate name (i.e., *transform_with_variable*) before each alternate production.

```
Package tedl;

Helpers

all = [0 .. 0xffff];
letter = [['a' .. 'z'] + ['A' .. 'Z']];
digit = ['0' .. '9'];
non_zero_digit = ['1'..'9'];
tab = 9;
cr = 13;
lf = 10;
eol = cr lf | cr | lf;        // This takes care of different platforms
sp  = ' ';

input_character = [all - [cr + lf]];

not_star =    [input_character - '*'] | eol;
not_star_not_slash = [input_character - ['*' + '/']] | eol;
a = 'a' ;
b = 'b' ;
c = 'c' ;
d = 'd' ;
e = 'e' ;
f = 'f' ;
g = 'g' ;
h = 'h' ;
i = 'i' ;
j = 'j' ;
```

```
k = ’k’ ;
l = ’l’ ;
m = ’m’ ;
n = ’n’ ;
o = ’o’ ;
p = ’p’ ;
q = ’q’ ;
r = ’r’ ;
s = ’s’ ;
t = ’t’ ;
u = ’u’ ;
v = ’v’ ;
w = ’w’ ;
x = ’x’ ;
y = ’y’ ;
z = ’z’ ;
escape_sequence = ’\b’ | ’\t’ | ’\n’ | ’\f’ | ’\r’ | ’\"’ | ’\’ ’’’ | ’\\’ ;
string_character = [input_character - [’"’ + ’\’]] | escape_sequence;

Tokens

number = digit+;
plus = ’+’;
minus = ’-’;
mult = ’*’;
div = ’/’;
mod = ’%’;
l_par = ’(’;
r_par = ’)’;
l_brace = ’{’;
r_brace = ’}’;
l_bracket = ’[’;
r_bracket = ’]’;
blank = (’ ’ | tab | eol)+;
delimiter = ’;’;
transition_op = ’->’;
comma = ’,’;
colon = ’:’;
card_op = ’#’;
assignment_op = ’:=’;
equality_op = ’=’;
inequality_op = ’!=’;
greater_than_op = ’>’;
greater_than_equal_op = ’>=’;
less_than_op = ’<’;
less_than_equal_op = ’<=’;
on = o n ;
run = r u n;
conjunction_op = a n d ;
disjunction_op = o r ;
negation_op = n o t ;
```

```
include_dec = i n c l u d e ;
predicate_dec = p r e d i c a t e ;
transform_dec = t r a n s f o r m ;
machine_dec = m a c h i n e ;
set_dec = s e t ;
response_dec =  r e s p o n s e ;
deployment_dec = d e p l o y m e n t;
property_dec = p r o p e r t y;
has = h a s;
extends = e x t e n d s ;
super = s u p e r ;
msec = m s e c ;
sec = s e c ;
min = m i n ;
hour = h o u r ;
day = d a y ;
week = w e e k ;
month = m o n t h ;
year = y e a r ;
unique = u n i q u e ;
in = i n ;
do = d o ;
offset = o f f s e t ;
var = v a r ;
set = s e t ;
action = a c t i o n;
siena = s i e n a;
qual = q u a l;
continuous = c o n t i n u o u s;
state = s t a t e;
event = e v e n t;
output = o u t p u t ;
end_of_line_comment = '//' input_character* eol?;
identifier = (letter | digit | '_' | '.' )+;
string_literal = '"' string_character* '"';

Ignored Tokens

blank, end_of_line_comment;

Productions
spec = elements*;
elements = {include} include_description | {event} event_description |
{predicate} predicate_description | {transform} transform_description |
{machine} machine_description | {set} set_description | {response}
response_description | {variable} variable_declaration | {deployment}
deployment_description | {siena} siena_description |
{state} state_description;
siena_description = siena string_literal delimiter ;
include_description = include_dec identifier delimiter ;
math_op = {plus} plus | {minus} minus | {mult} mult | {div} div| {mod} mod;
```

```
rel_op = {eq} equality_op | {neq} inequality_op | {gt} greater_than_op |
{gte} greater_than_equal_op | {lt} less_than_op | {lte} less_than_equal_op;
bool_op = {and} conjunction_op | {or} disjunction_op | {neg} negation_op;
inheritance_phrase = extends identifier;

// Event declaration
event_description = event identifier inheritance_phrase? l_brace event_attributes r_brace;
event_attributes = event_attribute* ;
event_attribute = identifier delimiter ;

// State  declaration
state_description = state identifier l_brace state_attributes r_brace ;
state_attributes = state_attribute* ;
state_attribute = do identifier delimiter ;

// Variable declaration
variable_declaration = {string_var} var identifier assignment_op string_literal
  delimiter | {int_var} var identifier assignment_op number delimiter;

// Predicate declaration
predicate_description = predicate_dec identifier predicate_continuous? l_brace
  predicate_body delimiter r_brace;
predicate_continuous = continuous ;
predicate_body = {pbs} predicate_body_simple | {pbc} predicate_body_compound;
predicate_body_compound = negation_op? identifier predicate_composer ;
predicate_composer = bool_op negation_op? identifier;
predicate_body_simple = identifier predicate_set_comparison;
predicate_set_comparison = rel_op predicate_comparator;
predicate_comparator = {pred_num} number | {pred_ident} identifier |
  {pred_trans} predicate_transform identifier;
predicate_transform = l_par identifier r_par ;

// Clause declaration
set_description = set_dec identifier inheritance_phrase? l_brace set_body r_brace;
set_body = {cb_to_cbs} set_body_simple delimiter  | {cb_to_cbd} set_body_derived delimiter;
set_body_simple = set_unique? identifier set_time_period set_time_offset set_qualifiers?;
set_unique = unique;
set_time_period = in number set_time_division;
set_time_offset = offset number set_time_division;
set_time_division = {td_to_msec} msec | {td_to_sec} sec | {td_to_min} min |
  {td_to_hour} hour | {td_to_day} day ;

set_qualifiers = set_qualifier+;
set_qualifier = qual identifier set_qualifier_value;
set_qualifier_value = rel_op identifier;

set_body_derived = super set_time_period? set_time_offset?;

// Machine declaration
machine_description = machine_dec identifier inheritance_phrase? l_brace machine_body r_brace;
machine_body = machine_rule+;
```

```
machine_rule = machine_condition machine_result delimiter;
machine_condition = identifier plus machine_transition;
machine_transition = identifier transition_op;
machine_result = identifier ;

// Response declaration
response_description = response_dec identifier inheritance_phrase? l_brace response_body r_brace;
response_body = response_command+;
response_command = {response_event} response_event_set | {response_action}
action identifier delimiter | {response_pair} response_pair;
response_event_set = output identifier delimiter;
response_pair = identifier assignment_op string_literal  delimiter;

// Transform declaration
transform_description = transform_dec identifier l_brace transform_body r_brace;
transform_body = {transform_with_num} math_op number |
{transform_with_variable} math_op identifier;

deployment_description = deployment_dec identifier l_brace deployment_body r_brace;
deployment_body = deployment_entry+ ;
deployment_entry = identifier deployment_location deployment_machine deployment_parent delimiter;
deployment_location = colon identifier;
deployment_machine = run identifier;
deployment_parent = transition_op identifier;
```

Appendix B

TEDL Source Files

This appendix lists the specifications used for the experiments presented in Chapter 5.

B.1 File siena.tedl

This file is used by all of the specifications. It allows the address of the Siena router to be declared in a single place for all specifications.

```
siena "tcp:sirius.cs.virginia.edu:23456" ;
```

B.2 Money Center Bank Failure Scenario

This is the specification for the scenario presented in Section 6.2.1. A state diagram of this scenario can be seen in Figure 6.2.

```
include siena.tedl ;

var InitialAlarmThreshold := 10 ;
var AlarmMinimumSafeThreshold := 0 ;
var Success := 1 ;

event MCBNoResponseAlarmEvent { }

set PreviousMinuteUniqueAlarms {
  unique MCBNoResponseAlarmEvent in 1 min offset 0 min ;
}

set SuccessfulRepairEvents {
  unique MCBRepairSuccessfulEvent in 1 min offset 0 min ;
}
```

```
set Previous10MinAlarms {
  MCBNoResponseAlarmEvent in 10 min offset 0 min ;
}

predicate MCBFailureTrigger {
  PreviousMinuteUniqueAlarms >= InitialAlarmThreshold ;
}

predicate MCBFailureRepaired {
  SuccessfulRepairEvents >= Success ;
}

response ReconfigureChildBranchBanks {
  action MCBFailureResponse ;
  sienaURI := "tcp:sirius:23456" ;
  attributeModelURI :=
    "http://vasser.cs.virginia.edu/willow/ANDREA/Resources/bank/bankAttributeModel.xml" ;
  backupMCB := "Richmond_MCB_001" ;
}

state Normal {
}

state MCBFailure {
  do ReconfigureChildBranchBanks;
}

machine MCBFailureDetector {
  Normal + MCBFailureTrigger -> MCBFailure;
  MCBFailure + MCBFailureRepaired -> Normal ;
}

deployment MCBFailureDetectorDeployment {
  MainMCBFailureDetector : centurion128 run MCBFailureDetector -> none ;
}
```

B.3 Coordinated Attack Scenario

This is the specification for the Coordinated Attack Scenario presented in Section 6.2.2. A state diagram of this scenario can be seen in Figure 6.3.

```
include siena.tedl ;

var UniqueInitialThreshold := 200 ;
var UniqueAdvancedInitialThreshold := 400 ;
var ReconfiguredNumber := 1 ;
var AttackMinimum := 10 ;

event SecurityEvent { }
```

```
event DoSSecurityEvent extends SecurityEvent { }
event PortscanSecurityEvent extends SecurityEvent { }
event AttackResponseSuccessfulEvent { }
event AttackResponseSuccessfulEvent extends AttackResponseSuccessfulEvent { }
event AdvancedAttackResponseSuccessfulEvent extends AttackResponseSuccessfulEvent { }

set PreviousByteUniqueSecurityEventSet {
  unique SecurityEvent in 8 min offset 0 min ;
}

set PreviousByteAttackResponseSuccessEventSet {
  AttackResponseSuccessfulEvent in 8 min offset 0 min ;
}

set PreviousByteAdvancedAttackResponseSuccessEventSet {
  AdvancedAttackResponseSuccessfulEvent in 8 min offset 0 min ;
}

predicate AttackEndedPredicate {
  PreviousByteUniqueSecurityEventSet <= AttackMinimum ;
}

predicate AttackPredicate {
  PreviousByteUniqueSecurityEventSet >= UniqueInitialThreshold ;
}

predicate AdvancedAttackPredicate {
  PreviousByteUniqueSecurityEventSet >= UniqueAdvancedInitialThreshold ;
}

predicate ReconfiguredPredicate {
  PreviousByteAttackResponseSuccessEventSet >= ReconfiguredNumber ;
}

predicate AdvancedReconfiguredPredicate {
  PreviousByteAdvancedAttackResponseSuccessEventSet >= ReconfiguredNumber ;
}

response AttackResponse {
  action CoordinatedAttackResponse;
  sienaURI := "tcp:sirius.cs.virginia.edu:23456" ;
  attributeModelURI :=
    "http://vasser.cs.virginia.edu/willow/ANDREA/Resources/bank/bankAttributeModel.xml" ;
}

response AdvancedAttackResponse {
  action AdvancedCoordinatedAttackResponse;
  sienaURI := "tcp:sirius.cs.virginia.edu:23456" ;
  attributeModelURI :=
    "http://vasser.cs.virginia.edu/willow/ANDREA/Resources/bank/bankAttributeModel.xml" ;
}
```

```
state NormalState {
}

state UnderAttackState {
  do AttackResponse;
}

state UnderAdvancedAttackState {
  do AdvancedAttackResponse;
}

state ReconfiguredState {
}

machine CoordinatedAttackDetector {
  NormalState + AttackPredicate -> UnderAttackState ;
  UnderAttackState + AdvancedAttackPredicate -> UnderAdvancedAttackState ;
  UnderAttackState + ReconfiguredPredicate -> ReconfiguredState ;
  UnderAdvancedAttackState + AdvancedReconfiguredPredicate -> ReconfiguredState ;
  ReconfiguredState + AttackEndedPredicate -> NormalState ;
  ReconfiguredState + AdvancedAttackPredicate -> UnderAdvancedAttackState ;
}
```

B.4 Regional-level Power Failure Scenario

This is the specification for the Regional-level Power Failure Scenario presented in Section 6.2.3. The state diagrams of the machines used for this scenario can be seen in Figures 6.4 and 6.5.

B.4.1 RegionalFailureDetector

```
include siena.tedl ;

var InitialAlertThreshold := 250 ;

event FailureAlertEvent { }
event PowerFailureAlertEvent extends FailureAlertEvent{ }
event RegionalPowerFailureAlertEvent extends FailureAlertEvent{ }

set PreviousMinuteUniqueAlerts {
  unique PowerFailureAlertEvent in 5 min offset 0 min ;
}

predicate RegionalFailureTrigger {
  PreviousMinuteUniqueAlerts >= InitialAlertThreshold ;
}

response RegionalFailureAlert {
```

```
  output RegionalPowerFailureAlertEvent ;
}

state Normal { }

state Failure {
  do RegionalFailureAlert ;
}

machine RegionalFailureDetector {
  Normal + RegionalFailureTrigger -> Failure ;
}
```

B.4.2 NationalFailureDetector

```
include siena.tedl ;

var InitialAlertThreshold := 2 ;

event FailureAlertEvent { }
event PowerFailureAlertEvent extends FailureAlertEvent{ }
event RegionalPowerFailureAlertEvent extends FailureAlertEvent{ }
event NationalPowerFailureAlertEvent extends FailureAlertEvent{ }

set PreviousMinuteAlerts {
  RegionalPowerFailureAlertEvent in 5 min offset 0 min ;
}

predicate NationalFailureTrigger {
  PreviousMinuteAlerts >= InitialAlertThreshold ;
}

response NationalFailureAlert {
  output NationalPowerFailureAlertEvent ;
}

state Normal { }

state Failure {
  do NationalFailureAlert ;
}

machine NationalFailureDetector {
  Normal + NationalFailureTrigger -> Failure ;
}
```

Bibliography

[1] L. Alvisi and K. Marzullo. WAFT: Support for Fault-Tolerance in Wide-Area Object Oriented Systems. In *Proceedings of the 2nd Information Survivability Workshop*, October 1998.

[2] Anonymous. Argus: An Architecture for Cooperating Intrusion Detection and Mitigation Applications, 2002. http://www.htc.honeywell.com/projects/argus/.

[3] Anonymous. syslog(3) man page, 2002.

[4] P. Bates. Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior. *ACM Transactions on Computer Systems*, 13(1), February 1995.

[5] N. Bhatti, M. Hiltunen, R. Schlichting, and W. Chiu. Coyote: A System for Constructing Fine-Grain Configurable Communication Services. *ACM Transactions on Computer Systems*, 16(4), November 1998.

[6] K. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12), December 1993.

[7] Antonio Carzaniga, David S. Rosenblum, and Alexander L Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.

[8] F. Cristian, B. Dancey, and J. Dehn. Fault-Tolerance in Air Traffic Control Systems. *ACM Transactions on Computer Systems*, 14(3):265–286, August 1996.

[9] M. Crosbie, B. Dole, T. Ellis, I. Krsul, and E. Spafford. IDIOT— Users Guide. The COAST Project. Technical Report TR-96-050, Dept. of Computer Science, Purdue University, September 1996.

[10] F. Cuppens and R. Ortalo. LAMBDA: A Language to Model a Database for Detection of Attacks. In *Proceedings of the Third International Workshop on Recent Advances in Intrusion Detection (RAID'2000)*, 2000.

[11] D. Curry. Intrusion Detection Message Exchange Format: Extensible Markup Language (XML) Document Type Definition, 2000. http://www.ietf.org/internet-drafts/draft-ietf-idwg-idmef-xml-10.txt.

[12] R. Deraison. The NASL2 reference manual , 2002. http://www.nessus.org.

[13] S. Eckmann, G. Vigna, and R. Kemmerer. STATL: An Attack Language for State-based Intrusion Detection. *Journal of Computer Security*, 10(1/2):71–104, 2002.

[14] M. C. Elder. *Fault Tolerance in Critical Information Systems*. PhD thesis, University of Virginia Department of Computer Science, May 2001.

[15] É. Gagnon. SableCC, An Object-Oriented Compiler Framework. Master's thesis, McGill University, 1998. http://www.sablecc.org/thesis.pdf.

[16] N. Gehani, H.V. Jagadish, and O. Shumeli. Composite Event Specification in Active Databases: Model and Implementation. In *Proc. 18th Interna-tional Conference on Very Large Data Bases*, 1992.

[17] F. Gérard. *Définition et Implémentation d'un Langage d'Analyse d'Audit Trails*. PhD thesis, Facultés Universitaires Notre-Dave de la Paix Namur (Belgium), 1998.

[18] Common Intrusion Detection Framework Working Group. A CISL Tutorial, 2000. http://www.gidos.org/tutorial.html.

[19] B. Gruschke. A New Approach for Event Correlation based on Dependency Graphs. In *Proceedings of the 5th Workshop of the OpenView University Association: OVUA'98*, April 1998.

[20] J. Habra, B. Le Charlier, A. Mounji, and I. Mathieu. ASAX: Software architecture and rule-based language for universal audit trail analysis. In Yves Deswarte et al., editor, *Computer Security - Proceedings of ESORICS 92*, volume 648 of LNCS, pages 435–450, Toulouse, France, November 1992. Springer-Verlag.

[21] R. M. Hall, D.M. Heimbigner, and A.L. Wolf. A Cooperative Approach to Support Software Deployment Using the Software Dock. In *Proceedings of the 1999 International Conference on Software Engineering*, 1999.

[22] M. Hasan. *The Management of Data, Events, and Information Presentation for Network Management*. PhD thesis, University of Waterloo, 1996.

[23] M. Henderson. A Framework for Event Correlation. Master's thesis, University of Queensland, 1999. http://elvin.dstc.edu.au/projects/correlation.

[24] J. C. Hill. Management and Adaptation of Large Distributed Applications. Ph.D. Proposal, University of Virginia, 2002.

[25] J. C. Hill, J. C. Knight, A. M. Crickenberger, and R. Honhart. Publish and Subscribe with Reply. Unpublished, University of Virginia, 2002.

[26] J.C. Hill and J.C. Knight. Selective notification: Combining forms of decoupled addressing for internet-scale command and alert dissemination, 2003. Submitted to SRDS2003 - 22nd Symposium on Reliable Distributed Systems, Florence, Italy (October 2003).

[27] G. Jakobson, M. Weissman, L. Brenner, C. Lafond, and C. Matheus. GRACE: Building Next Generation Event Correlation Services. In *Proceedings of the 2000 IEEE/IFIP Network Operations and Management Symposium*, 2000.

[28] Z. T. Kalbarczyk, R. K. Iyer, S. Bagchi, and K. Whisnant. Chameleon: A Software Infrastructure for Adaptive Fault Tolerance. *IEEE Transactions on Parallel and Distributed Systems*, 10(6), 1999.

[29] J. C. Knight and M. C. Elder. Fault Tolerant Distributed Information Systems. In *International Symposium on Software Reliability Engineering*, November 2001.

[30] J. C. Knight, D. Heimbigner, A. Wolf, A. Carzaniga, J. Hill, P. Devanbu, and M. Gertz. The Willow Architecture: Comprehensive Survivability for Large-Scale Distributed Applications. In *Proceedings of DSN-2002 The International Conference on Dependable Systems and Networks*, 2002.

[31] J. C. Knight, E. A. Strunk, and K. J. Sullivan. Towards a Rigorous Definition of Information System Survivability. In *DARPA Information Survivability Conference and Exposition (DISCEX 2003)*, April 2003.

[32] Calvin Ko, M. Ruschitzka, and K. Levitt. Execution Monitoring of Security-critical Programs in Distributed Systems: A Specification-based Approach. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, volume ix, pages 175–187, Oakland, CA, May 1997. IEEE Computer Society Press.

[33] S. Kumar and E. H. Spafford. A Pattern Matching Model for Misuse Intrusion Detection. In *Proceedings of the 17th National Computer Security Conference*, pages 11–21, Baltimore MD, USA, 1994.

[34] S. Kumar and E. H. Spafford. An Application of Pattern Matching in Intrusion Detection. Technical Report CSD-TR-94-013, The COAST Project, Dept. of Computer Sciences, Purdue University, West Lafayette, IN, USA, June 1994.

[35] J.-L. Lin, X. S. Wang, and S. Jajodia. Abstraction-Based Misuse Detection: High-Level Specifications and Adaptable Strategies. In *PCSFW: Proceedings of The 11th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1998.

[36] U. Lindqvist and P. Porras. Detecting Computer and Network Misuse Through the Production-based Expert System Toolset (P-BEST). In *1999 IEEE Symposium on Security and Privacy*, pages 146–161, May 1999.

[37] G. Liu, A. K. Mok, and E. Yang. Composite Events for Network Event Correlation. In *Proceedings of IM'99*, May 1999.

[38] J. Lobo, R. Bhatia, and S. Naqvi. A Policy Description Language. In *Proc. of AAAI 1999*, July 1999.

[39] D. C. Luckham and B. Frasca. Complex Event Processing in Distributed Systems. Technical Report CSL-TR-98-754, Stanford University, March 1998.

[40] L. Lymberopoulos, E. Lupu, and M. Sloman. An Adaptive Policy Based Management Framework for Differentiated Services Networks. In *Proc. 3rd IEEE Workshop on Policies for Distributed Systems and Networks (Policy 2002)*, pages 147–158, June 2002.

[41] M. Mansouri-Samani and M. Sloman. GEM: A Generalised Event Monitoring Language for Distributed Systems. *IEEE/IOP/BCS Distributed Systems Engineering Journal*, 4(2), June 1997.

[42] S. McCanne, C. Leres, and V. Jacobson. Tcpdump 3.4 Documentation, 1998. http://www.tcpdump.org/.

[43] L. Me. Gassata, a Genetic Algorithm as an Alternative Tool for Security Audit Trails Analysis. In *Proceedings of the First International Workshop on Recent Advances in Intrusion Detection (RAID '98)*, 1998.

[44] Cédric Michel and Ludovic Mé. ADeLe: an Attack Description Language for Knowledge-based Intrusion Detection. In *Proceedings of the 16th International Conference on Information Security (IFIP/SEC 2001)*, pages 353–365, June 2001.

[45] Sun Microsystems. *SunSHIELD Basic Security Module Guide*. SunSoft, 2000.

[46] A. Mounji. *Languages and Tools for Rule-Based Distributed Intrusion Detection*. PhD thesis, Computer Science Institute, University of Namur, Belgium, Sept 1997.

[47] Secure Networks. Custom Attack Scripting Language (CASL), 1998. http://www.sockpuppet.org/tqbf/casl.html.

[48] P. Ning, S. Jajodia, and X. S. Wang. Abstraction-based intrusion detection in distributed environments. *Information and System Security*, 4(4):407–452, 2001.

[49] Y. Papadopoulos. Model-based On-line Monitoring Using a State Sensitive Fault Propagation Model. In *SAFECOMP 02, 21st Int. Conf. on Computer Safety, Reliability and Security*, 2002.

[50] D.L. Parnas. Building Reliable Software in BLOWHARD. *ACM Software Engineering Notes*, 2(3):5–6, April 1977.

[51] V. Paxon. Bro: A System for Detecting Network Intruders in Real-time. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, USA, January 1988.

[52] L. Perrochon, E. Jang, S. Kasriel, and D. C. Luckham. Enlisting Event Patterns for Cyber Battlefield Awareness. In *Proceedings of DISCEX 00*, 2000.

[53] J.-P. Pouzol and M. Ducassé. From Declarative Signatures to Misuse IDS. In *Proceedings of the 4th International Symposium, RAID 2001*, 2001.

[54] J.-P. Pouzol and M. Ducassé. Formal Specification of Intrusion Signatures and Detection Rules. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*, 2002.

[55] Marcus Ranum. NFR Homepage, 2001. http://www.nfr.com.

[56] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of USENIX LISA 99 conference*, 1999.

[57] M. Roger and J. Goubault-Larrecq. Log Auditing through Model-Checking. In *Proceeding of the 14th Computer Security Foundations Workshop*. IEEE Computer Society Press, 2001.

[58] R. Sekar, Y. Cai, and M. Segal. A Specification-Based Approach for Building Survivable Systems. In *Proc. 21st NIST-NCSC National Information Systems Security Conference*, pages 338–347, 1998.

[59] R. Sekar, Y. Guang, S. Verma, and T. Shanbhag. A High-performance Network Intrusion Detection System. In *Proceedings of the 6th ACM conference on Computer and communications security*, pages 8–17, 1999.

[60] R. Sekar and P. Uppuluri. Synthesizing Fast Intrusion Prevention/Detection Systems from High-Level Specifications. In *Proceedings 8th Usenix Security Symposium*, pages 63–78, Washington DC, 1999.

[61] Shoreline. Shorewall, 2003. http://www.shorewall.net/.

[62] Snort. Snort homepage, 2001. http://www.snort.org.

[63] Computing Services Support Solutions. FLEA Overview, 2002. http://www.compsvcs.com/flea_overview.html.

[64] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 2002.

[65] J. Michael Spivey. Specifying a real-time kernel. *IEEE Software*, 7(5), September/October 1990.

[66] Kevin Sullivan, John C. Knight, Xing Du, and S. Geist. Information Survivability Control Systems. In *Proceedings of the Twenty-first International Conference on Software Engineering*. IEEE Computer Society Press, May 1999.

[67] S. J. Templeton and K. Levitt. A Requires/Provides Model for Computer Attacks. In *Proceedings of the New Security Paradigms Workshop 2000*, Sept 2000.

[68] E. Turner and R. Zachary. Securenet Pro Software's SNP-L Scripting System, 2000. http://www.intrusion.com.

[69] Alfonso Valdes and Keith Skinner. An Approach to Sensor Correlation. In *Third International Workshop on Recent Advances in Intrusion Detection (RAID'2000)*, 2000.

[70] Chenxi Wang. *A Security Architecture for Survivability Mechanisms*. PhD thesis, University of Virginia, October 2000.

[71] F. Wang, F. Gong, C. Sargor, K. Goseva-Popstojanova, K. Trivedi, and F. Jou. SITAR: A Scalable Intrusion Tolerance Architecture for Distributed Server. In *Proceedings of the IEEE 2nd SMC Information Assurance Workshop*, 2001.

[72] M. M. Williamson. Throttling Viruses: Restricting propagation to defeat malicious mobile code. In *ACSAC 2002 Las Vegas*, 2002.

[73] D. Zhu and A. Sethi. SEL, A New Event Pattern Specification Language for Network Management Event Correlation. Technical Report 2002-01, University of Delaware, 2001.

[74] A. Avižienis, J.-C. Laprie, and B. Randell. Fundamental Concepts of Dependability. In *Information Survivability Workshop 2000*, 2000.