Model Checking Implicit-Invocation Systems: An Approach to the Automatic Analysis of Architectural Styles

by

Jeremy Scott Bradbury

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
May 2002

Copyright © Jeremy Scott Bradbury, 2002

Abstract

In general, model checking and other finite-state analysis techniques have been very successful when used with hardware systems and less successful with software systems. It is especially difficult to analyze software systems developed with the implicit-invocation architectural style because the loose coupling of their components increases the size of the finite state model. It is the goal of this research to gain insight into the larger problem of how to make model checking a better analysis and verification tool for software systems. Specifically, we will extend an existing approach to model checking implicit-invocation systems. We will then proceed to evaluate our technique on several non-trivial examples.

Acknowledgments

I would like to thank my supervisor, Juergen Dingel, for his guidance and assistance in helping me write this thesis. I would also like to thank my thesis examining committee for their helpful suggestions. My committee consisted of:

- Stephen Brown (*Chairperson*)
- James Cordy (*Internal*)
- Juergen Dingel (*Supervisor*)
- Janice Glasgow (*Department Head*)
- Terry Shepard (*External*)

I would like to thank the Natural Sciences and Engineering Research Council of Canada (NSERC) for their generous financial support.

I would like to thank my parents, Goldie and Gerald, my sister, Pam, and my grandmothers, Jessie and Gladys, for their love and support.

Finally, I would like to dedicate this thesis in memory of Gordon Barrett and Warren Bradbury.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Problem 2

1.3 Thesis 3

1.4 Contributions 4

2 Background **6**

2.1 Implicit-Invocation Systems 6

2.1.1 Architectural Style 6

2.1.2 System Overview 9

2.1.3 System Structure 12

2.1.4 Examples 15

2.2 Analysis Techniques 19

2.2.1 Issues in Analysis of Implicit-Invocation Architectural Style . 20

2.2.2 Formal Methods 20

2.3 Model Checking Implicit-Invocation Systems 28

2.3.1 Development of a Generic Implicit-Invocation Model Checking Framework 29

2.3.2 Automated Tool 31

3 Event Representation Enhancements 35

3.1 Garlan & Khersonsky Technique 35

3.2 Problems with the Garlan & Khersonsky Technique 39

3.2.1 Modifying an SMV Model 40

3.2.2 Modifying XML Representation 41

3.3 Modified Approach: Adding Data to Events 42

3.3.1 Event Representation in XML 43

3.3.2 Formal Parameter Modifications 46

3.3.3 Event Dispatcher Modifications 46

3.3.4 Component Modifications 47

4 Delivery Policy Style Enhancements 50

4.1 Reasoning About Delivery Policies 51

4.1.1 What Information Affects Delivery? 51

4.1.2 How Does Information Affect Delivery? 52

4.1.3 Categorization of Delivery Policy Information 53

4.1.4 Delivery Policy Categorization 53

4.2 Garlan & Khersonsky Technique 55

4.2.1 Limitations 57

4.3 Modified Event Delivery Policy Representation 57

4.3.1 Theory Behind Modified Technique 57

4.3.2 Implementation of Delivery Policies in SMV 60

4.3.3 Encoding Policies in XML 62

4.3.4 Example using New Technique 64

5 Evaluation 66

5.1 Set-Counter Example 67

5.1.1 Background 67

5.1.2 Garlan & Khersonsky Implicit-Invocation Model of the Set-Counter Example 67

5.1.3 Our Implicit-Invocation Model of the Set-Counter Example . . 69

5.1.4 Analysis of Set-Counter Example 71

5.2 Active Badge Location System 73

5.2.1 Background 73

5.2.2 Implicit-Invocation Model of the Active Badge Location System 75

5.2.3 Analysis of Active Badge Location System 82

5.3 Unmanned Vehicle Control System 95

5.3.1 Background 95

5.3.2 Subscription Model of a Control System for Unmanned Vehicles 96

5.3.3 Implicit-Invocation Model of a Control System for Unmanned Vehicles 98

5.3.4 Analysis of Unmanned Vehicle Location System 100

6 Optimization Techniques for Model Checking 111

6.1 Advantages of Optimization 112

6.2 Model-Based Optimizations 113

6.2.1 Correctness and Failure Preserving Transformations 113

6.2.2 Data Abstraction 117

6.2.3 Reduction of Non-Determinism 118
6.3 Tool-Based Optimizations 119
6.3.1 Built-In Model Checker Parameters 119
6.3.2 Parallel Model Checking 120

7 Summary & Conclusions 122

7.1 Thesis Summary 122
7.2 Conclusions 124
7.3 Future Work 126
7.3.1 Optimization Techniques 127
7.3.2 Complete Automation of Model Generation 128
7.3.3 Alternative Intermediate Representation 129
7.3.4 Extension of Technique to Other Architectural Styles 131

Bibliography 132

A Glossary 138

B XML Document Type Definition (DTD) 143

C Set and Counter Example 151

C.1 Summary of Cadence SMV Results 151
C.1.1 gk_SC_immediate SMV Results 152
C.1.2 gk_SC_random SMV Results 153
C.1.3 mod_SC_1_immediate SMV Results 155
C.1.4 mod_SC_1_random SMV Results 156
C.1.5 mod_SC_2 SMV Results 157

D Active Badge Location System 159

D.1 Summary of Cadence SMV Results 159

D.1.1 poll_ABLS SMV Results 160

D.1.2 find_ABLS_immediate 162

D.1.3 find_ABLS_random 163

D.1.4 history_ABLS_queue_1 165

D.1.5 history_ABLS_queue_2 166

D.1.6 look_ABLS 168

D.1.7 with_ABLS 169

D.1.8 find_look_ABLS 171

E Unmanned Vehicle Control System Example 172

E.1 Summary of Cadence SMV Results 172

E.1.1 UVCS SMV Results: Rule Version Synchronization 173

E.1.2 UVCS SMV Results: Regional Vehicle Transfer 175

E.1.3 UVCS SMV Results: Collison Avoidance 176

F Variations on Event Delivery Policies 177

Vita 181

List of Tables

2.1 HP SoftBench Component Functionality 16

2.2 Segmentation of Jini Architecture 18

2.3 Common LTL Operators 27

6.1 State Count Results of `LookCorrectness` property 115

6.2 Analysis Results of `LookCorrectness` property 115

6.3 State Count Results of property LookCorrectness for look_ABLS system with variations in the number of locations 118

C.1 State Count Results of gk_SC_immediate 152

C.2 Analysis Results of gk_SC_immediate 153

C.3 State Count Results of gk_SC_random 153

C.4 Analysis Results of gk_SC_random 154

C.5 Counter-Example Results of gk_SC_random 154

C.6 State Count Results of mod_SC_1_immediate 155

C.7 Analysis Results of mod_SC_1_immediate 155

C.8 State Count Results of mod_SC_1_random 156

C.9 Analysis Results of mod_SC_1_random 156

C.10 Counter-Example Results of mod_SC_1_random 157
C.11 State Count Results of mod_SC_2 157
C.12 Analysis Results of mod_SC_2 158
C.13 Counter-Example Results of mod_SC_2 158

D.1 State Count Results of poll_ABLS 160
D.2 Analysis Results of poll_ABLS 161
D.3 Counter-Example Results of poll_ABLS 161
D.4 State Count Results of find_ABLS_immediate 162
D.5 Analysis Results of find_ABLS_immediate 162
D.6 Counter-Example Results of find_ABLS_immediate 163
D.7 State Count Results of find_ABLS_random 163
D.8 Analysis Results of find_ABLS_random 164
D.9 Counter-Example Results of find_ABLS_random 164
D.10 State Count Results of history_ABLS_queue_1 165
D.11 Analysis Results of history_ABLS_queue_1 165
D.12 Counter-Example Results of history_ABLS_queue_1 166
D.13 State Count Results of history_ABLS_queue_2 166
D.14 Analysis Results of history_ABLS_queue_2 167
D.15 Counter-Example Results of history_ABLS_queue_2 167
D.16 State Count Results of look_ABLS 168
D.17 Analysis Results of look_ABLS 168
D.18 Counter-Example Results of look_ABLS 169
D.19 State Count Results of with_ABLS 169
D.20 Analysis Results of with_ABLS 170

D.21 Counter-Example Results of with_ABLS 170
D.22 State Count Results of find_look_ABLS 171
D.23 Analysis Results of find_look_ABLS 171

E.1 State Count Results of Rule Version Synchronization Properties . . . 173
E.2 Analysis Results of Rule Version Synchronization Properties 174
E.3 Counter-Example Results of Rule Version Synchronization Properties 174
E.4 State Count Results of Regional Vehicle Transfer Properties 175
E.5 Analysis Results of Regional Vehicle Transfer Properties 175
E.6 State Count Results of Collison Avoidance Properties 176
E.7 Analysis Results of Collison Avoidance Properties 176

F.1 A Mutually Exclusive Priority Queue Based Delivery Policy 178
F.2 (Part A) An Environment Event Dependent Delivery Policy 179
F.3 (Part B) An Environment Event Dependent Delivery Policy 180

List of Figures

2.1 Comparison of Communication Channels 10

2.2 Implicit-Invocation System Structure 14

2.3 HP SoftBench with Implicit-Invocation Mechanism 17

2.4 Temporal Logic Model Checking Process 24

2.5 Semi-Automatic Analysis for Implicit-Invocation Systems 31

3.1 Announcement and Delivery for Component Announced Events . . . 37

3.2 Announcement and Delivery for Environment Announced Events . . . 38

3.3 Implicit-Invocation System Representations in Model Checking Process 40

4.1 Delivery Policy Categorization Levels for Environment Event Example 55

5.1 Implicit-Invocation Model of Set-Counter Example 68

5.2 Implicit-Invocation Model of Active Badge Location System 77

5.3 Subscription Model of Control System for Unmanned Vehicles 97

5.4 Implicit-Invocation Model of Control System for Unmanned Vehicles . 99

5.5 Regional Movement of Unmanned Vehicles 103

7.1 Fully Automated Model Checking Approach 130

Chapter 1

Introduction

1.1 Motivation

A study of model checking is motivated by the need for formal method techniques. As software systems become more integrated into our daily lives, our tolerance for failure decreases. In fact, in many cases failure has become unacceptable [CGP99]. Software has now become widely used in safety-critical systems such as nuclear power plants, air traffic control systems, medical instruments, weaponry, and embedded systems running in aircraft or automobiles.

The model checking process relies on building a finite state model of a system and checking that a desired property holds in the model. The check is performed as an exhaustive state space search, which is guaranteed to terminate since the model is required to be finite. In general, model checking and other finite-state analysis techniques have been very successful when used with hardware systems and less successful with software systems [CDH+00]. Implicit-invocation systems, along with other component-based systems, are especially challenging to model as finite state

machines and thus are an excellent class of software systems to study the problem of model checking software.

1.2 Problem

There are two reasons why model checking has not been successful when applied to software:

1. It is hard to construct a finite-state model of a software system because unlike with hardware, there is a larger gap between the artifacts produced by software developers and the artifacts that are accepted by model checkers. Often referred to as a semantic gap [HP99], the gap between artifacts has to be bridged by humans with little tool support. Thus, there is a possibility that spurious analysis results will occur because the finite-state model does not correspond with the software system.

2. The state spaces of the models constructed for software systems are often infinite or extremely large due to the use of variables ranging over infinite or large domains. For example, a system using a single integer can be in one of up to 4.2 billion possible states. The problem is exacerbated by the fact that the state space grows exponentially with the number of parallel processes in the system. Growth of a state space resulting from variables with large domains and high numbers of system components interacting is known as the state explosion problem [CGP99].

This thesis intends to alleviate both of these problems in the context of implicit-invocation systems. On the one hand, we will bridge the gap between artifacts by

automating parts of the process of generating the finite state model for a software system. By removing the human element in model generation we will decrease the occurrence of errors due to artifact conversion. On the other hand, we will apply optimization techniques such as abstraction to decrease the state space of the software models. Together the automation and optimization outlined in the following chapters will explore ways to make model checking a more effective and efficient method of analysis for non-trivial software systems.

1.3 Thesis

The primary goal of this research is to examine the application of automatic verification to the software analysis process. Specifically, we will be looking at how to automate the process of model checking event-based systems that are constructed using the implicit-invocation architectural style.

In an effort to choose a class of non-trivial software systems we decided to concentrate our efforts on implicit-invocation or publish-subscribe systems. These component-based systems are implemented using the implicit-invocation architectural style. One of the key features of this style is that it permits the easy integration of separately developed components. Implicit-invocation systems are challenging because of inherent concurrency that leads to state space explosion. Some examples of systems that use the implicit-invocation architectural style are Enterprise Java Beans, Sun's Jini [Inc99], and the HP SoftBench commercial tool kit [Ger90].

1.4 Contributions

It is the goal of this research to gain some insight into the larger problem of how to make model checking a better analysis and verification tool for software systems. Specific contributions include:

- Extend an existing approach proposed in [GK00]
- Evaluate the extended approach using simplified "real-world" implicit-invocation systems
- Identify optimization techniques

The organization of the thesis will be as follows:

- *Chapter 2:* We will provide background in the areas of architectural style and formal methods. Specifically we will discuss the implicit-invocation architectural style and the technique of model checking. In addition we will present the existing approach presented by [GK00].
- *Chapter 3:* We will outline event representation enhancements to [GK00] that will provide events the functionality of encapsulating data.
- *Chapter 4:* We will outline enhancements to the delivery policy representation in [GK00] and introduce a formal approach to thinking about implicit-invocation delivery policies in the context of propositional logic.
- *Chapter 5:* We will evaluate our extended approach. First, we will compare and contrast the approach in [GK00] with our extended approach by modeling

a set-counter example. Second, we will model two "real-world" systems to demonstrate the advantages of our extended approach.

- *Chapter 6:* It is impossible to model the "real-world" examples in Chapter 5 without using optimization techniques. This chapter focuses on optimization of the model, such as correctness preserving transformations and data abstraction, and optimization of the model checker tool.

- *Chapter 7:* The concluding chapter summarizes the thesis and discusses areas of future work.

We also provide appendices to complement the material presented in the chapters of this thesis:

- *Appendix A:* We provide a glossary of terminology associated with software engineering and more specifically formal methods and implicit-invocation. Common acronyms used throughout this thesis are also included.

- *Appendix B:* We provide the Document Type Definition (DTD) of our eXtensible Markup Language (XML) input representation of an implicit-invocation system. We use a DTD to specify the content and structure of elements within our XML input representation.

- *Apendices C, D, E:* We provided detailed results of the analysis from Chapter 5.

- *Appendix F:* We examine several variations on event delivery policies. This appendix supplements the material discussed in Chapter 4.

Chapter 2

Background

In this chapter we will first review implicit-invocation systems and several of the analysis techniques that have been applied to implicit-invocation systems. Section 2.3 will review existing work on model checking implicit-invocation systems. Our research is based on this work.

2.1 Implicit-Invocation Systems

2.1.1 Architectural Style

Software architectural styles are interesting in the context of software development because every software system that contains modularity has a style [Jac]. That is, the breakdown of the system into parts and the communication between them follows certain principles or objectives. In most cases the architectural style is accidental and is not planned. Often when developers choose a technique to organize a given software system they are unconsciously using a style. However, with the development

and identification of various architectural styles such as implicit-invocation, pipe-and-filter, and layered systems, developers can utilize a given architectural style which possesses desired properties. Although most use of architectural styles is not explicit [GS96], its presence in the majority of software systems make architectural styles an interesting tool for classifying software.

An architecture can be viewed using a common framework consisting of components and connectors. Components often encapsulate information or functionality while connectors describe the communication between components. Using this framework an architectural style can be defined as "...a vocabulary of components and connector types, and a set of constraints on how they can be combined" [GS96]. In other words, an architectural style constrains both the topology and the behavior of an architecture [Jac].

An instance of a style is an architecture that conforms to these topology and behavior specifications. Instances can often be variations on a strict implementation of the style thus making architectural styles a flexible concept. However, if an architecture does not obey the collection of inclusion rules specified in the style then it can not be considered an instance of the style. All of the instances of a particular style can be thought of as a family of systems. For information on instances of implicit-invocation see Section 2.1.4.

Implicit-invocation can be considered an architectural style because its instances all have a specific topology. Implicit-invocation systems consist of components which publish and subscribe to events. Components within this topology all follow specified behavior in the context of information sharing and method invocation. Components do not invoke methods directly, instead events are announced by components. These

event announcements implicitly invoke methods in other components. This occurs because other components can register for a given event by associating a procedure with it. Thus, when an event announcement occurs, all methods associated with the event are invoked [GS96].

Another example of an architectural style is a pipe-and-filter style [GS96]. Within this style components act as filters. Components read data streams from inputs and write data streams to outputs. The connectors which join the inputs of one component to the outputs of another are called pipes. Filters are independent of all other filters and possess no knowledge about other filters before or after them in the data stream. A specialization of the pipe-and-filter style is pipelining in which the topology of the filters is restricted to a linear sequence. A compiler is a common example of a pipe-and-filter. In a compiler program text is passed through a set of filters to produce executable code.

Layered systems are yet another example of an architectural style [GS96]. A layered system can be visualized as a hierarchy of layers in which each layer acts as a client to the layer below and provide service to the layer above. The connectors in a layered system are protocols that allow layers to interact. A topological constraint of this style is that a given layer can only interact with its adjacent layers. Layered systems are beneficial because they support designs using increasing abstraction levels, they support reuse of layers, and they support quick modifications since changing one layer only affects the adjacent layers. An example of a layered system is a protocol stack.

The implicit-invocation style, the pipe-and-filter style, and the layered system

style are all homogeneous examples of architectural styles. Heterogeneous architectures can also be constructed via hierarchical decomposition of both components and connectors.

2.1.2 System Overview

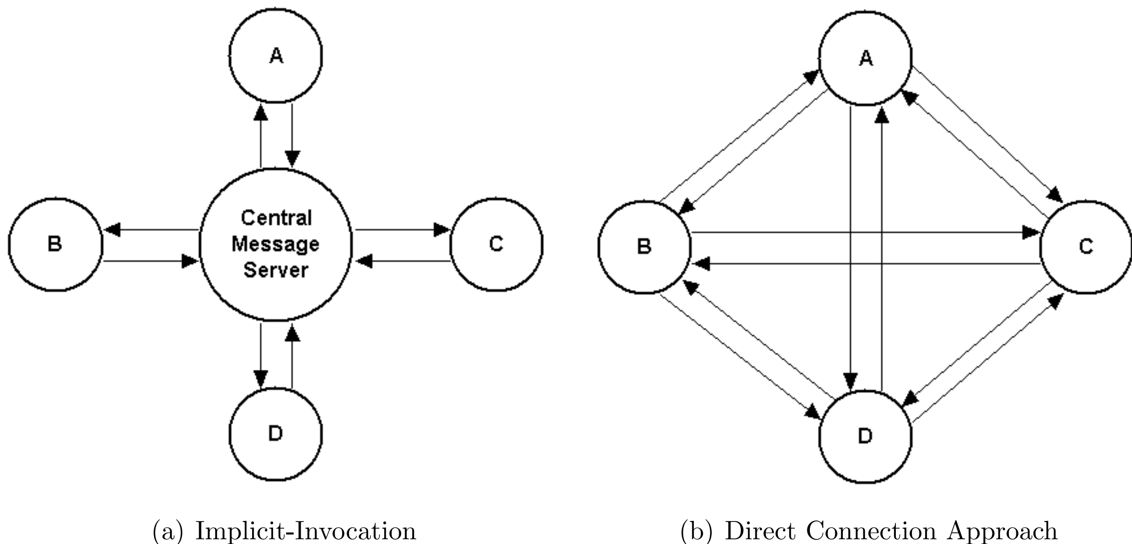
Implicit-invocation systems are event-based and provide component anonymity. Implicit-invocation systems have two primitives. First, components can announce or publish events. Second, other components can listen or subscribe to events that are announced.

A centralized message server or dispatcher is often used to keep track of which components subscribe to which events. Through the use of a delivery system the dispatcher will receive published events and use them to invoke the appropriate subscriber methods.

Independence of components is an important attribute of any implicit-invocation system. Components never know if any of the other components are sending or receiving events. In fact, components do not even know if the other components even exist to receive their events. This means that components are designed without knowing the identity of other components in the system. All information about which components are interested in which events is contained in the central message server. Since none of this information is stored in the components, it is easy to see how the components are independent [1]. Figure 2.1 demonstrates the difference between an implicit-invocation system and a system in which components refer to each other

[1] In a pure implicit-invocation system all components are independent. However, in many systems, components are allowed to communicate through a shared state. This creates dependencies between components that have a shared variable in common.

directly. The arrows represent communication channels. In the direct connection approach, each direct connection of components creates a dependency. In a worst case scenario all components are dependent on all other components. In Figure 2.1(b), we examine a small case of a system with four components labeled A, B, C, and D. The removal of one component, for instance, necessitates changes in all remaining components. In the implicit-invocation system all component-communication is done via event notification. In Figure 2.1(a), a component such as C can "register" interest in an event that can be announced by another component such as A.



(a) Implicit-Invocation (b) Direct Connection Approach

Figure 2.1: Comparison of Communication Channels

The loose coupling between the components makes it easy to make changes to an implicit-invocation system and reduces the implementation effort needed to add and remove components. For instance, the removal of component A in the implicit-invocation system in Figure 2.1(a) will not affect components B, C, or D. Loose coupling between components also eases adding new events, changing the announcer(s) of events, and changing the listener(s) who subscribe to a given event. In many

non-implicit-invocation systems it would be too much overhead for the components to be responsible for knowing who subscribes to their events and whom they receive events from. Having this amount of required component knowledge about the system would also reduce modularity. To summarize, the minimization of required component knowledge provides implicit-invocation the following advantages:

- *Increased re-usability of components:* it is easier to prepare a component used in one system for use in a different system or another version of the same system.
- *Ease of maintenance and extension:* a component can be added to or removed from the system more easily.

The minimization of required component knowledge also provides disadvantages for implicit-invocation systems:

- *Increased variability:* variability in event delivery can cause variability in the behavior of the system actions taken by components in response to event announcement. This creates an overall variability in system executions. For instance, event delivery may or may not preserve the ordering of the events, guarantee the delivery of duplicate events, or guarantee delivery within a certain time frame. How precisely event delivery is to proceed and which properties it is supposed to have is described in the delivery policy.
- *Decreased performance:* communication between components will experience decreased performance since components have less control over the timing of events and no knowledge of the identity of other components. Decreases in performance can also be attributed to the dispatcher possibly becoming a bottleneck since all events have to pass through the event dispatcher in order to

be delivered. Thus, a trade off exists between clean modular system design, with small interfaces consisting only of events, and the performance and ease of implementation that come with shared variables and increased component knowledge.

2.1.3 System Structure

An implicit-invocation system is characterized by the following six parameters:

- *Components*: The components of a system can be thought of as objects that encapsulate data and functionality. Components are usually accessible through well-defined interfaces. Depending on the system being discussed, the components may be implemented as processes, objects, or even servers.

- *Shared Variables*: Before one discusses events as a method of communication between the components of an implicit-invocation system, it is important to realize that information can also be communicated or shared through the use of shared variables. The use of shared variables for sharing information within the elements of an implicit-invocation system is fairly common. Shared variables can help to negate the possible performance problems of an implicit-invocation system. However, a major drawback of using shared variables in an implicit-invocation system is that the shared variables break the loose coupling of components by creating dependencies between components that share variables.

- *Events*: In implicit-invocation systems, events are the primary method of communication between components. An event can be as simple as a boolean flag or it can be as complex as a structured data type. Regardless of complexity,

events used in the system must be declared.

- *Event-method Bindings*: Event-method bindings represent the correspondence between events that are announced and the methods in a component instance that are invoked in response to these announcements.

- *Event Delivery Policy*: When an event is published, it is sent to the dispatcher or central message server. The dispatcher consults the delivery policy, which specifies how each event is to be transmitted. The event delivery policy can be thought of as a set of rules that define the announcement and delivery of events. One example of a delivery policy is when all events are sent out immediately upon being received by the dispatcher. Another delivery policy might instruct the dispatcher to deliver the events in a specified order based on priorities for different event types. Yet another delivery policy might suggest the all events are to be delayed and then randomly sent out in groups of an arbitrarily pre-determined size. There are many different variations for delivery policies. The dispatcher uses information from the delivery policy along with information about which components have subscribed to which events to send out the events.

- *Concurrency Model*: The final parameter of an implicit-invocation system is the concurrency model. This model determines if the system has a single thread of control or separate threads of control. Some of the variations on separate control threads are a single thread for each component, multiple threads for each component with concurrent invocation of different methods, or multiple threads for each component with concurrent invocation of any method.

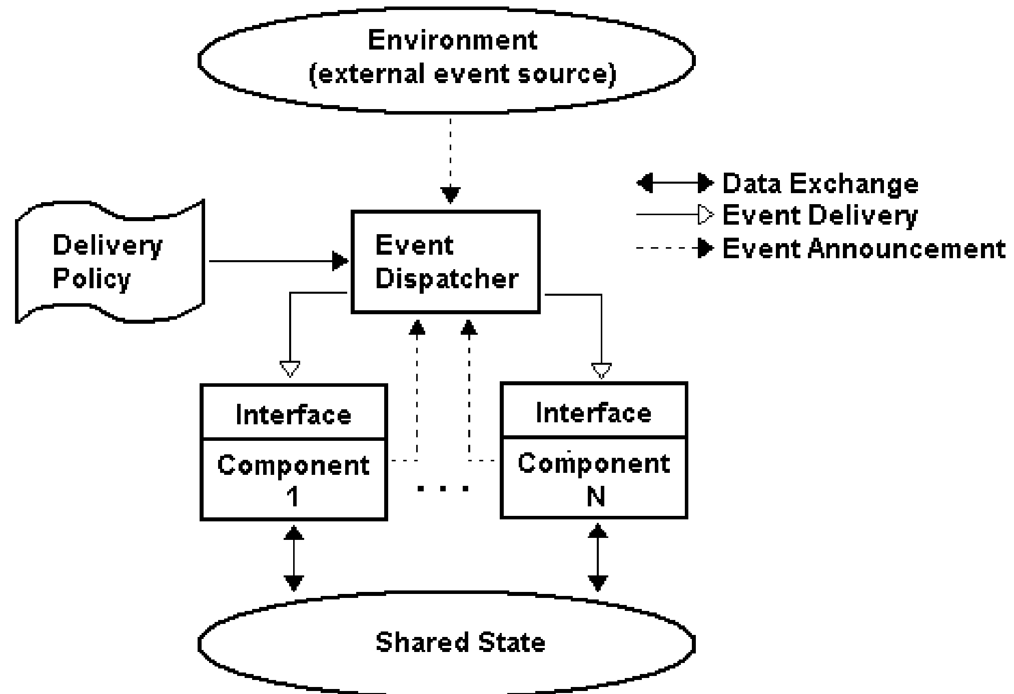


Figure 2.2: Implicit-Invocation System Structure

Figure 2.2 contains a diagram of how all of the parameters of an implicit-invocation system come together to form the overall system structure. Three main types of information exchange occur between different system parameters:

1. *Event Announcement*: The components in the system can announce an event provided that the type of event is one of the event types specified for the system. The environment can also act as an external event source and publish events.

2. *Event Delivery*: Upon receiving events from the components, the event dispatcher or central message server sends the events out to all subscriber components that have requested to receive a particular event type using the corresponding policy.

3. *Data Exchange*: Beside event announcement and delivery, data can be exchanged between the components through the shared state that contains any shared variables present in the system. There can also exist data exchange between the delivery policy and the shared state. This has been omitted from Figure 2.2 because we are only considering implicit-invocation systems with static delivery policies. If the delivery policy can access the shared state then it can change dynamically based on the state of certain shared variables. Data exchange also occurs between the delivery policy and the event dispatcher since the event dispatcher must consult with the delivery policy in order to determine how events should be delivered.

2.1.4 Examples

System designed using the implicit-invocation architecture, and more generally the publish-subscribe design pattern, are becoming increasingly used in a wide variety of application areas. Uses of systems that utilize the implicit-invocation mechanisms include [GS96]:

- Tool integration in programming environments.
- Separation of data presentation and data management in user interfaces.
- Support for incremental semantic checking in syntax-directed editors.

- Guarantee of consistency constraints in database management systems.

We will now examine two specific implicit-invocation systems: the HP SoftBench Commercial Toolkit [Ger90] and Sun's Jini and the JavaSpaces^{TM} Service [Inc99]. Other examples of publish-subscribe systems include Elvin [SA97], the Field system [Rei90], Gryphon [SBC^{+}98], LeSubscribe [PFL00], and Siena [CRW01].

HP SoftBench Commercial Toolkit

Component Name	**Component Function**
Development Manager	Manages the versions of files on which the other tools in the system operate.
Program Editor	Specifically designed for editing programs.
Program Builder	Simplifies the compile and link phases of the common edit-compile-link-debug loop into one component.
Static Analyzer	Aides the user in understanding source code and the structure of complex software systems.
Program Debugger	Proficient in software debugging tasks. Aides the user in understanding the behavior of complex software systems.
Integrated Help	A help system for all of the other components.

Table 2.1: HP SoftBench Component Functionality

Hewlett Packard SoftBench is a commercial software development toolkit that is implemented using implicit-invocation [Ger90]. The components in this toolkit include a development manager, program editor, program builder, static analyzer, program debugger, and integrated help. The functionality of each component is described in Table 2.1.

Consider the dependencies that could occur if the HP SoftBench components were all connected using direct communication channels, that is, without any use of the

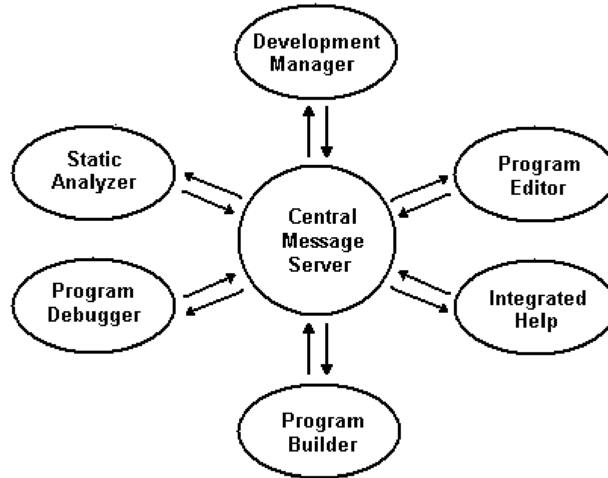


Figure 2.3: HP SoftBench with Implicit-Invocation Mechanism

implicit-invocation architectural style. The addition of a new component may thus require all existing components to be modified. In a large tool kit like SoftBench this would involve a lot of code modification and a considerable amount of time. Figure 2.3 shows how the HP SoftBench system acts with the use of the implicit-invocation architectural style. In this system diagram the arrows represent information exchange or communication channels. Although Figure 2.3 shows no dependencies, implying that all components are independent, some components might have dependencies because of shared variables.

Jini^{TM} and the JavaSpaces^{TM} Service

A Jini system is a dynamic distributed system consisting of devices and software components [Inc99]. All systems developed using the Jini architecture contain three attributes: infrastructure, a programming model, and services. The infrastructure

Attribute	Example	Description
Infrastructure	Discovery/Join	A service protocol that allows services to discover or join services of other members of the system and to advertise services to other members.
	Distributed Security	An extension of the security model of the Java platform that is integrated into Java Remote Method Invocation (RMI^{TM}) and developed for distributed systems.
	Lookup	A repository of services in which each service is stored as a Java object.
Programming Model	Leasing	A Renewable duration-based model for allocating resources between members.
	Transactions	A protocol that allows a group of entities to cooperate so that changes made occur to all entities or not at all.
	Events	An event notification interface based on the JavaBeans^{TM} event model.
Services	Printing	A service that can print from Java applications.
	Transaction Manager	Allows groups of objects to participate in the Transactions protocol.
	JavaSpaces^{TM} Service	Allows for the storage and communication of related groups of Java objects.

Table 2.2: Segmentation of Jini Architecture

consists of components that enable the construction of a Jini system. The programming model contains a set of interfaces that enable the construction of services which are entities within the system. Examples of these attributes used in the Jini architecture can be seen in Table 2.2.

The relationship between the implicit-invocation architecture and the Jini architecture is present within the programming model. The event notification interface used by Jini is based on the JavaBeans event model. It is developed in the implicit-invocation style and enables distributed event-based communication within a Jini

system. Objects in the system register interest in other object's events and receive event notification of the specified event types. The events programming model allows objects to communicate within the system and allows Jini systems to provide scalability guarantees.

The infrastructure of a Jini system is closely linked to the programming models. The lookup protocol uses the event notification interface along with the leasing programming model to ensure the delivery of events and the continued availability of the registered services. The event-based communication is also used by administrators to configure devices and discover problems within the system.

The JavaSpaces service [Inc00] is an instance of implicit-invocation because it is a Jini service that utilizes the event notification interface present in Jini. JavaSpaces also uses the leasing and transactions protocols.

2.2 Analysis Techniques

Architectural analysis can be defined as any software analysis technique that exploits the attributes of the architecture. We are taking a conventional approach to architectural analysis in the sense that we are applying the analysis technique of model checking to instances of the implicit-invocation style. Any further references to analysis will be in this context.

2.2.1 Issues in Analysis of Implicit-Invocation Architectural Style

In general, implicit-invocation systems are considered difficult to reason about and test making them difficult to analyze. For instance, there is no specific way to determine the effects of certain event announcements on the overall system. Moreover, it may be difficult to determine the effects of adding or removing components [DGJN98b]. In this thesis, we use the exhaustive state space exploration that is part of model checking to answer these kinds of questions.

The main reason why implicit-invocation systems are hard to analyze using model checking is the size of the state machine corresponding to a given implicit-invocation system. For a specific system the number of possible system executions is extremely large because the possibly large degree of parallelism and the system executions are not only affected by the events being sent and received, but also by the timing of these events. Timing has to be taken into consideration because when an event is published and eventually received will affect what the listener component does. In fact, the timing of events is just as important as the type of events. Some of the factors that affect timing are delays in delivery of events, ordering of events on delivery, delays in the announcement of events, and events that are not received at all.

2.2.2 Formal Methods

Formal methods can be defined as "mathematically-based languages, techniques, and tools for specifying and verifying systems" [CW96]. Formal methods are more comprehensive than testing or inspection. Formal methods are needed as a form of software verification because proving that a system satisfies a set of properties increases the

confidence in the system. There are two formal methods techniques that have been proposed for the analysis of implicit-invocation systems. The two methods are formal modeling and reasoning and model checking.

Formal Modeling and Reasoning

Formal modeling and reasoning involves the development of formal foundations that can be used to specify and reason about implicit-invocation systems [DGJN98a]. Specifically, a system is modeled as a collection of predicate logic formulas and theorem proving is then used to establish properties of the model. Advantages of this technique are that predicate logic is very expressive, is well known and understood, and excellent tool support for theorem proving is already in existence. A disadvantage of this technique is that it does not scale well. The size of a program that can be proven correct in this way is on the order of magnitude of a couple thousand lines of code [WVF96]. An additional concern with formal reasoning and verification is that it requires considerable proof machinery. Often while trying to prove an essential property, the theorem prover is required to find proofs for obvious or uninteresting properties. The amount of effort and expertise required to establish these properties is often the same as the level required to solve the essential properties. Also, the process of theorem proving requires user interaction as well as user expertise and training. It would be very hard for the average software developer to learn to use this method effectively.

Model Checking

Model checking is the main alternative to logical modeling and theorem proving in the formal analysis of implicit-invocation systems. In general, model checking relies on building a finite state model of a system and then checking that a desired property holds in the model. The check is performed as an exhaustive state space search, which is guaranteed to terminate since the model is finite [CW96]. The check is also efficient due to the use of sophisticated algorithms and data structures.

The state space in the context of model checking is a Kripke structure, also known as a Labeled Transition System (LTS) [Din00]. An LTS is a four tuple $M = (S, S_0, R, L)$ where

1. S is the finite set of states in the system.
2. S_0 is the set of initial states.
3. $R \subseteq S \times S$ is a total transition relation that defines all transitions between states in S. The relation is total because for every s in S, there exists a t in S such that $R(s, t)$ where $R \subseteq S \times S$. R is total for technical reasons that simplify the presentation.
4. $L : S \rightarrow 2^{AP}$ is a labeling function $\forall s \in S$. Each state is labeled with the atomic propositions (AP) that are true in that state. Specifically, for every p in AP and s in S, we have p in $L(s)$ if and only if p is true in s.

An LTS does not give any indications regarding the final states of a system. Final states are modeled with self loops. That is, $R(s) = \{s\}$.

The size of a program that can be model checked is usually specified in terms of the quantity of reachable states in the finite state machine and not in terms of the lines of

code in the original system. It is common for model checkers to handle systems with 100 to 200 state variables [CW96] and 10^{20} [WVF96] or even 10^{120} states [CW96]. A finite state model with 10^{1300} reachable states has even been verified [CGL92].

Model Checkers

There are two classes of model checkers: automaton model checkers and temporal model checkers.

In automaton model checking the system and the specification are both modeled as automata. The model checker compares the system's automaton representation to the specification automaton to determine if the system's behavior will conform to the specification [CW96]. This check typically involves determining whether the language accepted by the system automaton is a subset of the language accepted by the specification automaton [CGP99].

Temporal model checking is the technique that we used in this research. It was discovered independently by Quielle and Sifakis [QS81] and by Clarke and Emerson [CE81] in the early 1980s. In this type of model checking a system is modeled as a finite state transition system and any specifications are written in the form of temporal logic statements. Temporal logic is a variant of modal logic. Temporal logic formulas are interpreted over not just one state but sequences of states. The primary benefit of temporal logic is that it can be used to describe event ordering without explicitly introducing time [CGP99].

The model checker determines if a given finite state transition system is a model for a given specification. Temporal logic model checking can be thought of as a three-step process [CGP99]:

1. *Modeling.* Model the system as a finite state machine.

2. *Specification.* Express the specification that the system should satisfy as a temporal logic statement.

3. *Verification.* Input the model and the specification to a model checker. The output from the model checker will be "Yes" if the model satisfies the specification or "No" if the model does not satisfy the specification. If the answer is "No" and a counter example exists it will also be included in the output. The counter example can be used to identify the source of the violation and fix the bugs in the system.

The above three-step process is expressed visually in Figure 2.4 [Din00].

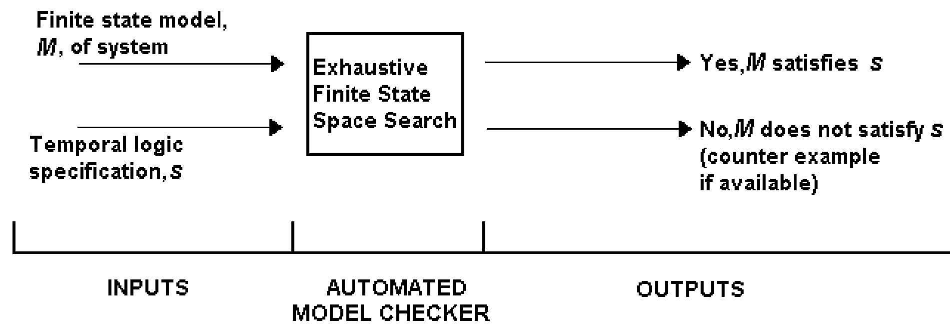


Figure 2.4: Temporal Logic Model Checking Process

The main advantages of using model checkers over other analysis techniques such as theorem proving are that model checkers are automatic and fast. Also, if the model does not satisfy the specification, a counter example is usually provided. Another benefit of model checking is that no user interaction is required during analysis. The main disadvantage of model checking in comparison to theorem proving is the state explosion problem and the fact that model checkers can not directly deal with systems

that have infinite state spaces[CW96].

Although model checking is considered a fully automatic process, it is only the verification step that is automatic. In most cases the user has to model the system as a finite state machine and express the specification in temporal logic by hand. It is this part of the model checking process that makes model checking very tedious and error-prone when applied to software systems. Another problem with model checking is that it is difficult and sometimes impossible to model the system adequately as a finite state machine. It is even more difficult to guarantee a small finite state model that can be used efficiently by the model checker. This is especially true for software systems, which are often more complex to model then hardware components. For instance, as new components are added to an implicit-invocation system the number of states in the finite state models grows at an exponential rate [McM92]. This is known as the state explosion problem and was discussed in Section 1.2.

Our research focuses on ways to automate the construction of finite state models of implicit-invocation systems. After the model is constructed it is checked using a model checker. For the purposes of this research we will be using the SMV or Symbolic Model Verifier model checker [CGP99]. There are several versions of SMV including NuSMV [CR98] and Cadence SMV [McM99a]. Cadence SMV will be used in this project because earlier work in this area has focused on developing an automatic tool that produces a model in a format acceptable to Cadence SMV. Another popular model checker, which could be used, is Spin [Bel97].

Temporal Logics

Temporal logics are often classified based on whether time has a linear or branching

structure [CGP99]. There are several forms of temporal logic including Linear Temporal Logic (LTL) and Computation Tree Logic (CTL). These two forms of temporal logic differ in the handling of branching in the computation tree. CTL allows for quantification over the set of paths from a current state as well as a quantification over the states in a selected path. LTL on the other hand allows quantification over states, but does not offer path quantification. Instead, each LTL formula is implicitly universally quantified over all paths originating from the initial states. In the context of model checking, the analysis of CTL formulas that do not hold will not always provide counter examples. For example, the analysis of a specification that states “Along at least one path, ϕ holds eventually”, that is EF ϕ, will not provide a counter example if EF ϕ does not hold. This is because the entire computation tree not one specific path is the actual counter example. In LTL, the analysis of a specification will always provide a counter example when the finite state model does not satisfy the specification.

For the purposes of this research we will only consider LTL as was done in [GK00]. We could easily extend this technique to include CTL properties since SMV translates LTL formulas into an equivalent CTL formula anyway [2]. The set of LTL formulas is defined inductively by:

1. Any atomic proposition is an LTL formula.

2. If α is an LTL formula then $\neg\alpha$ is an LTL formula.

3. If α and β are LTL formulas, then $\alpha \bullet \beta$ is an LTL formula where $\bullet$ is a boolean connective such as $\wedge$, $\vee$, $\rightarrow$.

[2]Note that this does not imply that all LTL formulas have an equivalent CTL form. Sometimes additional state variables and fairness constraints are added by Cadence SMV to the model in order to get an equivalent CTL formula.

4. If α and β are LTL formulas, then X α, G α, F α, and $(\alpha \cup \beta)$ are LTL formulas where X, G, F, and $\cup$ are temporal operators. See Table 2.3 for an informal explanation.

More details on temporal logic model checking using LTL are given in [CGP99].

LTL Operator	**Definition**
X ϕ	In the next state ϕ holds.
G ϕ	In all future states ϕ holds ϕ holds globally
F ϕ	In some future state ϕ holds ϕ holds eventually
$\phi 1 \cup \phi 2$	$\phi 1$ holds at least until $\phi 2$ does

Table 2.3: Common LTL Operators

In the discussion of temporal logic properties in Chapter 5 we use the safety-liveness taxonomy proposed by Alpern and Schneider [AS85]. In this taxonomy all properties fall into two separate classifications:

1. *safety properties:* something "bad" never happens during execution. In our discussion of properties we focus on safety properties of the form that in all states a "bad" event never occurs.

2. *liveness properties:* something "good" happens during execution. In our discussion of properties we examine variations of liveness properties including:

   - Something "good" happens infinitely often.
   - Something "good" happens once.
   - Something "good" eventually always happens.

Note the above definitions for safety and liveness are informal. Formal definitions can be found in [AS85].

The safety-liveness taxonomy was chosen because it is the most widely used in the classification of verification properties. Additionally, Alpern and Schneider show that any property can be represented as an intersection of a safety and a liveness property. Thus, by evaluating both safety and liveness properties we can state with reasonable certainty that our evaluation results can be extended to cover properties in general.

Alternatives to the safety-liveness taxonomy proposed in [AS85] include the taxonomy in [NC00] and the patterns approach [DAC99]. The Naumovich-Clarke taxonomy has a direct mapping to the safety-liveness taxonomy but has six classifications instead of two. The classification is based on the type of event sequences that the property contains. Specifically, if it contains only finite sequences, only infinite sequences, or both types of sequences. The classification is also based on the type of system executions that satisfy the property. Specifically, only finite executions, only infinite executions, or all executions. The patterns approach to property specifications for finite-state verification involves classifying properties based on recurring patterns. This concept is based on software design patterns. A complete list of the patterns in this approach can be found in [DAC99].

2.3 Model Checking Implicit-Invocation Systems

To the best of our knowledge, there is only one paper that specifically discusses the automatic analysis of the implicit-invocation architectural style [GK00]. As part of their research, co-authors David Garlan and Serge Khersonsky implemented a preliminary tool that automates parts of the process of developing a finite state model

for an implicit-invocation system.

2.3.1 Development of a Generic Implicit-Invocation Model Checking Framework

The approach proposed by Garlan and Khersonsky consists of two parts:

1. The development of a model for a reusable run-time infrastructure that implements event-based communication and the delivery policy.
2. The development of a model that captures component behavior specific to a particular implicit-invocation system.

Part 1 models five of the implicit-invocation system parameters: shared variables, events, event-method bindings, an event delivery policy, and a concurrency model. The system parameter modeled by part 2 is the components.

A problem with model checking implicit-invocation systems occurs in part 1 of the structural approach, which is concerned with a reusable run-time infrastructure. The problem occurs because some details of particular implicit-invocation systems such as the event delivery policy and the central message server or dispatcher differ from system to system. These differences create a problem in developing a reusable model. A solution to this problem is to develop a "generic implicit-invocation model-checking framework, that identifies the main structural elements of an implicit-invocation system suitable for model checking, and that permits one to specialize that generic model to match details of a particular implicit-invocation system" [GK00]. This process involves developing a reusable model that is flexible because it provides options for the

aspects of the model that may differ from system to system. Part 2 of the construction of a finite state model for implicit-invocation systems involves the development of finite-state approximations for each of the component behaviors or methods. In general the internal behavior of the components is not part of the generic model of the system and must be specified by the user.

The generic model is concerned with what happens between components. The development of finite-state approximations for the run-time infrastructure of the system is the main purpose of the generic model. The run-time infrastructure is responsible for overseeing the communication of components and includes the event dispatcher, the event queues, and the shared variable access.

In order to be able to model an implicit-invocation system as a finite state machine it is necessary that the following guidelines be followed:

- All data has to have a specified finite range
- The set of components and bindings have to be fixed at runtime
- The size of the event queue has to have a specified finite limit
- The sizes of the invocation queues have to have a specified finite limit

The smaller the above limits, the smaller the model, the smaller the resource requirements of the model checker, and the more likely the model checking process will complete successfully and not run out of memory or require a large quantity of time. Although all of these guidelines must be applied to some degree in order to guarantee a finite model, it is important to realize that too much abstraction will lead to a trivial model which does not adequately represent the original system.

2.3.2 Automated Tool

Overview

The process for automating the generation of the finite state models conceived by Garlan and Khersonsky can be seen in Figure 2.5. This process will be explained in more detail in the following sections.

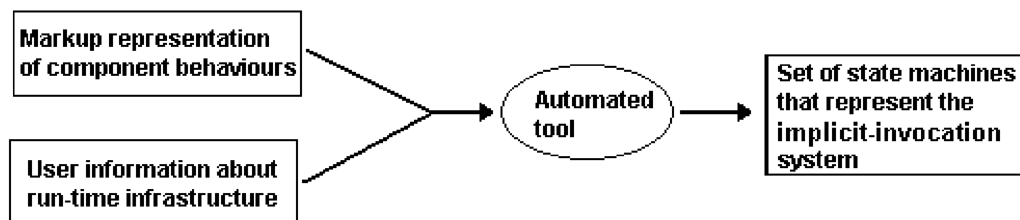


Figure 2.5: Semi-Automatic Analysis for Implicit-Invocation Systems

Required Inputs

According to Garlan and Khersonsky the first step in modeling an implicit-invocation system is to get input from the user describing the components and the way components are allowed to exchange information. For input the user must specify:

- A set of components, specifically component behavior
- A set of shared variables (optional)
- A list of events
- Event-method bindings

- Environment model (behavior specifications)

All of the above user inputs are implicit-invocation system parameters except the model of the environment. The environment is not considered part of the system and thus is not a parameter of the system. However, the environment is necessary from a modeling perspective because environment announced events are used to initiate the announcement of events by components in an implicit-invocation model.

The user also has to pick one of the following dispatch/delivery policies:

- *Immediate*: immediate invocation of subscriber methods
- *Delayed*: accumulate events before announcement to subscriber components

Finally, the user has to pick one of the following concurrency models:

- Single thread of control for all components
- Separate threads of control
  - Single thread for different groups of components
  - Single thread for each component
  - Multiple threads for each component
    - Concurrent invocation of different methods within a component
    - Concurrent invocation of any method within a component

Once the user has input the above information, an automated tool converts the information into a set of state machines as seen in Figure 2.5. This set of state machines can then be checked using a commercial model checker, such as the Cadence SMV model checker.

Model Generation

The automated tool builds the set of state machines that represents the implicit-invocation system by combining the state machines for component behaviors with a set of machine-generated state machines for the event notification. These machine-generated state machines are based on the user information provided. Specifically, the event notification state machines are generated in two separate parts:

1. Mechanisms that interact with the components of the system:
    - Handle event announcement and announcement acceptance
    - Handle event buffering and delivery
    - Handle method invocation and invocation acknowledgment
2. Mechanisms that implement the event delivery policy and event dispatch

Event notification is modeled in this way in order to separate aspects that do not change from system to system from aspects that do change. In general, all of the mechanisms specified in part 1 are constant for all systems and the mechanisms in part 2 vary depending on the event delivery scheme used. By splitting the event notification into these two parts the elements of the system that are not constant can be isolated. The mechanisms in part 1 are implemented as follows [GK00]:

- *Event announcement:* indicated by a binary announcement indicator and a set of announcement attributes such as priority.
- *Announcement acceptance:* each type of event has a binary announcement signal for each possible combination of an event's attributes (as specified in the announcement attributes).

- *Event Delivery:* implement event delivery signal using same method as used in event announcement.
- *Method invocation:* indicated by binary invocation signals and invocation arguments that are acquired from event arguments. The invocation arguments are optional.
- *Invocation acknowledgment:* use shared binary state variables. The variables are written by component methods and read by invocation queues.

The mechanisms in part 2 are implemented as follows [GK00]:

- *Dispatcher:* Reads announcement signals and updates the set of pending events immediately. These events are stored in the active events history data structure. The dispatcher then assigns delivery signals as specified by the delivery policy.
- *Delivery policy:* Continuously reading pending event information from the dispatcher and marking events to be delivered during a time cycle. This information is stored in the delivery directive data structure.

Although automated generation of the state model is beneficial, there is one major drawback to using an automated method. Automated methods involve the use of models that are machine-generated and fairly generic which means that the model is likely less efficient and probably has a greater state representation then a human-generated model.

Chapter 3

Event Representation Enhancements

Chapter 3 discusses event representation enhancements in the context of finite state machines. Specifically, finite state machines in SMV format. Chapter 3 is organized into 3 distinct parts. Section 3.1 looks at the Garlan and Khersonsky technique for modeling events. In Section 3.2 we discuss problems with this approach. We motivate the need for a new event representation by discussing the drawbacks of trying to model events containing data in [GK00]. In Section 3.3 we present our modifications to the original technique in [GK00]. We discuss in details the changes to the model that our modified approach requires.

3.1 Garlan & Khersonsky Technique

Component announced events have a name attribute and an optional semantics attribute. Garlan and Khersonsky represent component announced events in SMV as

boolean variables. These boolean variables are passed between component modules and the event dispatcher module via formal parameters. Note that types of components in an implicit-invocation system are modeled as modules in SMV and actual components are modeled as module instantiations. The terms component and module instantiation are used interchangeably.

Event announcement occurs from a component to the event dispatcher. If a component A announces a given event E then an output parameter of A associated with the announcement of E, called $paramOut_A(E)$ is set to true. The parameter $paramOut_A(E)$ is associated with an input parameter of the event dispatcher, $paramIn_D(E)$. The association between these formal parameters is defined in the main module of the SMV model. That is, $paramIn_D(E) := paramOut_A(E)$ is executed across every transition. The association between an output parameter of a component module and the input parameter of the event dispatcher is implemented as a one-to-one relationship. If the parameter for a given event announcement is true then the event dispatcher recognizes this as an event announcement and increments the pending delivery queue for that event.

Event delivery occurs from the event dispatcher to a component. Specifically, the event dispatcher delivers boolean events to appropriate subscriber components via formal parameters. When an event is delivered its corresponding formal parameter is set to true and it is removed from the delivery event queue in the event dispatcher component, which is then updated accordingly. An output parameter of the event dispatcher is associated with an input parameter for one or more subscriber components, resulting in a one-to-many relationship. In Figure 3.1, consider an event E sent from the event dispatcher to two subscriber components of event E, S_1 and S_2. In

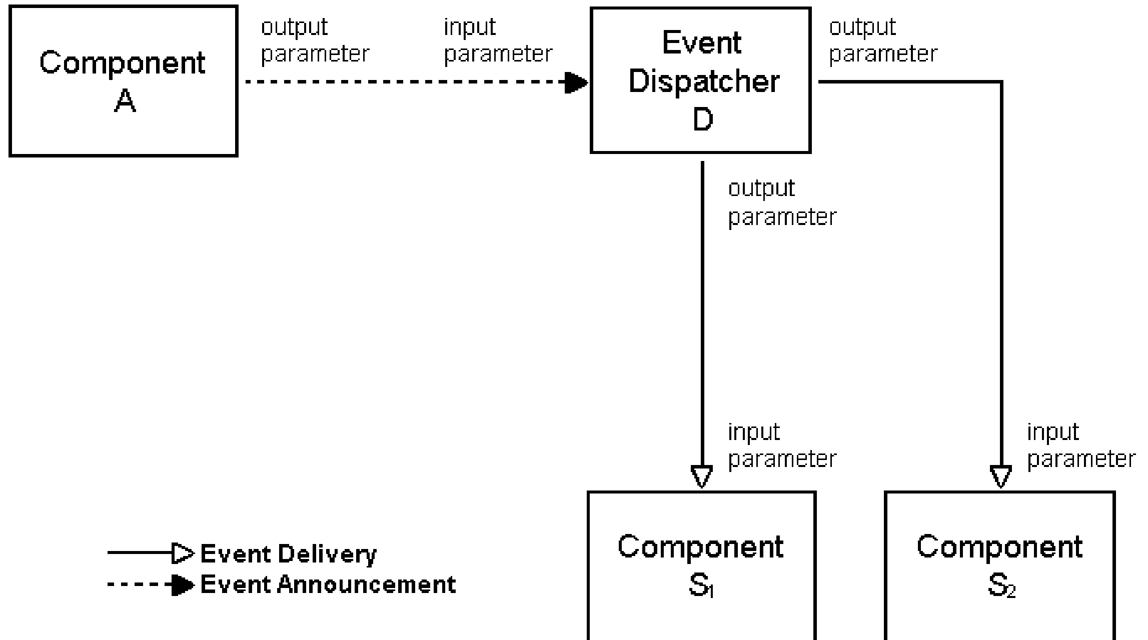


Figure 3.1: Announcement and Delivery for Component Announced Events

this case, the main module creates the association $paramIn_{S_1}(E) := paramOut_D(E)$ and $paramIn_{S_2}(E) := paramOut_D(E)$.

It is important to note that between event announcement and event delivery the delivery policy is used to determine when the delivery takes place. Variations on delivery policies are discussed in Chapter 4.

The other type of event is environment announced events that have a name attribute and an optional semantics attribute. Environment announced events also have two boolean properties: `always-announced` and `stops-eventually`. These properties are necessary because the environment is not modeled in the system. Also, these properties are unnecessary with component announced events because they are announced from within the methods of the component definitions. Thus, it is not

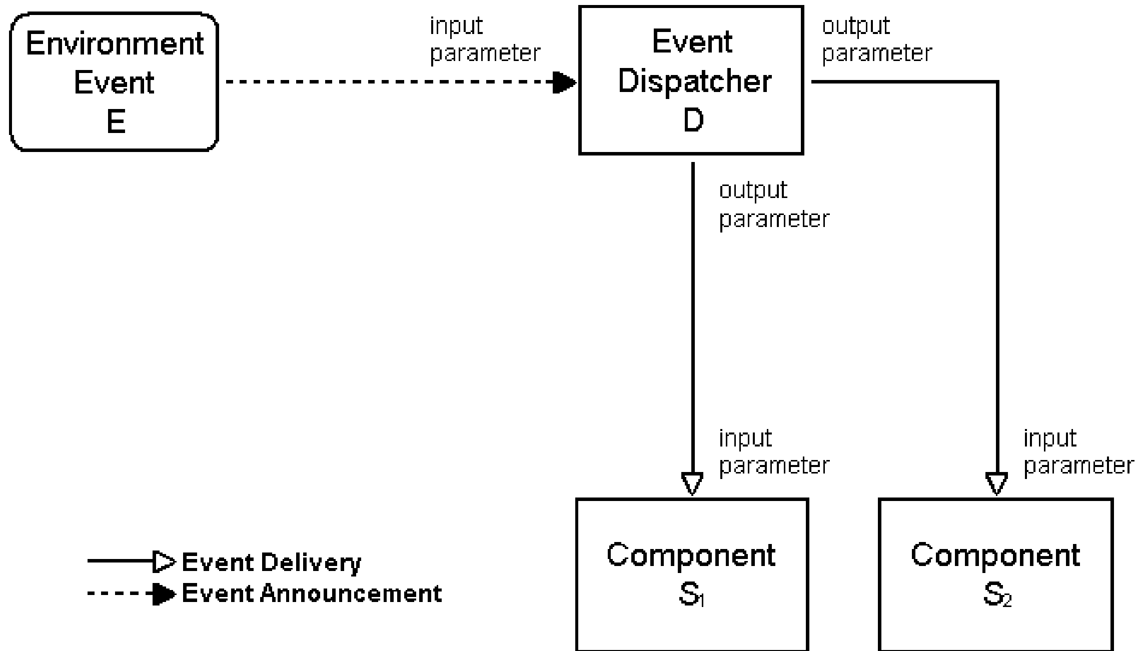


Figure 3.2: Announcement and Delivery for Environment Announced Events

necessary to have boolean properties to indicate if an event will eventually be announced and if the event will stop being announced once it has started. Environment announced events are also handled via boolean variables. There are 2 main differences in their announcement/delivery process in comparison to that of component announced events:

1. *Event announcement:* environment events are parameters of the main module and are thus connected directly to the event dispatcher formal parameters. For example, an environment announced event, E, is associated with a formal parameter of the event dispatcher in the main module through $paramIn_D(E) := E$. Event announcement is shown in Figure 3.2.

2. *Delivery Policy:* Since environment announced events are not true events in an implicit-invocation system they are not subject to the rules defined in the delivery policy. As a result, all environment announced events are delivered immediately upon being received by the dispatcher. The actual delivery of environment announced events from the dispatcher to the subscriber components is identical to event delivery for component announced events.

3.2 Problems with the Garlan & Khersonsky Technique

The technique of using boolean inputs and outputs that are sent from module to module via formal parameters has one major drawback: no information or data can be transmitted with the event. Thus, an event is limited in the sense that it contains no data except for its name. Any implicit-invocation system that relies on data being sent as part of the event cannot easily be modeled using the Garlan and Khersonsky technique.

Prior to discussing changes to the approach in [GK00] for modeling events with data it is important to discuss the different representations of the finite state model of an implicit-invocation system. The semi-automated approach described in the previous chapters of this thesis is implemented as shown in the model checking context in Figure 3.3. The user inputs the information about the implicit-invocation system in the eXtensible Markup Language (XML). XML is a metalanguage that is used to create specific markup languages for describing a particular document type [Fly02]. XML is not a fixed format predefined markup language. The XML representation is

then input to the translation tool and is used to generate a finite state machine in SMV format that is acceptable input to Cadence SMV.

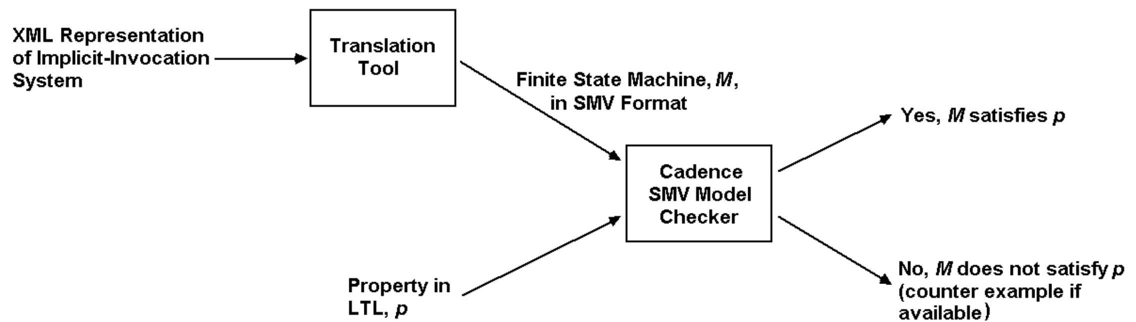


Figure 3.3: Implicit-Invocation System Representations in Model Checking Process

There are several possible approaches to modeling systems that have events containing data when using the Garlan and Khersonsky technique. However, the approaches are tedious and require non-automated modification of either the XML input representation or the SMV output representation of the model. The following subsections show the difficulty involved with non-automated modifications of either the XML or SMV representations of an implicit-invocation system.

3.2.1 Modifying an SMV Model

This approach proceeds as follows:

1. Assume events contain no data and model the system in XML with named events only.

2. Run the tool and generate the corresponding SMV model.

3. Modify this model to allow for the use of events that contain data.

This approach is not practical if there exists more than one event in the system that contains data because it becomes tedious and requires a major rework of the system model. In most cases it is almost easier to model the entire system by hand than it is to make the appropriate changes to the automatically generated model. The difficulty of modifying the model can be observed if one considers that modifying an event will require making changes to the event dispatcher and all components. Also, modifications by hand are error prone. The use of an inaccurate model may lead to spurious analysis results.

3.2.2 Modifying XML Representation

Consider a system which has an event called `number`. This `number` event contains as data an integer in the range 1..3. The XML specification could be modified so that a separate event is added for each possible event data value. In the case of the number event the system would contain three events labeled `number_1`, `number_2`, and `number_3`. The system would then have to be modified to handle the three new events. Problems with modifying the XML input representation include:

- The number of events would increase at a high rate if larger amounts of data elements were used.
- The number of events would also increase, but at a lower rate, if the range of data values was increased instead of the quantity of data.
- Creation of a lengthy redundant XML representation.
- Modifications are difficult since any changes will often have to be made multiple times.

- Only data ranging over a small domain can be easily represented as part of the event name. Therefore, data ranging over large domains cannot be represented in this approach and would have to be handled by modifying the SMV output as described in Section 3.2.1.

As a result of the above problems, this approach is beneficial for events that contain low numbers of event data elements with limited data value ranges and static values. Even when applicable this approach has undesirable qualities such as maintenance issues and redundancy.

3.3 Modified Approach: Adding Data to Events

It is obvious from the above analysis that events containing data, especially non-static data, cannot easily be represented in the Garlan and Khersonsky approach. Therefore, a new approach or a modified approach is needed. A modified approach using the Garlan and Khersonsky technique as a basis will have to support the following:

- Representation of events that contain static and non-static data
- Modification of formal parameters to allow for transmission of events that contain data
- Ability of component modules to read and write to event data
- Ability of event dispatcher module to store event data upon announcement until it is delivered to all listener components

These changes to the SMV representation of a model will also greatly affect the XML specification. As described in Section 2.1.3 an implicit-invocation system can

be characterized by six parameters. The modeling of the component and event parameters will have to be modified. A new technique for defining events in models will have to be developed and component definitions will have to be given the ability to read, write, and modify the data contained in events. The other four parameters will remain unchanged, including the event-method bindings.

It is clear that component announced events will need the ability to contain data. An example of when environment announced events with data are useful is when the data is used to initialize variables. However, it can be argued that it is unnecessary for events announced by the environment to have this ability. Unlike components, the environment can not modify the data in any events within the implicit-invocation system. Any data contained in environment events is therefore static from a software analysis perspective. Static data is defined as data that has a fixed value. It can be represented in the Garlan and Khersonsky grammar using the XML modifications outlined in Section 3.2.2.

3.3.1 Event Representation in XML

A Document Type Definition (DTD) is a formal description of a particular document type that includes the names of different types of elements, where the elements may occur, and how all of the elements fit together [Fly02]. We use a DTD to specify the content and structure of elements within our XML input representation of an implicit-invocation system. Our complete DTD is presented in Appendix B.

The portion of the DTD of a new technique for representing events that contain data is an extension of the original DTD for events. The original version of a component announced event in [GK00] was:

```
<!ELEMENT event EMPTY>
<!ATTLIST event
    name               CDATA    #REQUIRED
    semantics          CDATA    #IMPLIED>
```

The new version of a component announced event is:

```
<!ELEMENT event (event-data*)>
<!ATTLIST event
    name               CDATA    #REQUIRED
    semantics          CDATA    #IMPLIED>
```

The attribute list in the new version of a component announced event remains unchanged. The only difference is in the body of the event element. Any event that is described in the XML specification can now have zero or more event-data elements. Event-data elements are defined as:

```
<!ELEMENT event-data EMPTY>
<!ATTLIST event-data
    name               CDATA    #REQUIRED
    type               CDATA    #REQUIRED>
```

The definition of an event-data element is similar to the definition of a global variable. Event-data elements have no body but have two required attributes: name and data type. No initialization of any event data occurs in the event definition. Instead, event data is assigned values only in the components. An example of an XML definition for an event containing data is:

```
<event name="EventOne">
    <event-data name="DataOne" type = "0..10"/>
    <event-data name="DataTwo" type = "0..5"/>
    <event-data name="DataThree" type = "0..3"/>
</event>
```

The corresponding SMV representation of an event with data could use a structured[1] or a defined data type [2]. An example of a structured type version of the `EventOne` event would be:

```
MODULE EventOne()
{
    DataOne: 0..10;
    DataTwo: 0..5;
    DataThree: 0..3;
}
```

An instantiation of this module would look like:

```
eventOneInstance: EventOne();
```

The three fields in the structured data type instantiation could be referenced as:

```
eventOneInstance.DataOne
eventOneInstance.DataTwo
eventOneInstance.DataThree
```

An equivalent representation of the `EventOne` event using a defined type would be:

```
typedef EventOne struct {
    DataOne: 0..10;
    DataTwo: 0..5;
    DataThree: 0..3;
}
```

[1] A structured data type is a "...module with only type declarations and no parameters or assignments..." [McM99b] that is defined by the user.

[2] A defined data type is a "...special kind of module declaration with no parameters, and a slightly different syntax..." [McM99b] that is defined by the user.

In the defined type version of the `EventOne` event, an instantiation would be:

```
eventOneInstance: EventOne;
```

The fields in the defined type could be referenced using the same syntax as the structured data type version of the event. We have decided that the defined type is the more appropriate SMV representation for events because it is equivalent to the structured type version but does not refer to events as modules. A module representation of events might create confusion since components are also modules in SMV.

3.3.2 Formal Parameter Modifications

Recall, Garlan and Khersonsky used boolean formal parameters in the SMV modules for event announcement and delivery. To accommodate events containing data the formal parameters must be changed to defined event types. An additional modification to these defined types is that they should include a boolean flag to indicate announcement or delivery of an event. This flag would provide the same functionality as the boolean event types in the Garlan and Khersonsky approach. The flag will not be expressed in the XML representation but instead will be automatically added internally during the generation of the SMV model.

3.3.3 Event Dispatcher Modifications

For events that contain data, the event dispatcher will have to change event pending delivery queues from integer subranges to arrays of the defined event types. The arrays will be the same length as the maximum lengths of the original queues. It

would also be beneficial to keep the current event queue variables to represent the current size of the queue. When an event is received or delivered the contents of the queue are adjusted accordingly, as is the size of each queue.

3.3.4 Component Modifications

The main modification that will have to occur in regard to components is in the XML specification. An event announced by a component will still be declared in the same manner as specified in the document type definition below:

```
<!ELEMENT event-announced EMPTY>
<!ATTLIST event-announced
    name              CDATA #REQUIRED>
```

The component module will now have access to the data of events being announced. In the SMV model, data within the event will be referenced using the naming convention `announce_<event_name>.<data_name>`. This approach will also be used in the XML specification for events that are announced by a given component. A component should have both read and write access to data in events that it announces. Below is an example of a component writing to event data. Specifically in `MethodOne` of `ComponentOne` the value 1 is assigned to the `DataOne` data element of `EventOne`. After this assignment, `EventOne` is announced.

```
<component name="ComponentOne">

  <event-announced name="EventOne"/>

  <method name="MethodOne">
     <statement>
```

```
       <composed-statement>
         <statement>
           <assignment var-name="announce_EventOne.DataOne">1</assignment>
         </statement>
         <statement>
           <announcement>EventOne</announcement>
         </statement>
       </composed-statement>
     </statement>
  </method>
</component>
```

A component module will also have to access the data of events being received, that is, events that invoke a method of the component. The relationship between these events and the components is given in the event-method bindings. Any event that is bound via an event-method binding to a given component method should be read accessible within that component method. The data can be read in the form invoke_<method_name>_via_<event_name>.<data_name>. In the following example assume that the event-method bindings, specified elsewhere in the XML, state that method MethodTwo of ComponentTwo is associated with the delivery of EventOne.

```
<component name="ComponentTwo">

  <local-var name="localDataOne" type="0..10">0</local-var>

  <method name="MethodTwo">
     <statement>
       <assignment var-name="localDataOne">
       invoke_MethodTwo_via_EventOne.DataOne</assignment>
     </statement>
  </method>
</component>
```

It is important to notice that the events seem to be weakly coupled to the subscriber

methods. Ideally, a more abstract approach would be best. However, the approach used does generate models of reduced-size and it adheres to the properties of implicit-invocation.

Additionally, the pending invocation queues in the components will also have to be modified to handle defined data types using the same technique that is used to modify the event pending delivery queues in the event dispatcher. The pending invocation queues hold events that have not yet been announced.

Chapter 4

Delivery Policy Style Enhancements

Chapter 4 is organized into 3 distinct parts. In Section 4.1 we provide our own reasoning about implicit-invocation delivery policies including a categorization of policies in terms of complexity. This section is meant to provide an overview of delivery policies in general and is not specific to any past approaches to modeling the delivery policies of implicit-invocation systems. Section 4.2 looks at the Garlan and Khersonsky approach to modeling delivery policies. Finally, in Section 4.3 we combine the insight gained from our own reasoning in Section 4.1 with the approach to modeling delivery policies outlined in Section 4.2 to develop a modified event delivery policy representation.

4.1 Reasoning About Delivery Policies

Delivery policies are one of six parameters that characterize an implicit-invocation system. The event dispatcher consults the delivery policy during the delivery of all component announced events. There are many variations of delivery policies in existence. Examples of two variations on event delivery policies are presented in Appendix F. An efficient method of expressing the possible spectrum of policy styles is essential to the first step of model checking, the development of a system model. This chapter outlines a new approach to the representation of implicit-invocation delivery policies in the context of model checking.

4.1.1 What Information Affects Delivery?

Information[1] that affects event delivery falls into two categories: system attribute information and miscellaneous factors.

The primary system attributes that can affect the delivery of events in an implicit-invocation system include:

1. *Standard event types:* Different delivery mechanisms can be used depending on the event type.

2. *Environmental event types:* The announcement of an environmental event could be used as a signal for a change in the delivery policy.

3. *Shared data values:* Global variables can exist that contain information (system status, time, etc.) which can affect the delivery policy.

[1]We exclude the discussion of subscriber information in this section because we are concerned primarily with information that affects when and how delivery occurs, not to whom events are delivered.

4. *Number of events in event dispatcher queue:* The contents of the queue can initiate changes in the delivery policy. For example, events in a full event queue could be given priority and delivered immediately.

Although event delivery policies can be affected by a variety of different system attributes, information about the publisher or subscriber component can not affect the delivery policy. The use of such information in the delivery policy would violate the loose coupling of components specified in the implicit-invocation architectural style because the policy module would contain component information.

Miscellaneous factors, factors that are independent of the state of system attributes, are a second source of information in the delivery policy. Examples include the use of an immediate or random delivery mechanism or no delivery under certain conditions.

4.1.2 How Does Information Affect Delivery?

Information can affect how and when delivery takes place. On one hand, the state of system attributes primarily affects how events are delivered because the state of certain system attributes will invoke the use of miscellaneous delivery factors. Thus, different attribute states mean different miscellaneous factors are used. On the other hand, miscellaneous factors can affect when delivery occurs. For example, consider immediate and random delivery which are only based on miscellaneous factors. In the case of these delivery policies we can see that immediate delivery provides guarantees on when delivery occurs that are not provided when random delivery is invoked.

4.1.3 Categorization of Delivery Policy Information

Delivery policy information is categorized into two groups:

1. *Conditionals:* System attributes that affect how delivery occurs by influencing which delivery rules are invoked. Conditionals can be nested or combined with other conditionals. Examples of conditionals include the number of events of a given event type in the event dispatcher queue and the event type being delivered.

2. *Rules:* Delivery rules are miscellaneous factors that indicate when an event will be delivered. Examples include immediate delivery, random delivery, and no delivery.

4.1.4 Delivery Policy Categorization

The number of different event delivery policies for a given implicit-invocation system is infinite. Categorization of these different policies is important in order to gain greater understanding into policy patterns. Categorization also provides insight into how to represent delivery policies more efficiently in the context of model checking. The categorization scheme we have chosen is based on the complexity of the policy. A delivery policy is considered more complex as the levels of nested conditionals is increased. Specifically, delivery policies can be categorized from 0 to N where $N \in \mathbb{Z}^+$.

- *Level 0:* Event delivery policies that are based only on miscellaneous factors. These policies contain no decision structures and are not influenced by conditionals. How delivery occurs in a *Level 0* policy is static and predetermined. An

example of a *Level 0* delivery policy is an immediate or random policy applied to all events.

- *Level 1:* Event delivery policies that are based on miscellaneous factors and one conditional. The conditional is usually present in a decision making structure such as an if-statement. *Level 1* delivery policies are built on top of *Level 0* policies in the sense that a removal of the decision making layer will produce a *Level 0* policy. An example of a *Level 1* delivery policies is the mutually exclusive priority queue example in Appendix F where each event type is given a priority and only one event is delivered in each state. In this example the priority of the event type is the conditional and in the case where the conditional is satisfied, the immediate delivery rule is used.

- *Level 2:* Event delivery policies include two conditionals in a nested decision making structure. The environment event example in Appendix F is a *Level 2* delivery policy. This example is a *Level 2* policy because the presence of an environment event and the priority of event types provide two levels of conditions. Depending on the results of the conditionals, immediate delivery or random delivery is used as the miscellaneous rule.

- *Level N:* Event delivery policies include N nested decision making levels based on conditionals, where $N \in \mathbb{Z}^+$.

In a case where multiple levels of decision making exist the highest level of nesting will be used to determine the categorization. It is important to note that in the environment event example, a *Level 2* delivery policy, there is only one level of decision making for events delivered at random and there are two levels of decision making for

other event deliveries. The nesting of delivery policy levels in the environment event example can be seen in Figure 4.1.

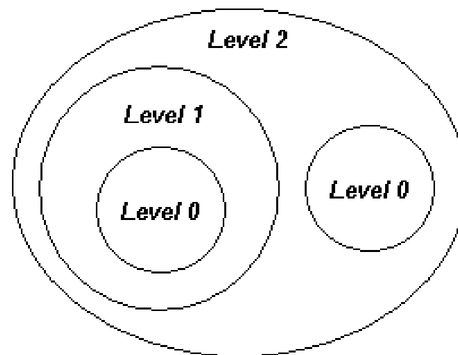


Figure 4.1: Delivery Policy Categorization Levels for Environment Event Example

4.2 Garlan & Khersonsky Technique

The approach in [GK00] for the event delivery policy of an implicit-invocation system is to allow for two distinct styles: immediate and random with guaranteed eventual delivery. The policy only uses miscellaneous factors and omits the use of conditionals. The delivery policy styles are restrictive because their purpose was only to provide some sample policies that could be used to reason about an automatic model generation approach for implicit-invocation systems.

The delivery policy is specified in the XML input as an attribute of the event-system:

```
<event-system name = "Example System"
          delivery-policy = <EventDeliveryPolicy>>
```

```
  <!-- Environment Announced Events -->
  <!-- Component Announced Events -->
  <!-- Component Definitions -->
  <!-- Global Data Variables -->
  <!-- Component Instantiations -->
  <!-- Event Method Bindings -->
  <!-- Properties to be Verified -->
</event-system>
```

In the above XML code <`EventDeliveryPolicy`> is replaced with `"Immediate"` or `"Random"`. When the automatic tool converts the XML input into the SMV output, the event delivery policy gets expressed in the event dispatcher module. If the delivery policy is `"Immediate"` then all event types are delivered according to the following case statement, with <`EventName`> replaced by the actual name of the event:

```
deliver_<EventName> :=
case {
        pending_<EventName> > 0 : 1;
        default : 0;
};
```

The above case statement automatically delivers the specified event provided that there are events of that type waiting to be delivered in the event delivery queue.

If the delivery policy is `"Random"` then all event types in the model are delivered in the style of the following example:

```
deliver_<EventName> :=
case {
        pending_<EventName> > 0 : {0, 1};
        default : 0;
};
```

In the above case statement the specified event may not immediately be delivered even if there are events waiting to be delivered in the event delivery queue. This is because the `deliver_<EventName>` flag will be set to 0 or 1 nondeterministically instead of 1 as in the `"Immediate"` example. One problem with the Garlan and Khersonsky implementation is that it does not guarantee that 1 will eventually be selected from {0,1}, as specified in [GK00]. Thus, there is no guarantee that an event will be delivered.

4.2.1 Limitations

The Garlan and Khersonsky approach to delivery policy representation is rigid and obviously does not cover all possible delivery policy styles. Any changes in the delivery policy technique must allow for an increase in the level of expressiveness and flexibility. By expressiveness we mean that more variations in delivery policies can be represented by the technique. A primary reason why Garlan and Khersonsky's approach lacks expressiveness and flexibility is the omission of conditionals. That is, it does not take into account any of the system attributes that can affect the delivery of events.

4.3 Modified Event Delivery Policy Representation

4.3.1 Theory Behind Modified Technique

Propositional logic is sufficient for representing a delivery policy of a specific event in isolation. Propositional logic formulas are built from a set of atomic propositions.

An atomic proposition is defined as an indecomposable declarative statement. The connectives in propositional logic are negation, conjunction, disjunction, implication, and equivalence. There are two types of propositions necessary in event delivery policies:

- *Boolean expressions:* The presence of an event can be represented in the form of a boolean expression, where $true$ means that an event of the specified type is waiting to be delivered by the dispatcher and a $false$ means that there is no event of that type waiting to be delivered. Also, shared data values that are boolean can be represented in the delivery policy as boolean expressions.

- *Relational expressions:* We want to support the comparison of, for instance, the number of pending events or shared data values. The relational expressions are acquired from comparing two integers using one of the following relational operators $=$, $<$, $>$, $\leq$, or $\geq$.

It is important to note that not every propositional logic formula represents a delivery policy. The policies we are considering can be expressed with propositional logic formulas of a specific structure. All policies are always of the following form:

$$
\begin{array}{l}
((guard_1) \Rightarrow (deliveryExpr_1)),\\
((guard_2) \Rightarrow (deliveryExpr_2)),\\
.\\
.\\
.\\
((guard_n) \Rightarrow (deliveryExpr_n))
\end{array}
$$

That is, the representation of a policy is a non-empty list of $(guard_i) \Rightarrow (deliveryExpr_i)$ formulas where $i \in \mathbb{Z}$. In an implicit-invocation system, the dispatcher executes the

delivery policy by determining the smallest value of i such that $guard_i$ is *true* and then making appropriate changes in the state to make the corresponding delivery expression, $deliveryExpr_i$, true. To indicate that a variable value within a delivery expression is in the next state or the current state we use primed variables. Primed variables are used in the formal specificaton language Z [Dil94] where unprimed variables indicate value in the current or before state and primed variables indicate value in the next or after state. The guards must be consistent and exhaustive. The guards in the delivery policy have to be consistent because they can represent a conjunction of the different levels of conditionals expressed in Section 4.1.4. The guards have to be exhaustive otherwise a situation could arise for which no delivery expression exists. The guards do not have to be disjoint because we choose only the first guard that is *true*. The guards within a delivery policy can overlap but only one delivery expression will be used. Additionally, the delivery expressions also have to be consistent. Consistency is required within the delivery expression and with the corresponding guard.

An example of a delivery policy represented in propositional logic is the event type-based priority queue in Appendix F. The representation of this policy is as follows:

$$
\begin{aligned}
&((pending_x > 0) \Rightarrow (deliver_x' \wedge \neg deliver_y' \wedge \neg deliver_z')),\\
&((pending_y > 0) \Rightarrow (\neg deliver_x' \wedge deliver_y' \wedge \neg deliver_z')),\\
&((pending_z > 0) \Rightarrow (\neg deliver_x' \wedge \neg deliver_y' \wedge deliver_z'))
\end{aligned}
$$

The above delivery policy is for a system with 3 event types: `x`, `y`, `z`. The priority queue based policy assumes no two events have equivalent priorities and that at most one event will be delivered in each state of the model. The priority values of the

events are x=1, y=2, z=3. If, for example, the number of events pending for x is 4, y is 2, and z is 2 and no other events are announced, the order of delivery of events should be x, x, x, x, y, y, z, and z. If an x is announced in the middle of the y's delivery then order should be x, x, x, x, y, x, y, z, and z. The XML representation of this delivery policy is given in Section 4.3.4 and the finite state model representation in SMV is given in Appendix F.

Additional examples of delivery policies represented in this propositional logic notation are presented in Appendix F and in Chapter 5.

4.3.2 Implementation of Delivery Policies in SMV

Event delivery policies can be described in terms of SMV as a sequence of decisions or conditionals that determine if a given event, x, will be delivered according to a delivery rule. The delivery rule is a miscellaneous rule that is applied in a given state when all decisions related to the delivery of the given x event succeed. It can be a rule such as immediate or random. Based on the rule a value 1 will be assigned to the variable `deliver_x` if the event will be delivered in the current state and a value of 0 will be assigned to `deliver_x` if the event will not be delivered in the current state.

The modified event delivery policy technique is capable of representing both event delivery conditionals and the actual event delivery assignment.

Event Delivery Decisions

Cadence SMV supports `if-statements`, `case statements`, `switch statements`, and `for loops`. To simplify our new approach we will only target `if-statements` and

will not target case-statements or for loops. Case statements are not needed in the representation of delivery policies since if-statements can be shown to be an equivalent representation of the delivery policy decision structures. Iterative constructs such as for loops are also not needed since they are essentially a short hand notation and provide no increased semantic expressiveness.

A policy containing propositional logic implication ($\Rightarrow$) can be implemented as an if-statement in SMV. Multiple embedded implications in a delivery policy can be represented as nested if-statements.

Event Delivery Assignments

There are two techniques for assigning values in SMV:

1. *Standard assignments:* semantically, standard assignments means that <variab-le> is equal to <expression>.

```
<variable> := <expression>
```

2. *Unit delay assignments:* allow a <variable> to have a specified initial value through the use of init operator. The execution of next(x) := e in state s causes e to be evaluated to some value v in s and x to have value v in the next state.

```
init(<variable>) := <expression>

next(<variable>) := <expression>
```

Either of these is sufficient for assigning a delivery value for a given event. Recall that event delivery assignments are controlled by delivery rules.

4.3.3 Encoding Policies in XML

The DTD of our technique consists of an element called the `delivery-policy`. This element contains at least one statement in its body and has an event name as an attribute. This attribute is optional and is used only if separate delivery policies are given for different events.

```
<!ELEMENT delivery-policy (policy-statement)>
<!ATTLIST delivery-policy
   event-name              CDATA     #IMPLIED>
```

The policy statement in the body of the delivery-policy can be a composed statement, an if-statement, a delivery statement, or a no-delivery statement. Within any of these statements all event names can be used. The specific pending delivery queue for a given event can also be referenced as `pending_<EventName>`.

```
<!ELEMENT policy-statement (composed-policy-statement |
                            policy-if-statement       |
                            delivery-statement        |
                            no-delivery-statement)>
```

The composed statement is used to allow for multiple statements within the body of the delivery-policy. It has the same purpose as the composed statement that occurs within the XML representation of a component method.

```
<!ELEMENT composed-policy-statement (policy-statement+)>
```

The if-statement has a condition as an attribute and has a truth statement. This conditional statement also has an optional false statement.

```
<!ELEMENT policy-if-statement (policy-statement,
                               policy-statement?)>
<!ATTLIST policy-if-statement
   condition               CDATA     #REQUIRED>
```

The delivery statement allows for an event specified in the body to be delivered. The event will be delivered according to a delivery rule that is specified as an attribute. Common delivery rules are "Immediate" and "Random". The specification of a rule as an attribute is optional with the default value being "Immediate".

```
<!ELEMENT delivery-statement (#PCDATA)>
<!ATTLIST delivery-statement
   rule                 CDATA     #IMPLIED>
```

The no-delivery statement has an empty body and no attributes. It is used when a situation is reached within the delivery policy where no events should be delivered.

```
<!ELEMENT no-delivery-statement EMPTY>
```

The delivery policy is integrated into the DTD for an event system as follows:

```
<!ELEMENT event-system (env-event+,
                        event*,
                        delivery-policy+,
                        component+,
                        global-data*,
                        component-instance+,
                        event-binding+,
                        property*)>
<!ATTLIST event-system
   name                 CDATA     #REQUIRED
   event-queue-size     CDATA     #IMPLIED>
```

In this new version of the `event-system` DTD the delivery-policy is no longer expressed as an attribute of the `event-system` element, as in Garlan and Khersonsky's approach, but is instead expressed in the body of `event-system` as an element.

4.3.4 Example using New Technique

The event type-based priority queue, specified in SMV, in Appendix F can be represented in XML as follows. This is the *Level 1* delivery policy example that is discussed in the context of propositional logic in Section 4.3.1, an informal description can be found there. This example demonstrates the structure and formatting of a delivery policy as specified in Section 4.3.3. Additionally, the example can not be expressed in [GK00] and thus demonstrates the increased flexibility of our technique over the delivery rules used in [GK00].

```
<delivery-policy>
  <policy-statement>
    <composed-policy-statement>
      <policy-statement>
        <policy-if-statement condition = "pending_x &gt; 0">
          <policy-statement>
            <delivery-statement rule="Immediate">x
            </delivery-statement>
          </policy-statement>
        </policy-if-statement>
      </policy-statement>
      <policy-statement>
        <policy-if-statement condition = "(pending_x = 0)
                                         & (pending_y &gt; 0)">
          <policy-statement>
            <delivery-statement rule="Immediate">y
            </delivery-statement>
          </policy-statement>
        </policy-if-statement>
      </policy-statement>
```

```
      <policy-statement>
        <policy-if-statement condition = "(pending_x = 0)
                                         & (pending_y = 0)
                                         & (pending_z &gt; 0)">
          <policy-statement>
            <delivery-statement rule="Immediate">z
            </delivery-statement>
          </policy-statement>
        </policy-if-statement>
      </policy-statement>
      <policy-statement>
        <policy-if-statement condition = "(pending_x = 0)
                                         & (pending_y = 0)
                                         & (pending_z = 0)">
          <policy-statement>
            <!--Delivery No Events -->
            </no-delivery-statement>
          </policy-statement>
        </policy-if-statement>
      </policy-statement>
    </composed-policy-statement>
  </policy-statement>
</delivery-policy>
```

Chapter 5, which focuses on an example-based evaulation of our model generation approach, presents additional delivery policies.

Chapter 5

Evaluation

In this chapter we provide an example-based evaluation of our modified model generation approach. A logical approach to presenting this evaluation would be to demonstrate each example using the Garlan and Khersonsky technique and our modified technique and then compare and contrast. However, this approach is not feasible since all but one of the examples can not be modeled using the Garlan and Khersonsky method.

We will first compare and contrast the two techniques in the modeling of a set-counter example. This demonstrates that both techniques achieve the same results on examples that can be handled by [GK00]. We will then proceed to model the Active Badge Location System (ABLS) and the Unmanned Vehicle Control System (UVCS) to demonstrate "real-world" systems that can be easily verified and analyzed using our approach. Note that the ABLS and UVCS systems can not be modeled in [GK00].

5.1 Set-Counter Example

5.1.1 Background

The set and counter example was originally presented in [SN90] and later discussed in [DGJN98a]. This was the primary example used by Garlan and Khersonsky to test their finite model building technique. The set-counter example is relatively small. It consists of only two components and four types of events. This example is beneficial in demonstrating the abilities of automated model generation. However, the example lacks the size and complexity needed to demonstrate that the technique is useful in the context of "real-world" situations.

5.1.2 Garlan & Khersonsky Implicit-Invocation Model of the Set-Counter Example

The set-counter example, gk_SC, is an implicit-invocation system from [GK00]. The model of the set-counter example is visually represented in Figure 5.1. There are 2 components in the system: `Set` and `Counter`. There are two environment announced events: `EnvAdd`, `EnvRemove`. There are also two component announced events `Insert` and `Delete`.

The `Set` component contains a set of values. The number of values in the set can be incremented or decremented, by one. The `Set` component contains two methods: `Add`, `Remove`. The `Add` method is used to increment the number of elements in the set by one and then to announce an `Insert` event, notifying any interested components of the additional element. The `Add` method is invoked via an environment

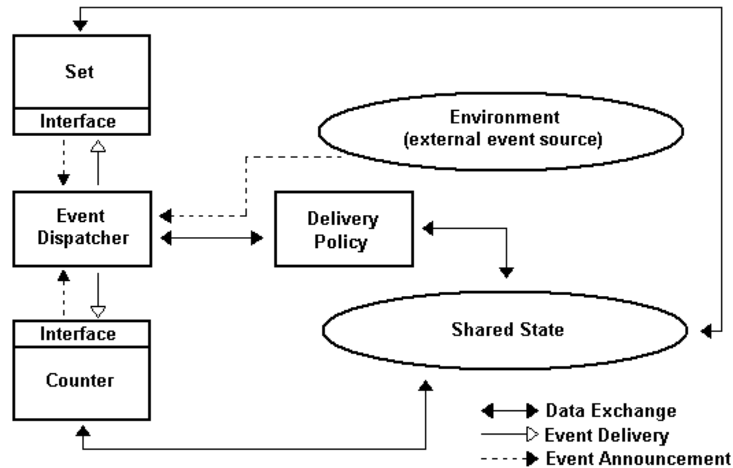


Figure 5.1: Implicit-Invocation Model of Set-Counter Example

announced event, `EnvAdd`. Similarly, the `Remove` method is invoked via an environment announced event, `EnvRemove`, and is responsible for removing an element from the set and announcing a `Delete` event. Both the `Insert` and `Delete` events are boolean and do not contain any data.

The `Counter` component is responsible for keeping count of the number of elements in the set. The communication of information from the `Set` component to the `Counter` component is done via the `Insert` and `Delete` events. The `Insert` event invokes the `Counter` method `Insert` which increments the counter value by one. The `Delete` event invokes the `Delete` method which decrements the counter by one.

The delivery of events in gk_SC can be done using one of two delivery policies, `"Immediate"` or `"Random"`. The `"Immediate"` delivery policy is a *Level 0* policy and is represented in the notation from Section 4.3.1 as:

$$(true) \Rightarrow (deliver_Insert' \wedge deliver_Delete')$$

The `Random` delivery policy is also a *Level 0* policy since no decision making levels are present. `"Random"` can be expressed as:

$$(true) \Rightarrow ((deliver_Insert' \vee \neg deliver_Insert') \wedge (deliver_Delete' \vee \neg deliver_Delete'))$$

The above formal representation of `"Random"` is equivalent to $(true) \Rightarrow (true)$ because there are no constraints.

From now on we will use gk_SC_immediate to refer to gk_SC using `"Immediate"` delivery and gk_SC_random to refer to gk_SC using `"Random"` delivery.

5.1.3 Our Implicit-Invocation Model of the Set-Counter Example

We will examine two versions of the set-counter example, generated using our approach. Although syntactic differences exist, the first version, mod_SC_1, is an exact semantic replica of gk_SC. The purpose of mod_SC_1 is to verify that the variations in syntax between gk_SC and mod_SC_1 do not affect the analysis of properties or affect the attributes of the model checking process. Attributes include model size and verification time. The second version, mod_SC_2, differs from gk_SC and mod_SC_1 in that its events contain data and it uses a more complex delivery policy. The goal of mod_SC_2 is to demonstrate extensions of the set-counter example that the Garlan and Khersonsky technique can not handle.

mod_SC_1

There are no significant differences between the XML representations of mod_SC_1 and gk_SC, except that syntactically the delivery policy is expressed in mod_SC_1 as an element within the `event-system` element as opposed to an attribute of the `event-system` element. For details regarding the difference in XML delivery policy representation see Section 4.3.3.

mod_SC_2

The set-counter example mod_SC_2 differs from mod_SC_1 and gk_SC in that it allows the addition and removal of multiple elements of the set. The XML representation of the `Insert` and `Delete` events is modified to:

```
<event name="Insert">
    <event-data name="numberOfElements" type = "1..4"/>
</event>

<event name="Delete">
    <event-data name="numberOfElements" type = "1..4"/>
</event>
```

The delivery policy also has an increased complexity and utilizes the size of the event queues to determine if `"Immediate"` or `"Random"` delivery should be used. This *Level 1* delivery policy is primarily used to demonstrate the limitations of Garlan and Khersonsky's approach using the tool in [GK00]. The policy is as follows:

- If the number of `Insert` events pending in the event dispatcher is greater than the number of `Delete` events pending then:
  - `"Immediate"` delivery is used for the `Insert` event.

    - "Random" delivery is used for the `Delete` event.

- Otherwise:

    - "Random" delivery is used for the `Insert` event.

    - "Immediate" delivery is used for the `Delete` event.

The formal representation is:

$$(pending_Insert > pending_Delete)$$
$$\Rightarrow ((deliver_Insert') \wedge (deliver_Delete' \vee \neg deliver_Delete')),$$
$$(pending_Insert <= pending_Delete)$$
$$\Rightarrow ((deliver_Insert' \vee \neg deliver_Insert') \wedge (deliver_Delete'))$$

5.1.4 Analysis of Set-Counter Example

For all of the versions of the Set-Counter example discussed above, we attempt to verify the following properties originally presented in [GK00]:

- *AlwaysCatchesUp:* In the LTL version of this property below, `theSet` refers to an instance of the `Set` component and `theCounter` refers to an instance of the `Counter` component. This property determines if the number of items in the set will always eventually be equal to the value stored in the counter.

```
G F (theSet.setSize = theCounter.counter)
```

- *SomethingInterestingHappens:* This property is used to verify that eventually the value of the counter will not be equivalent to zero, thus implying that items are eventually added to the set.

```
F (theCounter.counter ~= 0)
```

- *GlobalDataCheck:* This property contains no LTL operators which means that it applies only to the initial state. Within the example gk_SC, Garlan and Khersonsky include a global variable called `CopyOfSetSize`. This property is written to via the output correspondence variable `setSizeCopy` in `theSet` and is read by the input correspondence variable `setSizeCopy` in `theCounter`. This property verifies that when the system starts the input and output correspondence for the global variable, `CopyOfSetSize`, are equivalent.

```
theCounter.setSizeCopy = theSet.setSizeCopy
```

The results for gk_SC_immediate and gk_SC_random when compared to the results of mod_SC_1_immediate and mod_SC_1_random are equivalent. This demonstrates that the set-counter example can be modeled using the modified approach to achieve equivalent results to those presented in [GK00]. The results can be found in Section C.1.1, Section C.1.2, Section C.1.3, and Section C.1.4 in Appendix C.

The results for mod_SC_2 are also consistent with the expected results for a system with non-determinism in its delivery policy [1]. That is, the results were the same as the results of gk_SC_random and mod_SC_1_random. The results for mod_SC_2 can be found in Section C.1.5 in Appendix C.

[1] Any delivery policy with any random behavior is non-deterministic because `"Random"` delivery is represented as an arbitrary selection from the set {0,1} where 0 represents no delivery in the current state and 1 represents delivery.

5.2 Active Badge Location System

5.2.1 Background

The Active Badge Location System (ABLS) [WHFG92], developed at Olivetti Research Ltd., is an electronic tagging system for locating people in an office setting. Active Badges are considered an innovative solution to the conventional pager system that is often used for personal location. The main advantage that Active Badges have over pagers is that they allow for the location of the Active Badge wearer to be transmitted. In the case of pagers, the page sender relies on the response of the page receiver to call via telephone regarding their current situation. Some applications of the ABLS system are:

- A receptionist locating an employee without the use of a public-address system
- Automatic transfer of telephone calls

Active Badge Location Systems have application not only in office environments but also in hospitals, schools, and other public areas where locating people can be difficult.

The ABLS system contains three types of processes: Active Badges, sensors, and a main station. The Active Badges are carried by all people in the system. They emit a unique code via pulse-width modulated infrared signals every 15 seconds. The code is emitted for approximately a tenth of a second.

The sensors are part of a network and are placed throughout the physical location of the system. The sensors are controlled from the RS232 port of any workstation [WHFG92]. A sensor picks up output from badges in its range. The current system design supports the use of up to 128 sensors. Each sensor has the capability to buffer the 20 most recent sightings in a first-in first-out (FIFO) data structure.

It is important to notice that there is a 1/150 chance that the signals from two badges located in the same physical location will collide. To increase the chances in a heavily populated system that all people will be located, the badges will have slightly different frequencies and lose synchronization after a period of a couple of minutes [WHFG92].

The main station oversees the entire system by polling the sensors for badge sightings information, processing all information received from sensors, and displaying data visually. The FIFO data structures in the sensors allow the master station to multi-task since it does not need to grab badge sightings from all sensors in real time.

There are five commands that can be executed within the ABLS system:

1. *FIND (name)*: Provides the current location of the Active Badge holder with name *name*. If the Active Badge holder is currently out of the range of the system sensors then the most recent location in the past 5 minutes is returned.

2. *WITH (name)*: Locates an Active Badge holder and returns a list of other badge holders in the same location.

3. *LOOK (location)*: Allows for an analysis of Active Badge holders [2] specified location and returns all badge holders at the given location specified as a parameter.

4. *NOTIFY (name)*: Allows the Active Badge the facility of a pager and will notify the Active Badge holder via an audible or vibration notification.

5. *HISTORY (name)*: Provides a report of the location history of the specified

[2] The term Active Badge holders assumes that an Active Badge is always in the holder's possession. We do not consider cases where the Active Badge holder might not be carrying the device.

Active Badge holder. The time length for this report should be specified in the system and is directly dependent on the history data stored by the main station.

5.2.2 Implicit-Invocation Model of the Active Badge Location System

An implicit-invocation model of the Active Badge Location System will consist of three types of workstations:

- Main Workstation (exactly one)
- Request Workstation (one or more)
- Sensor Workstation (one or more)

In the context of implicit-invocation each type of workstation can be considered a component.

The main workstation has three primary responsibilities:

1. *Information Retrieval*: information on the location of Active Badges is received by polling of the sensor workstations.
2. *Information Storage*: Information on the current and previous locations of Active Badges is stored in an appropriate data structure.
3. *Command Execution*: Receive and fulfill the five commands described above in Section 5.2.1.

The request workstations randomly issue supported commands, such as Find and Look, based on environment announced events. The parameterized name and location information used in the commands is chosen at random within a request workstation. In reality, the request workstation could be any sensor workstation. However, for simplicity we have chosen to model it separate from sensor workstations. This is an acceptable assumption since it does not affect the integrity of the system.

The sensor workstations are responsible for sending polling results to the main workstation whenever a polling event is received. The information regarding the Active Badge locations is determined randomly. Active Badges are not modeled as components because in an implicit-invocation model of the system they would not communicate via the central event dispatcher. Instead, Active Badges communicate directly with sensor workstations via sensors. All information regarding Active Badges is sent via events to the main workstation from a sensor workstation. Since the addition of components in an implicit-invocation system causes exponential growth in the size of the model, we have chosen to model the sensor workstations in one component. This technique is a common optimization technique known as correctness preserving transformations that is discussed in Section 6.2.1. Figure 5.2 is a generalized representation of the ABLS implicit-invocation system and does not contain this optimization.

We have now defined the three component types: `MainWorkstation`, `RequestWorkstation`, `SensorWorkstations`. It is still necessary to define the event types present in the system as well as the event-method bindings. In discussing system bindings we will often refer to the component an event is associated with instead of the method within the component. This relationship is known as an event-component binding

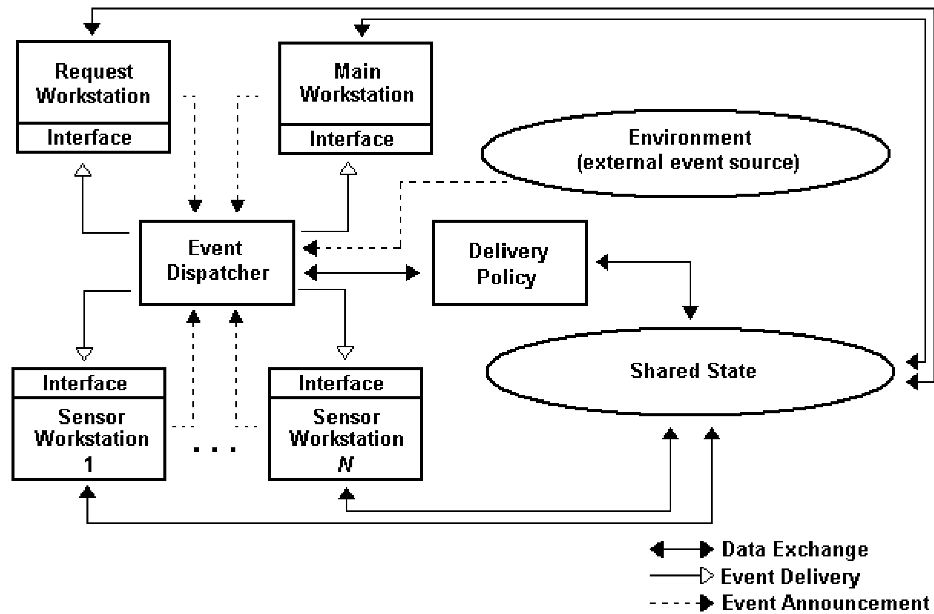


Figure 5.2: Implicit-Invocation Model of Active Badge Location System

and is a looser description of the relationship between events and the methods they invoke.

Event types for communication between the `MainWorkstation` component and the `SensorWorkstations` component are needed in the context of polling. The main workstation requests polling information from the sensor workstations and the sensor workstations send back information regarding the location of all Active Badge holders "close" to their workstation. The term "close" means that the Active Badge is communicating directly via a sensor with the sensor workstation.

- *Poll Event:* The `Poll` event is invoked by the announcement of an environment event, `EnvPoll`, which is guaranteed to always be announced and guaranteed

to always stop being announced. Environment events have two flags, as specified in Appendix B, that indicate if an event is always announced and always stops being announced. These flags are necessary because environment events invoke the announcement of component-announced events and their own invocation has to be handled by the system. All environment announced events in this system have these guarantees. The `Poll` event is delivered from the `MasterWorkstation` to the `SensorWorkstations` component. It has no event data and its delivery is used to invoke the announcement of the `PollResult` event.

- *PollResult Event:* The `PollResult` event is invoked via the announcement of a `Poll` event. This event is announced by the `SensorWorkstations` component and is delivered by the event dispatcher to the `MasterWorkstation`. A `PollResult` event is assumed to contain information regarding the location identification number of the sensor workstation sending the event and the Active Badge holders that are "close" to the specified sensor workstation. Since our models all use one component to represent all sensor workstations in the system, we need to send back information on the location of all Active Badge holders in the system. To simplify the `PollResult` event further we announce one `PollResult` for every person. Each of these events contains a `nameID` and a `locationID`. When a `PollResult` is delivered to the `MasterWorkstation` component it causes the database in the component to be updated. It does not invoke the announcement of any additional events.

Event types are also needed for the execution of the commands specified in Section 5.2.1. This communication takes place between a request workstation and the main

workstation. The following list of command events are paired by command. The first event in the pair is the request command sent from a request workstation to the main workstation and the second event is the result sent from the main workstation to the request workstation in response to the request command.

- FIND Command
  - *Find Event:* A `Find` event is invoked by an environment announced event, `EnvFind`. The `Find` event contains only one data item, the `nameID` of the person to find.
  - *FindResult Event:* A `FindResult` event is invoked via the delivery of a `Find` event. The master workstation determines the last known location in the database of the `nameID` given in the `Find` event. This `nameID` and the `locationID` from the database are the two event-data elements in the `FindResult` event.
- WITH Command
  - *With Event:* A `With` event is invoked by an environment announced event, `EnvWith`. The `With` event, like the `Find` event, contains only one data item, a `nameID`. The `nameID` identifies a person whose status will be checked in terms of the criterion "alone" versus "with others".
  - *WithResult Event:* A `WithResult` event is invoked via the delivery of a `With` event. The master workstation determines the last known location in the database of the `nameID` given in the `With` event. This location is then used to determine other Active Badge holders with an equivalent last known location. In the actual ABLS specifications, the entire list of Active

Badge holders with equivalent last known locations is stored as event data. However, from a model checking perspective we have reduced the size of the system by reducing the event data to contain the `nameID`, originally sent in the `With` event, and a boolean value, `withPeople`, that states if the person is alone or not.

- LOOK Command
  - *Look Event:* A `Look` event is invoked by an environment announced event, `EnvLook`. The `Look` event contains a data item, `locationID`. The `locationID` identifies a location and the system will check if the specified location is empty or contains Active Badge holders.
  - *LookResult Event:* A `LookResult` event is invoked by the delivery of a `Look` event. The `locationID` specified in the `Look` event is used to determine if there exist Active Badge holders at that location. In the actual ABLS specifications, the entire list of Active Badge holders at the location should be stored as event data. For similar reasoning as specified for the `WithResult` event, the `LookResult` event data is reduced to contain the `locationID`, originally sent in the `Look` event, and a boolean value, `peopleAtLocation`, that states if the location is empty or not.
- NOTIFY Command
  - *Notify Event:* A `Notify` event is invoked by an environment announced event, `EnvNotify`. The `Notify` event contains an event data item, `nameID`. The `Notify` event expects no results and performs the task of a standard pager.

- HISTORY Command
  - *History Event:* A `History` event is invoked by an environment announced event, `EnvHistory`. The `History` event contains a data item, `nameID`. The `History` event is similar to the `Find` event except that the announcer of a `History` event is interested in more information then just the last known location. The announcer is interested in the history of movements made by the Active Badge holder.
  - *HistoryResult Event:* A `HistoryResult` event is invoked via the delivery of a `History` event. The master workstation determines the last three known location in the database of the `nameID` given in the `History` event. The value three is a small arbitrary value that was chosen to help control model size. The `nameID` and an array `locationID` are the two event-data elements returned in the `HistoryResult` event.

We have clearly defined the component, event, and event-component binding parameters of an implicit-invocation system. A final parameter that needs definition is the concurrency model. The ABLS system uses a concurrency model in which a single thread of control is used for all components, including the event dispatcher.

We do not discuss the other two parameters of an implicit-invocation system in the context of this model. No shared variables are used in the system so a discussion on this parameter is unnecessary. The delivery policy is also not discussed here because we will discuss the use of specific event delivery policies in the context of the analysis in Section 5.2.3. Normally, for a system like ABLS only one consistent delivery policy would be needed. However, in our analysis we use varying delivery policies to demonstrate how the delivery policy can affect the verification of properties.

5.2.3 Analysis of Active Badge Location System

The analysis of the ABLS system is divided into six separate cases. Each case focuses on a specific aspect of the system. Case 1 focuses on the process of polling and specifically on the occurrence of `Poll` events and on the expectations associated with polling the locations of Active Badge holders. Case 2 and Case 3 are concerned with the correctness of event data after delivery and how the correctness is affected by variations in delivery policies. Case 4 and Case 5 are concerned with the correctness of event data prior to announcement. These cases are independent of the delivery policy. Case 6 verifies the correctness of data in multiple events.

Due to the size of the entire ABLS model, each case uses only a partial model of the system. Justification for using partial models is given in Section 6.2.

Case 1: Analysis of Polling

The model used to analyze polling and related properties regarding Active Badge holders contains only the `MainWorkstation` component and the `SensorWorkstations` component, representing multiple sensor workstations. The events announced in this partial system are the environment announced event `EnvPoll` and the component announced events `Poll` and `PollResult`. The `Poll` and `PollResult` events are delivered using a *Level 0* delivery policy. The policy is defined as:

$$(true) \Rightarrow (deliver_Poll' \wedge deliver_PollResult')$$

The request workstation and all of the events regarding the commands issued by the request workstation have been omitted from the model because they are outside the intuition discussed in Section 6.2.1.

We will refer to this partial system as poll_ABLS. Using the poll_ABLS system the following LTL properties were analyzed:

- *PollEventAlwaysAnnounced:* A property that verifies if eventually in all future states a `Poll` event will be announced.

  ```
  F G(Master.announce_Poll.flag = 1)
  ```

- *PollEventAlwaysEventuallyAnnounced:* A liveness property that verifies if eventually a `Poll` event will be announced infinitely often.

  ```
  G F(Master.announce_Poll.flag = 1)
  ```

- *Person2CanBeLocated:* A reachability property that verifies if a person will eventually be located in `Master` database. Note that `Master` is an instance of the `MasterWorkstation` component. For optimization reasons we only verify that `person_2` will eventually be located in the `Master` database. This property could easily be repeated for all people in the system.

  ```
  F(Master.database[2][0] ~= -1)
  ```

- *OnceLocatedPerson2AlwaysLocated:* A property that verifies that a person will eventually be located in the `Master` database and then stay in the system. The optimized version of this property is that `person_2` will eventually be located in the `Master` database and stay in the system.

  ```
  F G(Master.database[2][0] ~= -1)
  ```

If the ABLS system is modeled correctly, all properties except *PollEventAlways EventuallyAnnounced* should be false. That is, the system should always keep polling eventually and the system should not be able to guarantee the location of Active Badge holders in relation to the system. The reason that a person's location can not be guaranteed is that it is possible in our model for a person to move to a location outside of the area covered by the sensors. For example, if the ABLS system is used in an office setting, Active Badge holders will no longer be in the system once they leave the physical location of the office. Table D.2 in Appendix D shows that in fact all of these properties except *PollEventAlwaysEventuallyAnnounced* are determined to be false when the partial model is model checked using Cadence SMV.

Case 2: Find Command Correctness Verification

The model used to analyze the correctness of the `Find` command and the contents of the `FindResult` event requires a partial system consisting of the main workstation component and a request workstation component. The only events announced in this partial system are the environment announced event `EnvFind` and the component announced events `Find` and `FindResult`. The sensor workstations component and the polling events are omitted and replaced with an environment event `envDatabase` that updates the contents of the database directly. The database update is simply equivalent to the `PollResult` update that normally occurs in the sensor workstations. By updating the database directly we are able to reduce the number of events and the number of components, thus creating a smaller model. A smaller model means that the verification time for properties will be reduced. Using the partial system the following LTL properties were analyzed:

- *FindCorrectnessImmediate:* It should always be the case that if the current state in the master workstation is `sendFindResults` and the value of the database entry, requested in the `Find` event in the next state, is value v then the value of the `FindResult` event's `locationID` received by the request workstation in the same state is also v. Note that the `sendFindResults` state in the main workstation is invoked by a `Find` event. In the optimized version of the above general property, we verify the case where the `Find` event is interested in getting the location of `person_2`.

```
G((((Master.state = sendFindResults) & X(Master.database[2][0] = 1))
-> X(Request.invoke_receiveFindResult_via_FindResult.locationID = 1))
&(((Master.state = sendFindResults) & X(Master.database[2][0] = 0))
-> X(Request.invoke_receiveFindResult_via_FindResult.locationID = 0))
&(((Master.state = sendFindResults) & X(Master.database[2][0] = -1))
-> X(Request.invoke_receiveFindResult_via_FindResult.locationID = -1)))
```

- *FindCorrectnessInNextState:* This property is the same as the previous property with the exception that we are testing to see if the value received by the request workstation is equivalent to the database value in the next state, not in the same state.

```
G((((Master.state = sendFindResults) & X(Master.database[2][0] = 1))
-> X X(Request.invoke_receiveFindResult_via_FindResult.locationID = 1))
&(((Master.state = sendFindResults) & X(Master.database[2][0] = 0))
-> X X(Request.invoke_receiveFindResult_via_FindResult.locationID = 0))
&(((Master.state = sendFindResults) & X(Master.database[2][0] = -1))
-> X X(Request.invoke_receiveFindResult_via_FindResult.locationID = -1)))
```

- *FindCorrectnessInTwoStates:* This property is the same as the previous properties except that we are testing to see if the value received by the request workstation is equivalent to the database value in two states, not in the same state or the following state.

- *FindCorrectnessInThreeStates:* This property is the same as the previous properties except that we are testing to see if the value received by the request workstation is equivalent to the database value in three states.

In the above properties, we want to determine the number of states required for the request workstation to receive a value because this knowledge provides insight into guarantees we can place on the delivery of events. The number of states required to receive a value is directly dependent on the delivery policy in use. If the ABLS system is modeled correctly and uses a *Level 0* immediate delivery policy then the property *FindCorrectnessInNextState* should be the only property that is true. This policy can be formalized as:

$$(true) \Rightarrow (deliver_Find' \wedge deliver_FindResult')$$

If the system uses a *Level 0* random delivery policy, all properties should be false, since we can not guarantee that the `FindResult` event will be delivered in a specified number of steps.

$$(true) \Rightarrow ((deliver_Find' \vee \neg deliver_Find') \wedge (deliver_FindResult' \vee \neg deliver_FindResult'))$$

All of the results from Case 2 are in Sections D.1.2 and D.1.3.

Case 3: History Command Correctness Verification

Verification of the History command correctness is similar to the verification of the Find command except that multiple location IDs have to be checked – not just the most recent. The same partial system that is used for the Find command is used for the History command with the exception that the `EnvFind`, `Find`, and `FindResult` events are replaced by the `EnvHistory`, `History`, and `HistoryResult` events. Specifically, the LTL properties being verified are:

- *HistoryCorrectnessImmediate:* It should always be the case that if the current state in the master workstation is `sendHistoryResults` and the value of any database entry, requested in the `History` event in the next state, is value v then the value of the the `HistoryResult` event's corresponding `locationID` received by the request workstation in the same state is also v. Note that the `sendHistoryResults` state in the main workstation is invoked by a `History` event. In the optimized version of the above general property, we verify the case where the `History` event requests information on the database entry for `person_2`. This optimized property could easily be used for all people in the system.

```
G((((Master.state = sendHistoryResults) & X(Master.database[2][0] = 0))
-> (X(Request.invoke_receiveHistoryResult_via_HistoryResult
             .locationID[0] = 0)))
&(((Master.state = sendHistoryResults) & X(Master.database[2][0] = -1))
-> (X(Request.invoke_receiveHistoryResult_via_HistoryResult
```

```
             .locationID[0] = -1)))
&(((Master.state = sendHistoryResults) & X(Master.database[2][1] = 0))
-> (X(Request.invoke_receiveHistoryResult_via_HistoryResult
             .locationID[1] = 0)))
&(((Master.state = sendHistoryResults) & X(Master.database[2][1] = -1))
-> (X(Request.invoke_receiveHistoryResult_via_HistoryResult
             .locationID[1] = -1)))
&(((Master.state = sendHistoryResults) & X(Master.database[2][2] = 0))
-> (X(Request.invoke_receiveHistoryResult_via_HistoryResult
             .locationID[2] = 0)))
&(((Master.state = sendHistoryResults) & X(Master.database[2][2] = -1))
-> (X(Request.invoke_receiveHistoryResult_via_HistoryResult
             .locationID[2] = -1))))
```

- *HistoryCorrectnessInNextState:* This property is the same as the previous property with the exception that we are testing to see if the values received by the request workstation is equivalent to the database values in the next state, not in the same state.

- *HistoryCorrectnessInTwoStates:* This property is the same as the previous properties with the exception that we are testing to see if the values received by the request workstation is equivalent to the database values in two states.

- *HistoryCorrectnessInThreeStates:* This property is the same as the previous properties with the exception that we are testing to see if the values received by the request workstation is equivalent to the database values in three states.

To make the analysis in Case 3 different from that of Case 2 and to make it more interesting we examined the system with a *Level 1* queue based delivery policy that used immediate delivery. Two variations were examined, one in which the `HistoryResult` event has the highest priority, history_ABLS_queue_1, and one in which it did not, history_ABLS_queue_2. history_ABLS_queue_1 is defined formally as:

$$
\begin{array}{l}
(pending_HistoryResult > 0) \\
\Rightarrow (\neg deliver_History' \wedge deliver_HistoryResult'), \\
(pending_HistoryResult = 0 \wedge pending_History > 0) \\
\Rightarrow (deliver_History' \wedge \neg deliver_HistoryResult'), \\
(pending_HistoryResult = 0 \wedge pending_History = 0) \\
\Rightarrow (\neg deliver_History' \wedge \neg deliver_HistoryResult')
\end{array}
$$

history_ABLS_queue_2 is defined as:

$$
\begin{array}{l}
(((pending_History > 0) \\
\Rightarrow (deliver_History' \wedge \neg deliver_HistoryResult')) \\
\wedge((pending_History = 0 \wedge pending_HistoryResult > 0) \\
\Rightarrow (\neg deliver_History' \wedge deliver_HistoryResult')) \\
\wedge((pending_History = 0 \wedge pending_HistoryResult = 0) \\
\Rightarrow (\neg deliver_History' \wedge \neg deliver_HistoryResult'))
\end{array}
$$

In history_ABLS_queue_1, the system in which the `HistoryResult` has the highest priority and is delivered immediately, the results should be equivalent to the Case 2 example with immediate delivery. That is, *HistoryCorrectnessInNextState* should be true and all other properties should be false. In history_ABLS_queue_2 all properties should be false. This is because we can not make any guarantees on the number of states required to send the event. Tables of results for Case 3 are found in Sections D.1.4 and D.1.5.

Case 4: `Look` Command Correctness Verification

In Case 4 we verify reachability and liveness properties that examine the announcement of `Look` events. In addition we also verify the correctness of the `LookResult` event which is sent by the master workstation in response to a `Look` event announcement. Unlike Case 2 and Case 3, the verification of Look command correctness is delivery policy independent, thus a discussion of the delivery policy is not required. This is because we verify the correctness of the `LookResult` event within the announcing component, that is prior to being received by the event dispatcher.

- *LookReachability:* Eventually the `Request` instance of the `RequestWorkstation` component will announce a `Look` event.

```
F (Request.announce_Look.flag = 1)
```

- *LookLiveness:* Eventually the `Request` instance of the `RequestWorkstation` component will announce a `Look` event infinitely often.

```
G F(Request.announce_Look.flag = 1)
```

- *LookEventuallyAlwaysAnnounced:* Eventually the `Request` instance of the `RequestWorkstation` component will announce a `Look` in all future states.

```
F G(Request.announce_Look.flag = 1)
```

- *LookCorrectness:* Verifies that if a `LookResult` indicates that people are at location l, with a true value for `peopleAtLocation`, then the last known location of at least one of the people in the database is l. Additionally, this property verifies that if no people are at location l according to the variable `peopleAtLocation` in the `LookResult` event then the last known location of all people in the database is not l. To simplify this property we examine only the case where the Look command is interested in `location_1`. This property could be applied to all locations in the database.

```
G((((Master.announce_LookResult.flag = 1)
& (Master.announce_LookResult.peopleAtLocation = 1)) ->
((Master.database[0][0] = Master.announce_LookResult.locationID)
| (Master.database[1][0] = Master.announce_LookResult.locationID)
| (Master.database[2][0] = Master.announce_LookResult.locationID)))
& (((Master.announce_LookResult.flag = 1)
& (Master.announce_LookResult.peopleAtLocation = 0)) ->
((Master.database[0][0] ~= Master.announce_LookResult.locationID)
& (Master.database[1][0] ~= Master.announce_LookResult.locationID)
& (Master.database[2][0] ~= Master.announce_LookResult.locationID))))
```

LookReachability, *LookLiveness*, and *LookEventuallyAlwaysAnnounced* are false because the model specifies that `Look` events happen randomly and that their repeated future announcement is not guaranteed. *LookCorrectness* is true and verifies that the variable `peopleAtLocation` in the `LookResult` event is being set correctly by the master workstation, based on the information in the database. Results of the above properties are presented in detail in Section D.1.6.

Case 5: With Command Correctness Verification

Case 5 is similar to Case 4 except the reachability, liveness, and correctness properties are in the context of the `With` event instead of the `Look` event. The properties are defined as:

- *WithReachability:* Eventually the `Request` instance of the `RequestWorkstation` component will announce a `With` event.

  ```
  F (Request.announce_With.flag = 1)
  ```

- *WithLiveness:* Eventually the `Request` instance of the `RequestWorkstation` component will announce a `With` event infinitely often.

  ```
  G F(Request.announce_With.flag = 1)
  ```

- *WithEventuallyAlwaysAnnounced:* Eventually the `Request` instance of the `RequestWorkstation` component will announce a `With` event in all future states.

  ```
  F G(Request.announce_With.flag = 1)
  ```

- *WithCorrectness:* Property that verifies that if for person p the `withPeople` event data element of a `WithResult` is true then at least one other person is at the same location as p. Also, if for p the `withPeople` event data element of a `WithResult` is false then nobody is at the same location as p. The model for verifying this general case is too large, as a result we use a simplified version

where person p is actually `person_2`. This optimized version can be extended to cover all people within the system.

```
G((((Master.announce_WithResult.flag = 1)
& (Master.announce_WithResult.withPeople = 1)) ->
((Master.database[2][0] = Master.database[1][0])
| (Master.database[2][0] = Master.database[0][0])))
& (((Master.announce_WithResult.flag = 1)
& (Master.announce_WithResult.withPeople = 0)) ->
((Master.database[2][0] ~= Master.database[1][0])
& (Master.database[2][0] ~= Master.database[0][0]))))
```

As in Case 4, the reachability and liveness properties are false because the model specifies that `With` events happen randomly with no guarantee of repeated future announcement. For the same reason, *WithEventuallyAlwaysAnnounced* is also false. The correctness property is true and verifies that the variable `withPeople` in the `WithResult` event is being set correctly based on the information in the master workstation database. Results of the above properties are in Section D.1.7.

Case 4 and Case 5 verify the correctness of data within components and do not verify any correctness of data traveling between components. Therefore, these cases do not test the implicit-invocation delivery mechanism for events. The importance of Case 4 and Case 5 is the fact that these cases demonstrate how this technique can be extended to other software that does not conform to the implicit-invocation architectural style. There is a direct connection between correctness within a component and testing stand alone applications or command-line programs. Models for stand alone applications can be modeled as modules within SMV. The module input

parameters can be used to represent command-line parameters or sources of external information.

Case 6: Find and Look Command Correctness Verification

Case 6 demonstrates how correctness properties can be applied to multiple events, specifically the `Find` and `Look` events. There is only one property verified in this case:

- *FindLookCorrectness:* This property examines the situation where the `Master` instance of a `MasterWorkstation` component sends the results of a Find command in one state and then sends the results of a Look command in the next state. The Find command is requesting the location of person p and the Look command is verifying if people are at location l. This property verifies that if person p is at location l then the `locationID` in the `FindResult` event will be l and the boolean data `peopleAtLocation` will return true for location l in the `LookResult` event. Also, if person p and all other people in the system are not at location l then the `FindResult` should indicate this for person p and the `peopleAtLocation` variable in the `LookResult` event should be false. The simplified version of this property verifies the case for `person_2` and `location_1`.

```
G((((Master.state = sendFindResults) & X(Master.state = sendLookResults)
& (Master.database[2][0] = 1))
-> (X X(Request.invoke_receiveFindResult_via_FindResult.locationID = 1)
& X X X(Request.invoke_receiveLookResult_via_LookResult.peopleAtLocation)))

& (((Master.state = sendFindResults) & X(Master.state = sendLookResults)
& (Master.database[2][0] ~= 1) & (Master.database[1][0] ~= 1)
& (Master.database[0][0] ~= 1))
```

```
-> (X X(Request.invoke_receiveFindResult_via_FindResult.locationID ~= 1)
& X X X(Request.invoke_receiveLookResult_via_LookResult
              .peopleAtLocation = 0))))
```

The *FindLookCorrectness* event was verified to be true in the ABLS system. This property was verified using an immediate delivery policy:

$$(true) \Rightarrow (deliver_Find' \wedge deliver_FindResult' \wedge deliver_Look' \wedge deliver_LookResult')$$

The results of this property are in Section D.1.8.

5.3 Unmanned Vehicle Control System

5.3.1 Background

As a third example we have chosen to model a control system for unmanned vehicles (UVCS) developed by Stuurman and van Katwijk [SvK98]. The example has "real-world" applications since it is designed for the Maasvlakte port system. Maasvlakte, the main port for Northwestern Europe, is a large complex located to the west of Rotterdam.

Any control system developed for unmanned vehicles has to take into consideration the issues of scalability, evolvability, and on-line change capacities. The architecture presented in [SvK98] uses distributed processes communicating via the subscription model. The subscription model is similar to implicit-invocation except that processes subscribe to channels not events.

We will first briefly explain the control system for unmanned vehicles in terms of the subscription model before explaining how we modeled the control system for unmanned vehicles as an implicit-invocation system.

5.3.2 Subscription Model of a Control System for Unmanned Vehicles

The subscription model assumes that [SvK98]:

- The number of channels available is unbounded
- Defined data types exist for each channel

Under the subscription model, processes can write data to any channel and subscribe to listen to the data on any channels.

Each vehicle is controlled by a vehicle process that is given a long-term plan by a planner system. The vehicle process must deduce a detailed short-term plan from the given long-term plan. Besides the plans, the vehicle process must know the position of the vehicle it controls in a 2-dimensional plane, the velocity of the vehicle, and any important vehicle characteristics such as length and width.

The entire area that the vehicles move within is broken up into 10 x 10 regions. The regions are controlled by region processes. Vehicles in each region send their short-term plans to that region's channel and listen to the short-term plans of other vehicles in their region and the eight surrounding regions.

Vehicle processes are comprised of four objects: vehicle state, transmitter, receiver manager, and receiver. The vehicle state object constructs the current vehicle information including the rule version, the position, identity, speed, and short-term plan.

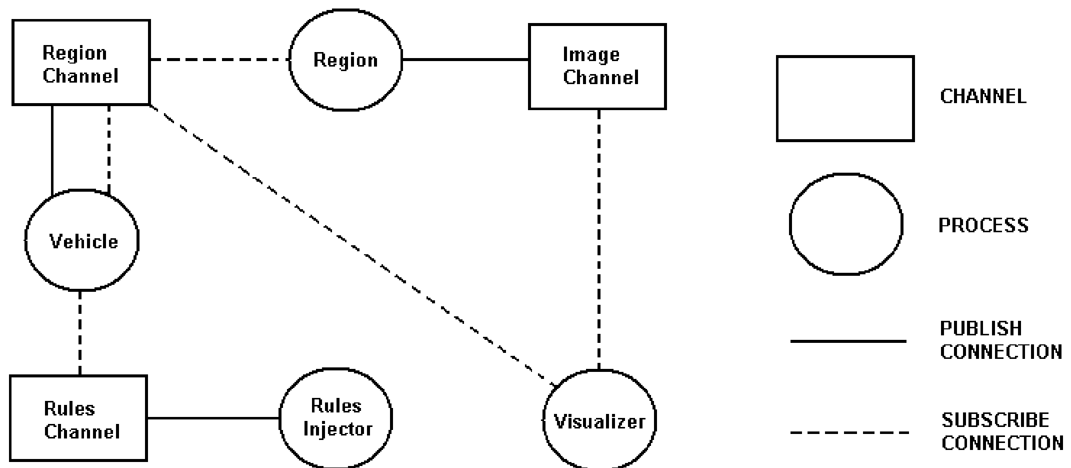


Figure 5.3: Subscription Model of Control System for Unmanned Vehicles

The transmitter object transmits the vehicle information to the appropriate region channel. The receiver manager object determines to which channels the vehicle process should subscribe. Finally, the receiver object listens for information from other vehicle processes in the appropriate regions by connecting to the receiver manager object. In the context of the subscription model of the UVCS system, the details of the vehicle processes are hidden. Note, this is also the case in the implicit-invocation model presented in Section 5.3.3.

Region processes listen to their region channel and collect data on all vehicles in their region. This information is compiled to create a summary of information that is sent to a visualizer process through the image channel. The visualizer gathers the summary information from all regions, using the image channel, and displays it externally. The visualizer also has the ability to zoom in on a region process by listening to the region's channel.

To avoid collisions the system supports a set of traffic rules for the movement of vehicles. The version of the rules being used can be updated by the rules injector process, which publishes new rules to the rules channel. Vehicle processes listen to the rules channel for new versions of the traffic rules. To avoid conflict in the event of a possible collision, vehicles send the version of the rules they are using along with their short-term plan to the region channel. In the case of a possible collision the traffic rules are used to decide which vehicle should wait. In the event that vehicles are using different rule versions a default rule is used instead.

The relationship between the distributed processes and the channels of the subscription model for the UVCS are visible in Figure 5.3.

5.3.3 Implicit-Invocation Model of a Control System for Unmanned Vehicles

In our implicit-invocation model the processes of the UVCS system will be referred to as components. There will be only one visualizer component, only one rules injector component, at least one region component, and possibly multiple vehicle components. Figure 5.4 is a generalized representation of the UVCS implicit-invocation system that clearly shows the components of the system.

Events and event-component bindings in the UVCS system are listed below.

- *RegionInfo Event:* The `RegionInfo` event is announced by a `Region` component and contains a summary of data collected on all vehicles in the publisher region. Additionally, a region identification number is also included as event data. The `Visualizer` is the only component that subscribes to the `RegionInfo` event.

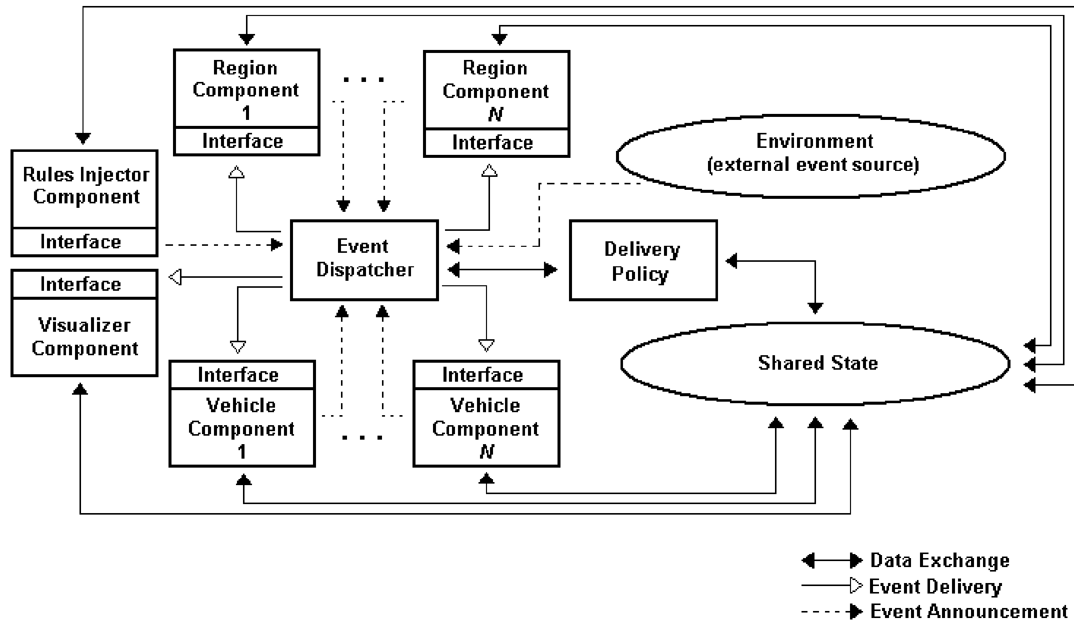


Figure 5.4: Implicit-Invocation Model of Control System for Unmanned Vehicles

- *VehicleInfo Event:* The `VehicleInfo` event is announced by a `Vehicle` component. The event data sent with the event includes the identification number of the vehicle, the region identification number of the region that the vehicle is currently in, a short term plan for the movement of the vehicle, and a version of the traffic rules that the vehicle is currently using. For the purpose of keeping the model small we will not include additional information such as vehicle speed and size in the event data. As a result of this information exclusion, all vehicles are assumed to occupy only one "square" in a region grid and to move at a constant speed. Other vehicles as well as the region that the vehicle is occupying subscribe to the `VehicleInfo` event.

- *RulesInfo Event:* The `RulesInjector` component announces the `RulesInfo` event. All `Vehicle` components subscribe to this event. The `RulesInjector` event is used by the UVCS to announce new traffic rules to be used in the case of possible collision. The version number of the traffic rules is also announced with the `RulesInfo` event.

For the UVCS system, the concurrency model, an additional parameter of an implicit-invocation system, is a single thread of control for each component.

The delivery policy parameter will be discussed on a case by case basis in Section 5.3.4. The other implicit-invocation parameter, shared variables, is excluded because no shared variables exist in the UVCS system.

5.3.4 Analysis of Unmanned Vehicle Location System

Case 1: Rule Version Synchronization

This case is a correctness case and specifically checks the status of rule versions within the vehicles. The check determines if variations exist between rule version and checks to see if all vehicles are using the most current version of traffic rules published by the rules injector component.

- *ruleVersionConsistent:* This property verifies within a UVCS system with 5 vehicles that the rules version for each vehicle will always be equivalent with the rules version of all other vehicles.

```
G ((Vehicle1.ruleVersion = Vehicle2.ruleVersion)
& (Vehicle1.ruleVersion = Vehicle3.ruleVersion)
& (Vehicle1.ruleVersion = Vehicle4.ruleVersion)
```

```
&(Vehicle1.ruleVersion = Vehicle5.ruleVersion))
```

- *ruleVersionCurrentandConsistentAlways:* This property verifies that within a 5 vehicle UVCS system that all of the rules versions used by the vehicles are equivalent to each other and to the `currVersion` variable in the `RulesInjector` component instance.

```
G ((Vehicle1.ruleVersion = theRulesInjector.currVersion)
& (Vehicle2.ruleVersion = theRulesInjector.currVersion)
& (Vehicle3.ruleVersion = theRulesInjector.currVersion)
&(Vehicle4.ruleVersion = theRulesInjector.currVersion)
&(Vehicle5.ruleVersion = theRulesInjector.currVersion))
```

The *ruleVersionConsistent* property is true when using immediate delivery, since then all vehicles are guaranteed to get the updated rules version in the same state, thus the value of `ruleVersion` in each vehicle component will be consistent. The delivery policy is defined formally as:

$$(true) \Rightarrow (deliver_RulesInfo1' \wedge deliver_RulesInfo2' \wedge deliver_RulesInfo3' \\ \wedge deliver_RulesInfo4' \wedge deliver_RulesInfo5')$$

The *ruleVersionCurrentandConsistentAlways* property is false because it is possible that the variable `currVersion` in the `RulesInjector` is updated but the updated version has not yet been delivered to the vehicle components. The results of these properties can be found in Section E.1.1.

Case 2: Regional Vehicle Transfer

The vehicle transfer case is concerned with the movement of vehicles across region boundaries. The case is used to ensure that when a vehicle moves from one region to another the change is handled correctly. Figure 5.5 demonstrates the possible movement of an unmanned vehicle in Case 2. Each region is modeled as a 5 x 5 grid. Each location in a region grid is denoted by an (x,y) pair with the pair (1,1) being in the bottom left hand corner of a region. Additionally, the vehicle is moved by modifying the direction and then applying the direction to the current location. The directions are 2 for East, 3 for North, 4, for West, and 5 for South. The vehicle movement is generated randomly and does not follow a preset path.

- *regionalMovement:* This property verifies that an unmanned vehicle that starts at location (3,3) in Region 1 will successfully move into a boundary region when the appropriate conditions are met.

```
G(((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.xpos = 5 & Vehicle1.direction = 2)
-> (X (Vehicle1.currRegion = 2)))
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.ypos = 5 & Vehicle1.direction = 3)
-> (X (Vehicle1.currRegion = 3)))
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.xpos = 1 & Vehicle1.direction = 4)
-> (X (Vehicle1.currRegion = 4)))
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.ypos = 1 & Vehicle1.direction = 5)
-> (X (Vehicle1.currRegion = 5))))
```

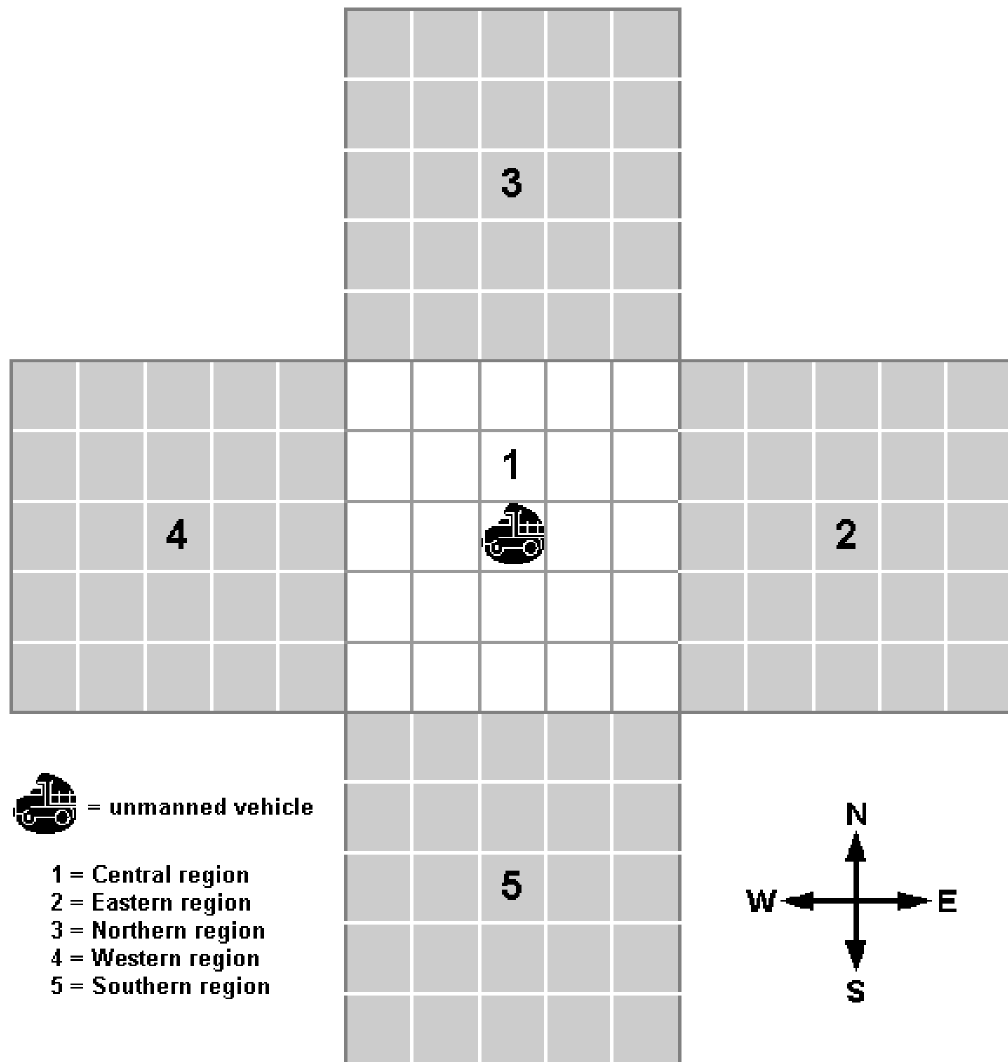


Figure 5.5: Regional Movement of Unmanned Vehicles

- *validXYMovementBetweenRegions:* This property verifies that an unmanned vehicle that starts at location (3,3) in Region 1 will successfully move into the correct (x,y) location of the boundary region when the appropriate conditions are met.

```
G(((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.xpos = 5 & Vehicle1.direction = 2)
-> (X (Vehicle1.xpos = 1)))
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.ypos = 5 & Vehicle1.direction = 3)
-> (X (Vehicle1.ypos = 1)))
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.xpos = 1 & Vehicle1.direction = 4)
-> (X (Vehicle1.xpos = 5)))
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.ypos = 1 & Vehicle1.direction = 5)
-> (X (Vehicle1.ypos = 5))))
```

- *validXYMovementWithInARegion:* This property verifies that an unmanned vehicle that starts at location (3,3) in Region 1 will successfully move into the correct (x,y) location in situations other than boundary cases.

```
G(((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.xpos = 1 & Vehicle1.direction = 2)
-> (X (Vehicle1.xpos = 2)))
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.xpos = 1 & Vehicle1.direction = 3)
-> (X (Vehicle1.xpos = 1)))
```

```
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.xpos = 1 & Vehicle1.direction = 5)
-> (X (Vehicle1.xpos = 1)))

&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.xpos = 2 & Vehicle1.direction = 2)
-> (X (Vehicle1.xpos = 3)))
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.xpos = 2 & Vehicle1.direction = 3)
-> (X (Vehicle1.xpos = 2)))
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.xpos = 2 & Vehicle1.direction = 4)
-> (X (Vehicle1.xpos = 1)))
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.xpos = 2 & Vehicle1.direction = 5)
-> (X (Vehicle1.xpos = 2)))

&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.xpos = 3 & Vehicle1.direction = 2)
-> (X (Vehicle1.xpos = 4)))
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.xpos = 3 & Vehicle1.direction = 3)
 -> (X (Vehicle1.xpos = 3)))
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.xpos = 3 & Vehicle1.direction = 4)
-> (X (Vehicle1.xpos = 2)))
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.xpos = 3 & Vehicle1.direction = 5)
-> (X (Vehicle1.xpos = 3)))
```

```
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.xpos = 4 & Vehicle1.direction = 2)
-> (X (Vehicle1.xpos = 5)))
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.xpos = 4 & Vehicle1.direction = 3)
-> (X (Vehicle1.xpos = 4)))
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.xpos = 4 & Vehicle1.direction = 4)
-> (X (Vehicle1.xpos = 3)))
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.xpos = 4 & Vehicle1.direction = 5)
-> (X (Vehicle1.xpos = 4)))

&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.xpos = 5 & Vehicle1.direction = 3)
-> (X (Vehicle1.xpos = 5)))
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.xpos = 5 & Vehicle1.direction = 4)
-> (X (Vehicle1.xpos = 4)))
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.xpos = 5 & Vehicle1.direction = 5)
-> (X (Vehicle1.xpos = 5)))

&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.ypos = 1 & Vehicle1.direction = 2)
-> (X (Vehicle1.ypos = 1)))
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.ypos = 1 & Vehicle1.direction = 3)
```

```
-> (X (Vehicle1.ypos = 2)))
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.ypos = 1 & Vehicle1.direction = 4)
-> (X (Vehicle1.ypos = 1)))

&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.ypos = 2 & Vehicle1.direction = 2)
-> (X (Vehicle1.ypos = 2)))
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.ypos = 2 & Vehicle1.direction = 3)
-> (X (Vehicle1.ypos = 3)))
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.ypos = 2 & Vehicle1.direction = 4)
-> (X (Vehicle1.ypos = 2)))
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.ypos = 2 & Vehicle1.direction = 5)
-> (X (Vehicle1.ypos = 1)))

&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.ypos = 3 & Vehicle1.direction = 2)
-> (X (Vehicle1.ypos = 3)))
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.ypos = 3 & Vehicle1.direction = 3)
-> (X (Vehicle1.ypos = 4)))
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.ypos = 3 & Vehicle1.direction = 4)
-> (X (Vehicle1.ypos = 3)))
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.ypos = 3 & Vehicle1.direction = 5)
```

```
-> (X (Vehicle1.ypos = 2)))

&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.ypos = 4 & Vehicle1.direction = 2)
-> (X (Vehicle1.ypos = 4)))
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.ypos = 4 & Vehicle1.direction = 3)
-> (X (Vehicle1.ypos = 5)))
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.ypos = 4 & Vehicle1.direction = 4)
-> (X (Vehicle1.ypos = 4)))
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.ypos = 4 & Vehicle1.direction = 5)
-> (X (Vehicle1.ypos = 3)))

&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.ypos = 5 & Vehicle1.direction = 2)
-> (X (Vehicle1.ypos = 5)))
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.ypos = 5 & Vehicle1.direction = 4)
-> (X (Vehicle1.ypos = 5)))
&((Vehicle1.state = sendVehicleInfo & Vehicle1.currRegion = 1
& Vehicle1.ypos = 5 & Vehicle1.direction = 5)
-> (X (Vehicle1.ypos = 4))))
```

- *validVehicleInfoReceived:* This property verifies that the valid region information was eventually received for `Vehicle1` by the `regionI` instance of the `Regions` component.

```
G ((((Vehicle1.state = sendVehicleInfo) & X(Vehicle1.currRegion =1)) ->
F(regionI.invoke_receiveVehicleInfo_via_VehicleInfo.currentRegion = 1))
&(((Vehicle1.state = sendVehicleInfo) & X(Vehicle1.currRegion =2)) ->
F(regionI.invoke_receiveVehicleInfo_via_VehicleInfo.currentRegion = 2))
&(((Vehicle1.state = sendVehicleInfo) & X(Vehicle1.currRegion =3)) ->
F(regionI.invoke_receiveVehicleInfo_via_VehicleInfo.currentRegion = 3))
&(((Vehicle1.state = sendVehicleInfo) & X(Vehicle1.currRegion =4)) ->
F(regionI.invoke_receiveVehicleInfo_via_VehicleInfo.currentRegion = 4))
&(((Vehicle1.state = sendVehicleInfo) & X(Vehicle1.currRegion =5)) ->
F(regionI.invoke_receiveVehicleInfo_via_VehicleInfo.currentRegion = 5)))
```

The delivery policy used in this system is a *Level 2* policy that can be formalized as:

$$((pending_VehicleInfo_Count = 3) \wedge (pending_EnvVehicleInfo > 0)) \\ \Rightarrow (deliver_VehicleInfo')$$

All of the above properties evaluate to true indicating that vehicles are moving correctly within the model of the system and that vehicle components are passing valid vehicle information to `regionI`. Details regarding the results of these properties can be found in Section E.1.2.

Case 3: Collision Avoidance

The collision avoidance case analyzes the ability of vehicles within the UVCS to avoid occupying the same grid location in the same region at the same time.

An expression that an event never occurs under a set of specific conditions is known as a safety property [BB+01]. A demonstration that two vehicles never collide

under a situation in which their short term plans call for them to occupy the same location simultaneously is an example of a safety property.

It is obvious that any collision avoidance property would hold if vehicles are not permitted to move, providing that vehicles are not initialized to start in the same location. Therefore, the movement of vehicles is an important aspect of this model. We model vehicle movement by providing vehicles with an initial position and a short term plan of five directions. As a vehicle moves, new directions are added to the plan randomly. With the exception of the first five movements, vehicles in this model do not follow a preset path.

- *collisonAvoided:* This safety property verifies that the two vehicles moving in the same region will never crash. In this context, crash is defined as both vehicles occupying the same x and y position on the grid. Specifically, `Vehicle1` and `Vehicle2` will never both be in the same region with the same x and y position.

```
G (~(Vehicle1.currRegion = Vehicle2.currRegion)
 | ~(Vehicle1.xpos = Vehicle2.xpos)
 | ~(Vehicle1.ypos = Vehicle2.ypos))
```

This property should hold if the traffic rules are correct and if there is no delay in the delivery of events. Using a *Level 0* immediate delivery policy we verified that the safety property *collisionAvoided* is true. A formal representation of the delivery policy is:

$$(true) \Rightarrow (deliver_VehicleInfo1' \wedge deliver_VehicleInfo2')$$

Complete details of the analysis are presented in Section E.1.3.

Chapter 6

Optimization Techniques for Model Checking

In this chapter we justify the use of optimization techniques for model generation, specifically abstraction. The use of abstraction is a common technique that often yields a reduction in the size of the finite state model. Two common abstraction techniques, cone of influence reduction and data abstraction, were utilized in the generation of the models in Chapter 5. Both the cone of influence reduction, a type of correctness and failure preserving transformation, and data abstraction are performed by hand prior to the generation of the SMV model. Additionally, techniques such as other correctness and failure preserving transformations and reduction of non-determinism were also used.

The use of optimization techniques is discussed because many of the models verified in Chapter 5 would be too large to be verified without the use of abstraction. Thus, optimization is critical to model checking large software systems.

In addition to model-specific optimizations we discuss model checker tool optimizations. The majority of research related to optimization in the context of model checking focuses on the model and not on the tool. We discuss some built-in model checker dependent options of Cadence SMV used in the generation of our models. We also discuss alternatives to standard sequential model checkers such as the parallel model checkers in [GMS01] and [BLW01].

6.1 Advantages of Optimization

Optimization is defined as "An act, process, or methodology of making something as fully perfect, functional, or effective as possible" [Onl02]. When model checking large systems, optimization techniques are used to enhance performance by generating a smaller state space and a quicker verification time for properties. In many other areas optimizations mean the difference between inefficient and optimal performance. When model checking large systems the use of optimization techniques can mean the difference between verifying properties and not receiving any verification results [1].

The following sections contain examples of the advantages of certain model-specific techniques, specifically, decreases in model size and in verification time.

[1] Since the model is finite, the analysis will terminate, however in an extremely large model this process can often take a long time since the complexity of CTL model checking is $O(|S| \cdot (|S| + |R|))$ where S is the set of states and R is the total transition relation. Since model checkers such as Cadence SMV provide no data regarding the time required to verify a given property it is often the case that the person running the model checker will terminate the verification process prior to completion.

6.2 Model-Based Optimizations

When using model-based optimizations we want the resulting abstract model to correspond to the original model. More precisely, if a property holds in the abstract model, then it holds in the original model. Otherwise, analysis results may be spurious and thus useless. Similarly, if a property fails in the abstract model than it should also fail in the original model. Since some optimizations incur a loss of information, it is not always possible to achieve this tight correspondence.

6.2.1 Correctness and Failure Preserving Transformations

A correctness and failure preserving transformation leaves the truth or failure of a property φ unchanged. That is, $M \models \varphi \Leftrightarrow M' \models \varphi$. The use of these types of transformations always has to preserve the both the validity and failure of the specifications being verified. The correctness and failure preserving transformations used in the examples in Chapter 5 fall into two categories: cone of influence reduction and combinational correctness and failure preserving transformations. Cone of influence reduction is concerned with correctness and failure preserving omissions of variables. Combinational correctness and failure preserving transformations are concerned with the combination of variables into a smaller number of variables.

Cone of Influence Reduction

Cone of influence reduction involves the removal of variables that do not influence variables in the properties or specifications currently being verified. Let V be the set of variables for an implicit-invocation system. Let V_i be the set of variables in the specification. The variables in the cone of influence C are defined by three rules

[CGP99]:

1. $V_i \subseteq C$

2. Consider all $v_j \in V$. If there exists $v_i \in C$ such that v_i is dependent on v_j, then $v_j \in C$. Note that a variable v_i depends on v_j if and only if v_j directly or indirectly affects an assignment to v_i.

3. C is the minimal set of variables that satisfy rules 1 and 2.

Once C has been constructed it is used to reduce the size of the system. Let E_l be the set of variables in the left side on an assignment. If $\forall v_i \in E_l, v_i \notin C$ then the assignment is removed from the system. Otherwise, the assignment is kept as part of the reduced model.

Recall the six parameters that compose an implicit-invocation system discussed in Section 2.1.3. The removal of two of the parameters, components and events, are both ways that a model can be optimized. The removal of events and components in implicit-invocation systems is an instance of cone of influence reduction.

System elements such as events and components can be removed if they do not, directly or indirectly, affect the analysis of the property currently being verified.

In the case of events, the reduction can be substantial. This is because multiple system variables are used to model one event in an implicit-invocation system. In the case of components, the removal of a component can also greatly reduce the size of the model. Recall that the addition of components to an implicit-invocation causes exponential growth in model size.

As a demonstration of the importance of cone of influence reductions we will examine the `LookCorrectness` property in two versions of the ABLS system: `abls_Look`

and abls_Look_Find. The two versions are constructed with the following components and events included:

- abls_Look

  Components: RequestWorkstation, MasterWorkstation

  Events: EnvPoll, EnvLook, Look, LookResult

- abls_Look_Find

  Components: RequestWorkstation, MasterWorkstation

  Events: EnvPoll, EnvLook, Look, Find, LookResult, FindResult

System	Number of States Reached	Reached States BDD Size	State Count Time for User (sec.)	Number of State Count Iterations
abls_Look	2.41356e+07	4759	0.390562	22
abls_Look_Find	9.94850e+09	183408	73.84	30

Table 6.1: State Count Results of LookCorrectness property

System	Analysis Result	BDD Nodes Allocated	Verification Time for User (sec.)	Number of Verification Iterations
abls_Look	True	296553	11.7269	472
abls_Look_Find	True	87550456	99841.8	2073

Table 6.2: Analysis Results of LookCorrectness property

The two versions are identical except that the `abls_Look_Find` version contains two events that are outside of the cone of influence. Events `Find` and `FindResult` do not influence the validity of the property. Table 6.1 shows the difference in model size and Table 6.2 shows the difference in verification times and nodes allocated during the analysis of the property. Recall that the result for the `LookCorrectness` property is true. Note that for the `LookCorrectness` property the verification time increases from the `abls_Look` system to the `abls_Look_Find` system by more than three orders of magnitude (Table 6.2). We conclude that analysis is often impossible or impractical without the use of optimization.

Combinational Correctness and Failure Preserving Transformations

Recall the six parameters that compose an implicit-invocation system discussed in Section 2.1.3. The removal and combination of two of the parameters, components and events, are both ways that a model can be optimized. Note that these system-level optimizations are event dependent and can not always be utilized.

Combinational correctness and failure preserving transformations can be applied to both the event and component parameters of an implicit-invocation system.

Identical instances of a component can be transformed into one component provided that no events are sent from one component instance to another. In the case that event communication occurs between instances of the same component, the instances can still possibly be combined provided that the events in question are not part of the current analysis and that these events do not affect the delivery of events in any properties being analyzed.

Multiple events announced by a given component can be transformed into one

event if the events are announced simultaneously and delivery of one does not affect the delivery of the others, for example, in the case of synchronous immediate delivery. If a priority queue delivery is used in which one event has a different priority then the other, the combination may lead to spurious analysis results.

An example of component transformation that preserves correctness occurs in the ABLS system. In this system the Sensor Workstations were combined into one component to reduce the size of the model. An additional example of this technique occurs with the combination of the Vehicle components and the Region components in the UVCS system.

6.2.2 Data Abstraction

In addition to correctness and failure preserving transformations that involve the removal and combination of variables within an implicit-invocation system, the data within the system can be abstracted to provide smaller models and quicker verification times.

Data abstraction relies on the observation that there usually exists a simple relationship among data values within system specifications that involve data paths. Data abstraction is a process in which a mapping is found between actual data values and a smaller abstract set of data values [CGP99]. The system with the abstract data values, in place of the actual data values, is then modeled.

An example of data abstraction is the reduction of the range of variables. Consider the look example results outlined in Table 6.3 for various numbers of locations. In this context, locations refer to the number of different locations in the system and more specifically, the number of sensor workstations. Note that the size of the BDD

for the reached states of the location range [-1..10] is abnormally high. Even though the number of reached states for this case grows at a consistent rate, the BDD does not. This anomaly could be the result of a memory leak or a bug in the Cadence SMV model checker. Currently, the exact cause of this anomaly is not known.

Location Range	**Number of States Reached**	**Reached States BDD Size**	**State Count Time for User (sec.)**	**Number of State Count Iterations**
-1..0	386683	2174	0.37	21
-1..1	2.41356e+07	4759	0.60	22
-1..2	3.81536e+07	5546	0.63	22
-1..3	3.08658e+08	6564	0.81	22
-1..4	1.66806e+09	7692	0.93	22
-1..5	6.87490e+10	8972	1.175	22
-1..10	9.29408e+12	212488	87.25	22
-1..15	2.16993e+14	32703	6.67	22
-1..20	2.22387e+15	48304	10.99	22

Table 6.3: State Count Results of property LookCorrectness for look_ABLS system with variations in the number of locations

6.2.3 Reduction of Non-Determinism

In the original system proposed in [GK00] all data required initialization including local variables and input and output correspondences for global variables. This requirement did not extend to event data since events contained no data except their name. In the current form the DTD allows for the use of event data without initialization. This is because restricting event data can be overly obtrusive and provide unwanted constraints on events since event data initialization may not be an attribute of the system being modeled. An initialization of event data means that every component that announces that event has to have the same initial data values. This

contradicts the loose coupling that should exist between implicit-invocation components. However, in systems where event data initialization does not compromise the system integrity, it can be useful for reducing the model size. In Cadence SMV, the failure to initialize a variable means that it can take on all possible values [McM99a]. Representing all possible values will increase the set of initial states. Thus the utilization of event data initialization causes a reduction in the nondeterministic selection of an initial state from the set of all possible states.

6.3 Tool-Based Optimizations

6.3.1 Built-In Model Checker Parameters

The Cadence SMV model checker contains several built in features that provide a more efficient analysis. These features are controlled as parameters in the command-line version of the model checker. The following are features of Cadence SMV that were used to optimize the analysis of the implicit-invocation systems presented in Chapter 5:

- `-f`: Use forward search to compute the reachable states and then restrict the model checking to only these states. This option is useful in cases where the state-space is sparse, that is, cases where the reachable states are only a small portion of the entire state-space.

- `-h`: Use built-in heuristics to order the variables in the Binary Decision Diagram(BDD). The size of the BDD depends on variable ordering [Rud93].

In most cases, the use of "`-f`" and the consequent reduction from the entire state space to only the reachable states provides obvious benefits in terms of model size and analysis time.

The variable ordering heuristics provided by "`-h`" improve the model analysis efficiency by reducing the size of the BDD. There are other variable ordering alternatives to using the "`-h`" heuristics. The "`-sift`" option is another variable ordering technique which attempts to minimize the BDD size by using sifting. Sifting is a process that reorders all of the existing nodes by sifting the variables up and down to find the optimum positions. The total BDD node size for each permutation generated is recorded and the permutation that minimizes the BDD size is selected [Rud93]. An additional alternative for variable ordering in Cadence SMV is to use "`-i`". This option allows the order of BDD variables to be done by hand via specification in the variable order file. In general, global variables should appear at the beginning of the order and variables that appear close together in formulas should appear close together in the variable order.

The use of the state reduction technique "`-f`" and the variable ordering heuristics provided by "`-h`" are beneficial in the modeling of the implicit-invocation systems from Chapter 5 and thus were included in the discussion of model checker optimizations.

6.3.2 Parallel Model Checking

Parallel model checking was not used as an optimization technique in any of the examples from Chapter 5. Its inclusion in this chapter is because of the advantages parallel model checking could provide to the modeling of large implicit-invocation

systems. Based on the results obtained from the ABLS and UVCS it is apparent that sequential model checking using current optimization techniques is not enough to model larger "real-world" systems. Parallel model checking may provide an alternative for modeling these larger systems which does not reduce the model size, but instead reduces the verification time. This is accomplished by splitting the model checking tasks over more than one processor.

The area of parallel model checking focuses on two primary topics: parallel model generation and parallel state space search. Parallel model generation or parallel state space construction can be accomplished by having each processor in a parallel system be responsible for generating part of the state space. Recall from Section 2.2.2 that the state-space in the context of model checking is a Kripke structure or a Labeled Transition System (LTS) [Din00]. After the generation is complete the parts of the LTS are then merged. [GMS01] discuss parallel state space construction for model checking using enumerative verification. Enumerative verification actually enumerates all reachable states of the state space while symbolic verification uses techniques such as BDDs to encode the state space. Recall that we use symbolic verification not enumerative in the model checking of implicit-invocation systems.

The parallelization of model checking can be taken a step forward by not only using parallel state space construction but also search the constructed state space in parallel. A description of this parallelization is out of the scope of this thesis. One version is presented in detail in [BLW01].

Chapter 7

Summary & Conclusions

7.1 Thesis Summary

Model checking is a three step process: modeling, specification, and verification. Traditionally, the verification step is the only step that is automatic while modeling and specification are done manually.

Model checking is used primarily in hardware analysis and is restricted to limited use in software. The two primary reasons for the limited use of model checking with software systems is the difficulty of the modeling step and the large models that are often produced. The partial automation of the modeling step simplifies the complex task of converting software artifacts to modeling artifacts. Additionally, spurious results attributed to artifact conversion are reduced. Thus, a more automated approach to modeling provides increased usability for performing model checking on software systems. The semi-automated approach used in this thesis was summarized in Figure 3.3.

The task of automating the generation of models for all types of software systems is

too broad an issue to tackle. Our limitation of only considering systems that conform to the implicit-invocation style provides insight into the utilization of architectural styles in the automation of generation of models.

Implicit-invocation systems consist of 6 parameters: components, shared variables, events, event-method bindings, an event delivery policy, and a concurrency model. These parameters were discussed in detail in Section 2.1.3. The work presented in this thesis is an extension of work conducted at Carnegie Mellon University by Garlan and Khersonsky [GK00]. We extend the approach in [GK00] by focusing on the events parameter and the event delivery policy parameter. Indirectly, this work caused alterations to the modeling of components as well.

The events parameter was extended to allow for the inclusion of data. This is an important extension since the majority of implicit-invocation system utilize events containing data. The inclusion of data into events resulted in major alterations to the model of the event dispatcher and the component event queues.

The delivery policy parameter was also extended from the binary representation of policies in [GK00]. The `"Immediate"` and `"Random"` policies, used by Garlan and Khersonsky, were incorporated as delivery rules. Propositional logic was used to describe delivery policies that can contain more complex conditions than was possible in [GK00].

In addition to modifying the parameter-based automation of [GK00] we also provided extensive evaluation of our modified technique. Only limited evaluation was done using the original technique. Specifically, a set-counter example was modeled. Our evaluation included a comparison of the set-counter example modeled using the original technique and our modified technique. The results of this evaluation were that

both techniques provided models that produced equivalent analysis output when subjected to the verification step of model checking. We then modeled the Active Badge Location System (ABLS) and the Unmanned Vehicle Control System (UVCS). Both the ABLS and the UVCS systems are significantly larger than the set-counter example. For both of them, partial systems were modeled and the properties of each model were analyzed. The analysis of the semi-automatically generated models for the ABLS and UVCS systems demonstrated the increased usability of automating the modeling step of the model checking process.

During the evaluation of our approach it became clear that in order to model significant implicit-invocation systems optimization would have to be used. If optimization techniques were not used many of our models would have been too large to verify in a reasonable amount of time. By using optimization techniques we were able to avoid any serious problems as a result of the state explosion problem [CGP99]. For the systems discussed in our evaluation, we used standard optimization techniques such as cone of influence reduction, data abstraction, and reduction of non-determinism. In addition to standard techniques, we also considered architectural style specific optimizations for implicit-invocation systems. An example of an architectural style specific optimization is combinational correctness and failure preserving transformations.

7.2 Conclusions

Even with optimization, accurate models of "real-world" systems are often extremely large. The size of these models requires state-of-the-art computers and a large quantity of patience. While model checking the ABLS and UVCS systems in Chapter 5, it

became obvious that it is not feasible to model some larger systems even when exhaustively using model optimizations. Although some larger system are not feasible to model check in their entirety, it is clear that the modeling of partial "real-world" systems is possible. Our model checking approach, in the context of implicit-invocation systems, is viable since the loose coupling of components provides natural partitions for developing partial systems. Although our approach is only explored in the context of implicit-invocation systems, we are confident that similar approaches could be developed to model check other architectural styles.

We feel it is important to consider our model checking technique in comparison to other software quality assurance techniques that are commonly used in industry. We conclude that although model checking is a promising method of analysis, it is not yet ready to replace techniques such as testing and inspection in mainstream industry development. However, model checking does have some distinct advantages over currently used software quality assurance techniques.

The advantage of using model checking over testing is that model checking provides a guarantee about the behaviour of a given software system. On the one hand, since in testing it is rarely feasible to test all possible test cases, it is impossible to guarantee that the software works correctly. On the other hand, if a system is modeled correctly, we can provide such a guarantee for given properties that are verified using model checking. The current advantage of testing over model checking is that model checking is still limited by the state-explosion problem and the generation of models is often much more difficult than generating a set of test cases. These are both problems we have made contributions towards alleviating.

Model checking has a similar advantage over inspection since inspection does not

provide a guarantee about the behaviour of a given software system. An advantage of inspection over model checking is that inspection finds defects in the code and model checking finds problems in the model. Once a problem is found in a finite state model, the error in the source code that caused the problem still has to be located. This is also an advantage of inspection over testing, where incorrect program behaviors, known as bugs, have to be traced back to the code defects that cause them.

In conclusion, our approach is viable for most significant implicit-invocation system, especially when partial models are considered. In the context of implicit-invocation, we have been able to make contributions to alleviating problems in model generation through partial automation. Additionally, we have tried to reduce the size of "real-world" models through the use of optimization. However, comparing our model checking approach, in the context of implicit-invocation systems, to accepted techniques such as testing and inspection, it is clear that additional research still needs to be done before model checking is readily usable outside of hardware analysis and the analysis of safety-critical software systems.

7.3 Future Work

Based on observations made during this research, there are four primary topics for future work in the area of model checking:

1. Optimization techniques
2. Complete automation of model generation
3. Alternative intermediate representation

4. Extension of our technique to other architectural styles

7.3.1 Optimization Techniques

Optimization of the Model

We have taken an extensive look at model-based optimizations in Chapter 6. However, additional work on optimization techniques for implicit-invocation models is still required. Some specific optimization topics that require future work include:

1. *Program slicing:* the benefits of program slicing are discussed in Section 7.3.2. Slicing involves finding the relevant parts of a program that could affect values computed at some specified point in the program. Work needs to be done to apply slicing to a program using a temporal logic specification as the slicing criterion. By identifying and removing parts of a program that are irrelevant, or do not influence the validity of a specification, we reduce the size of the finite state model generated.

2. *Data abstraction:* this optimization technique was discussed in Chapter 6. However, more aggressive use of data abstraction is possible and needs to be explored.

3. *Tool-specific optimizations:* tool-specific optimization techniques need to be explored.

4. *Compositional approaches:* in these approaches we analyze parts of a system in isolation. In the context of compositional approaches, we need to better understand how to partition implicit-invocation systems and determine if we can

make improvements to these approaches by better leveraging the loose coupling of implicit-invocation components.

Optimization of the Model Checker

The majority of work in the area of model checking concentrates on the model and not the tool. Additional work has to focus on developing better and more efficient model checkers that utilize technologies such as parallel computation. Research into model checker algorithms and tools will increase the likelihood of model checking being more readily accepted as a mainstream software analysis technique.

7.3.2 Complete Automation of Model Generation

The complete automation of finite state model generation, from source code, is a direct extension of this thesis in the context of the implicit-invocation architectural style. This research presents a semi-automated approach to taking software artifacts of an implicit-invocation system as input and generating a finite state machine as output. The process is only a partial solution and is not completely automatic since user interaction is required in developing the intermediate representation. Future work needs to focus on bridging the gap between source code and the intermediate representation. Until this gap is bridged, the possibility of spurious results, although reduced, is still possible since the intermediate representation, currently written in XML, is generated by hand. Figure 7.1 shows the current research in the context of an overall approach to fully automating the generation of a finite state machine. It also highlights various other parts of the process which require additional work. Areas such as program slicing and language transformation are well researched. However,

more research is needed to determine how such techniques can be used in the specific application of generating finite state models from source code.

Previous work on automatic model generation has focused on generating models for all types of software. For example, Bandera [CDH+00] is a collection of integrated program transformation and analysis components. Bandera currently inputs Java source code and produces a finite state machine as output. The main contrast between work conducted by the Bandera group and our work is that the Bandera group is gaining insight by looking at a larger class of programs. Our focus is on a smaller scale and is more concerned with providing efficient models that are optimized in the context of a specific architectural style. The broad focus of Bandera reduces optimization since architectural specific techniques are not considered.

7.3.3 Alternative Intermediate Representation

An additional area of research is the exploration into appropriate intermediate representations. Intermediate representations are important for indirection. Without an intermediate representation it would be difficult to support multiple forms of software artifact inputs and produce different finite state model outputs. Thus, we do not directly transform source code into a finite state model.

The current XML representation is adequate as an intermediate representation but may not be the best choice. This is especially clear from the current technique in which the XML is generated by hand. Non-automated generation of XML is extremely tedious since XML is very verbose. As a result of the verbose nature of XML, the intermediate representation is lengthy and difficult to comprehend. A first step in looking at alternative intermediate representations would be to find a representation

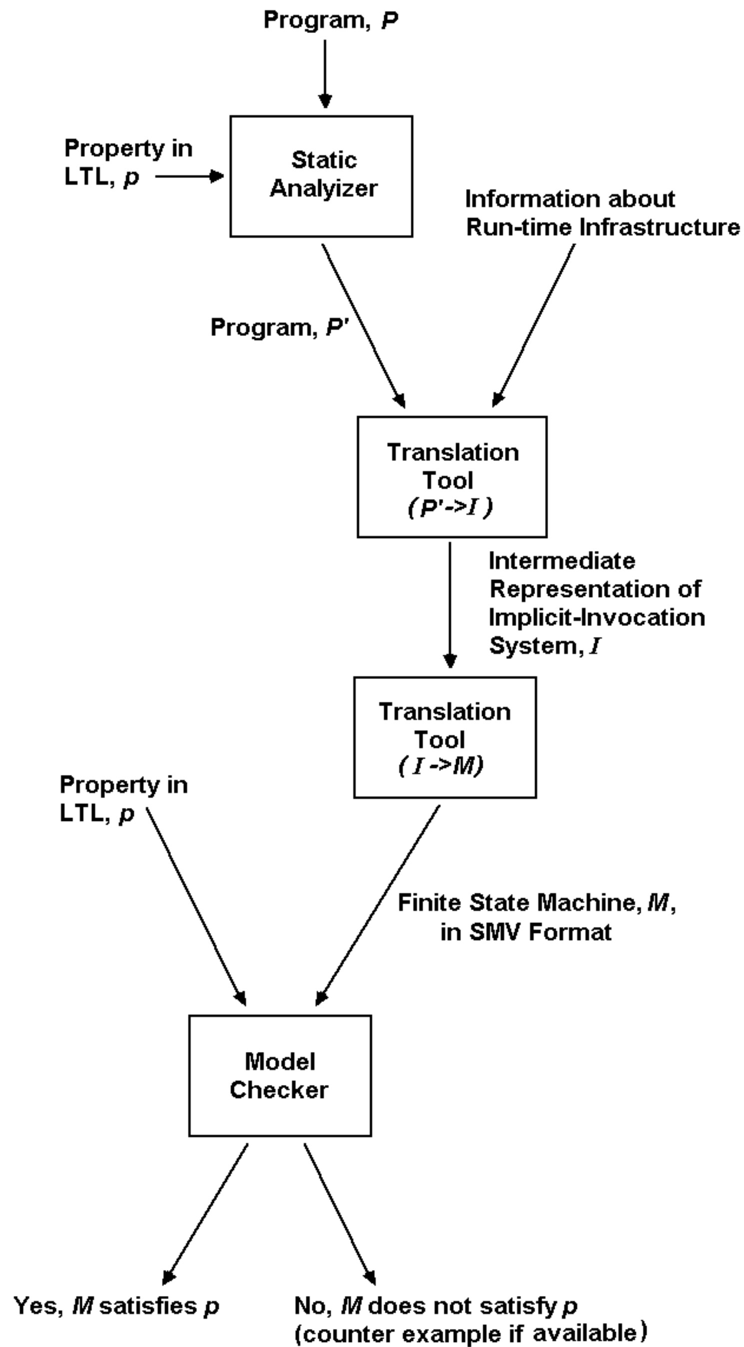


Figure 7.1: Fully Automated Model Checking Approach

that provides the same semantic abilities as XML without the extensive use of tags.

7.3.4 Extension of Technique to Other Architectural Styles

Additional future work could involve extending ideas presented here into a more general framework for automatic model generation of additional software systems that utilize other architectural styles. Architecture styles provide insight into how to model a particular group of software systems and can be used to build tools to generate finite state machines from source code. The knowledge gained from generating models of implicit-invocation systems could be extended to generate models for pipe-and-filter systems, layer system or even stand-alone programs.

Conversely, tools such as Bandera [CDH+00] could be improved from both a model generation and an optimization perspective if information regarding architectural styles of software systems was taken into account.

Bibliography

[AS85] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, pages 181–185, Oct. 1985.

[BB+01] Beatrice Berard, Michel Bidoit, et al. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer, 2001.

[Bel97] Bell Labs. *Basic Spin Manual*, Aug. 1997.

[BLW01] Benedikt Bollig, Martin Leucker, and Michael Weber. Local parallel model checking for the alternation free μ-calculus. Technical report, RWTH Aachen, Mar. 2001.

[CDH+00] James Corbett, Matthew Dwyer, John Hatcliff, et al. Bandera: Extracting finite-state models from Java source code. In *Proceedings 2000 International Conference on Software Engineering*, June 2000.

[CE81] E. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop, volume 131 of Lecture Notes in Computer Science*. Springer-Verlag, May 1981.

[CGL92] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Proceedings of Principles of Programming Languages*, 1992.

[CGP99] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, 1999.

[CR98] Alessandro Cimatti and Marco Roveri. *NuSMV 1.1 User Manual*. Instituto per la Ricerca Scientifica e Tecnologica (IRST), an institute of Istituto Trentino di Cultura, 1998.

[CRW01] 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, Aug. 2001.

[CW96] Edmund Clarke and Jeannette Wing. Formal methods: State of the art and future directions. Technical report, Working Group on Formal Methods for ACM Workshop on Strategic Directions in Computer Research, Dec. 1996.

[DAC99] Matthew Dwyer, George Avrunin, and James Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the* 21st *International Conference on Software Engineering*, pages 411–420, May 1999.

[DGJN98a] Juergen Dingel, David Garlan, S. Jha, and David Notkin. Reasoning about implicit invocation. In *Proceedings of the Sixth International Symposium on the Foundations of Software Engineering*, Nov. 1998.

[DGJN98b] Juergen Dingel, David Garlan, S. Jha, and David Notkin. Towards a formal treatment of implicit invocation using rely/guarantee reasoning. *Formal Aspects of Computing*, pages 193–213, 1998.

[Dil94] Antoni Diller. *Z: An Introduction to Formal Methods.* John Wiley and Sons, 1994.

[Din00] Juergen Dingel. CISC 835 class notes, Department of Computing and Information Science, Queen's University, 2000.

[Fly02] Peter Flynn. The XML FAQ, Jan. 2002. Originally maintained on behalf of the World Wide Web Consortium's XML Special Interest Group.

[Ger90] Colin Gerety. A new generation of software development tools. *Hewlett-Packard Journal*, pages 48–58, Jun. 1990.

[GK00] David Garlan and Serge Khersonsky. Model checking implicit-invocation systems. Carnegie Mellon University, 2000.

[GKN92] David Garlan, Gail Kaiser, and David Notkin. Using tool abstraction to compose systems. *IEEE Software*, pages 30–38, 1992.

[GMS01] Hubert Garavel, Radu Mateescu, and Irina Smarandache. Parallel state space construction for model-checking. In *Proceedings of* 8th *International SPIN Workshop on Model Checking of Software*, May 2001.

[GN91] David Garlan and David Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *VDM '91: Formal Software Development Methods*, pages 31–44, Oct. 1991.

[GS96] David Garlan and Mary Shaw. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice-Hall, 1996.

[HP99] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. Technical report, NASA Ames Research Center, Mar. 1999.

[Inc99] Sun Microsystems Inc. *Jini ^{TM} Architectural Overview: Technical White Paper*, Jan. 1999.

[Inc00] Sun Microsystems Inc. *JavaSpaces ^{TM} Service Specification*, Oct. 2000.

[Jac] Daniel Jackson. Automatic analysis of architectural style. School of Computer Science, Carnegie Mellon University.

[McM92] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, May 1992.

[McM99a] K. L. McMillan. *Getting Started with SMV*. Cadence Berkeley Labs, Mar. 1999.

[McM99b] K. L. McMillan. *The SMV Language*. Cadence Berkeley Labs, Mar. 1999.

[NC00] Gleb Naumovich and Lori Clarke. Classifying properties: an alternative to the safety-liveness classification. In *Proceedings of the* 8th *ACM Sigsoft Symposium on the Foundations of Software Engineering*, pages 159–168, Mar. 2000.

[Onl02] Meriam Webster Online. Collegiate dictionary, 2002.

[PFL00] J. Pereira, F. Fabret, and F. Llirbat. Le subscribe: Publish and subscribe on the web at extreme speed. In *VLDB'2000*, 2000. Demo paper.

[QS81] J. P. Quielle and J. Sifakis. Specification and verification of concurrent system in cesar. In *Proceedings of the* 5th *International Symposium in Programming*, 1981.

[Rei90] Steven P. Reiss. Connecting tools using message passing in the field environment. *IEEE Software*, pages 57–66, Jul. 1990.

[Rud93] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design*, pages 42–47, Nov. 1993.

[SA97] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings AUUG97*, Sept. 1997.

[SBC+98] R. Strom, G. Banavar, T. Chandra, et al. Gryphon: An information flow based approach to message brokering. In *International Symposium on Software Reliability Engineering*, 1998.

[Sha95] Mary Shaw. Architectural issues in software reuse: It's not just the functionality, it's the packaging. In *Proceedings IEEE Symposium on Software Reusability*, Apr. 1995.

[SN90] K. Sullivan and David Notkin. Reconciling environment integration and software evolution. In *Proceedings of SIGSOFT '90: Fourth Symposium on Software Development Environments*, Dec. 1990.

[SvK98] Sylvia Stuurman and Jan van Katwijk. On-line change mechanisms the software architectural level. In *SIGSOFT '98 Proceedings*, pages 80–86, 1998.

[WHFG92] Roy Want, Andy Hopper, Veronica Falcao, and Jon Gibbons. The active badge location system. *ACM Transactions on Information Systems*, pages 91–102, Jan. 1992. Olivetti Research Ltd.

[WVF96] Jeannette Wing and Mandana Vaziri-Farahani. A case study in model checking software systems. *Science of Computer Programming*, Apr. 1996.

Appendix A

Glossary

Below is a glossary of terminology associated with software engineering and more specifically formal methods and implicit-invocation. Common acronyms used throughout this thesis are also included.

ABLS

Active Badge Location System. An electronic tagging system developed by Olivetti Research Ltd. (ORL).

Architectural Style

"...a vocabulary of components and connector types, and a set of constraints on how they can be combined" [GS96]. In other words, an architectural style constrains both the topology and the behavior of an architecture [Jac].

BDD

Binary Decision Diagram. A directed acyclic graph (DAG). The graph is rooted and is composed of terminal and nonterminal vertices [CGP99]. In the context

of the model checking algorithm, "A BDD is a particular data structure which is very commonly used for the symbolic representation of state sets" [BB+01].

CTL

Computation Tree Logic. Allows for quantification of paths from a current state as well as quantification over the states in a selected path.

DTD

Document Type Definition. A formal description of a particular document type that includes the names of different types of elements, where the elements may occur, and how all of the elements fit together. [Fly02].

Finite State Model

See *Labeled Transition System (LTS).*

Formal Methods

"mathematically-based languages, techniques, and tools for specifying and verifying systems" [CW96].

Implicit-Invocation

A component-based architectural style in which components "publish" events and "subscribe" to other events. The components are decoupled since delivery of events is handled by an event dispatcher or a central message server.

Logic

"A science that deals with the principles and criteria of validity of inference and demonstration: the science of the formal principles of reasoning" [Onl02].

LTL

Linear Temporal Logic. Allows for quantification over states but not over paths. Since LTL offers no quantification over paths, all formulas are implicitly universally quantified over all paths originating from the initial state.

LTS

Labeled Transition System. Also referred to as Kripke structures or generalized as the state-space. An LTS is a four tuple $M = (S, S_0, R, L)$ where

1. S is the finite set of states in the system
2. S_0 is the set of initial states
3. $R \subseteq S \times S$ is a total transition relation that defines all transitions between states in S. The relation is total because for every s in S, there exists a t in S such that $R(s, t)$ where $R \subseteq S \times S$.
4. $L : S \rightarrow 2^{AP}$ is a labeling function $\forall s \in S$. Each state is labeled with the atomic propositions (AP) that are true in that state. Specifically, for every p in AP and s in S, we have p in $L(s)$ if and only if p is true in s.

Model Checking

Relies on building a finite state model of a system and then checking that a desired property holds in the model. The check is performed as an exhaustive state space search, which is guaranteed to terminate since the model is finite [CW96].

Publish-Subscribe

See *Implicit-invocation.*

Reachable States

The states in a Labeled Transition System that are reachable from one of the initial states.

SMV

Symbolic Model Verifier. A tool for the verification and simulation of an SMV model using symbolic model checking techniques.

Temporal Logic

A variant of modal logic in which formulae are interpreted over not just one state but a sequence of states.

Theorem Proving

Logic contains a description of syntactically well-formed formulas and rules of inference. Deduction is the process of deriving a new formula from existing formulas using the rules of inference. A theorem prover implements the logic and its rules, that is, it checks syntactic correctness.A property is verified if a proof can be derived from the set of assumptions using specified inference rules [CW96]. Most theorem provers offer an automatic proof search mode in which the tool attempts to find a derivation for some desired formula automatically. Logics as expressive as predicate logic are undecidable which means that no theorem

prover will ever be able to prove all theorems fully automatically. Thus, user interaction will always be required.

UVCS

Unmanned Vehicle Control System. A control system for unmanned vehicles designed specifically for use at the Maasvlakte port system in Northwestern Europe.

XML

Extensible Markup Language. A metalanguage that is used to create specific markup languages for describing a particular document type [Fly02]. XML is not a fixed format predefined markup language.

Appendix B

XML Document Type Definition (DTD)

A Document Type Definition (DTD) is a formal description of a particular document type that includes the names of different types of elements, where the elements may occur, and how all of the elements fit together [Fly02]. We use the following DTD to specify the content and structure of elements with in our XML input representation of an implicit-invocation system. The DTD contains a main element called `event-system` which represents an entire implicit-invocation system including the components and the run-time infrastructure. Structurally, the `event-system` element can contain other elements such as a `delivery-policy` element. Below we outline how all of the elements including `event-system` and `delivery-policy` are structured and how they are connected.

```
<?xml version='1.0' encoding='UTF-8'?>

<!--
```

```
  Event system consists of
  (1) a set of one more events announced by the environment
  (2) a set of zero or more internal events
  (3) one event delivery policy style
  (4) a set of one or more component descriptions
  (5) a set of zero or more of global data variables
  (6) a set of one or more component instantiations
  (7) a set of one or more event bindings
  (8) a set of zero or more properties to be verified

  Event system has two attributes:
  (1) name
  (2) event queue size, the maximum number of pending events of each type
      The default is 3
-->

<!ELEMENT event-system (env-event+,
                        event*,
delivery-policy+,
component+,
global-data*,
component-instance+,
event-binding+,
property*)>
<!ATTLIST event-system
    name               CDATA    \#REQUIRED
    event-queue-size   CDATA    \#IMPLIED>

<!--
  An event announced by environment has the following attributes
  (1) name
  (2) [Optional] event semantics property
  (3) flag indicating whether this event is always announced (default: true)
  (4) flag indicating whether this event eventually stopped being announced
      (default: true)
-->

<!ELEMENT env-event EMPTY>
<!ATTLIST env-event
    name               CDATA    \#REQUIRED
    semantics          CDATA    \#IMPLIED
    always-announced   CDATA    \#IMPLIED
    stops-eventually   CDATA    \#IMPLIED>

<!--
  A component-announced event has these attributes:
  (1) name
  (2) [Optional] event semantics property
```

```
  The body of an event contains zero or more event-data elements
-->

<!ELEMENT event (event-data*)>
<!ATTLIST event
    name              CDATA    \#REQUIRED
    semantics         CDATA    \#IMPLIED>

<!--
  An event-data element is defined as:
-->

<!ELEMENT event-data EMPTY>
<!ATTLIST event-data
    name              CDATA    \#REQUIRED
    type              CDATA    \#REQUIRED>

<!--
  An event delivery policy has an event name as attribute [Optional].
  If this attribute is omitted then the delivery policy will apply to
  all events. The body of the delivery policy consists of a
  policy-statement
-->

<!ELEMENT delivery-policy (policy-statement)>
<!ATTLIST delivery-policy
event-name CDATA \#IMPLIED>

<!--
  policy-statement is a complex element:
-->

<!ELEMENT policy-statement
(composed-policy-statement  |
 policy-if-statement |
 delivery-statement |
   no-delivery-statement)>

<!--
  composed policy statement:
-->

<!ELEMENT composed-policy-statement (policy-statement+)>

<!--
  if-statement has a condition as an attribute. It has a policy-statement
  has a truth statement and has an optional false statement.
-->
```

```
<!ELEMENT policy-if-statement (policy-statement, policy-statement?)>
<!ATTLIST policy-if-statement
condition CDATA \#REQUIRED>

<!--
  A delivery statement has the event name to be delivered in its body and
  has the delivery rule specified as an attribute [Optional]. The default
  delivery rule is immediate delivery
-->

<!ELEMENT delivery-statement (\#PCDATA)>
<!ATTLIST delivery-statement
rule CDATA \#IMPLIED>

<!--
  No delivery statement:
-->

<!ELEMENT no-delivery-statement EMPTY>

<!--
  Global data is identified by name and type
-->

<!ELEMENT global-data EMPTY>
<!ATTLIST global-data
    name              CDATA    \#REQUIRED
    type              CDATA    \#REQUIRED>

<!--
   Component specification includes:
   (1) a set of zero or more events announced by the component
   (2) a set of zero or more inputs (shared global data for reading)
   (3) a set of zero or more outputs (shared global data for writing)
   (4) a set of zero or more local variables
   (5) a set of one or more methods

   Component has the following attributes:
   (1) name
   (42 method queue size, the maximum number of pending invocations for every
       method (an invocation is delayed if another method in the same component
       is running). The default is 3
-->

<!ELEMENT component (event-announced*,
                     input*,
     output*,
```

```
                     local-var*,
     restriction?,
                     method+)>
<!ATTLIST component
    name               CDATA    \#REQUIRED
    method-queue-size  CDATA    \#IMPLIED>

<!--
  Event announced is a simple element that has one name attribute
-->

<!ELEMENT event-announced EMPTY>
<!ATTLIST event-announced
    name              CDATA \#REQUIRED>

<!--
  Input data is identified by name and type
-->

<!ELEMENT input EMPTY>
<!ATTLIST input
    name              CDATA \#REQUIRED
    type              CDATA \#REQUIRED>

<!--
  Output data is identified by name, type, and init value
-->

<!ELEMENT output (\#PCDATA)>
<!ATTLIST output
    name              CDATA \#REQUIRED
    type              CDATA \#REQUIRED>

<!--
  Local data is identified by name, type, and init value
-->

<!ELEMENT local-var (\#PCDATA)>
<!ATTLIST local-var
    name              CDATA \#REQUIRED
    type              CDATA \#REQUIRED>

<!--
  Restriction is a boolean value that is assumed to be true at all times
-->

<!ELEMENT restriction (\#PCDATA)>
```

```
<!--
  Method has a name and a statement to be executed when this method is called
-->

<!ELEMENT method (statement)>
<!ATTLIST method
    name              CDATA \#REQUIRED>

<!--
  statement is a complex element:
-->

<!ELEMENT statement
    (composed-statement |
     if-statement       |
     assignment         |
     announcement)>

<!--
   Composed statement consists of one or more other statements
-->

<!ELEMENT composed-statement (statement+)>

<!--
  An if statement has a condition, and a true-condition statement,
  and optionally a false-condition statement
-->

<!ELEMENT if-statement (statement, statement?)>
<!ATTLIST if-statement
    condition         CDATA \#REQUIRED>


<!--
  Assignment statement assigns a value to a component variable
  The variable name is an attribute, and the value is in the body
-->

<!ELEMENT assignment (\#PCDATA)>
<!ATTLIST assignment
    var-name          CDATA  \#REQUIRED>

<!--
  An announcement statement announces an event specified in the element body
-->

<!ELEMENT announcement (\#PCDATA)>
```

```
<!--
  Component instance specifies component name and instance name.
  It contains correspondence for inputs and outputs
-->

<!ELEMENT component-instance (input-correspondence*, output-correspondence*)>
<!ATTLIST component-instance
    component-name    CDATA   \#REQUIRED
    instance-name     CDATA   \#REQUIRED>

<!--
   Input and output correspondences are very similar; they establish
   association between global data and component inputs/outputs
-->

<!ELEMENT input-correspondence EMPTY>
<!ATTLIST input-correspondence
    input-name        CDATA   \#REQUIRED
    global-data-name  CDATA   \#REQUIRED>

<!ELEMENT output-correspondence EMPTY>
<!ATTLIST output-correspondence
    global-data-name  CDATA   \#REQUIRED
    output-name       CDATA   \#REQUIRED>

<!--
   Event binding specifies which methods in which components are to be
   invoked on a particular event
-->

<!ELEMENT event-binding (method-binding+)>
<!ATTLIST event-binding
    event-name        CDATA   \#REQUIRED>

<!--
   Method binding identifies a component instance and a method in it
-->

<!ELEMENT method-binding EMPTY>
<!ATTLIST method-binding
    instance-name     CDATA   \#REQUIRED
    method-name       CDATA   \#REQUIRED>

<!--
  A property is an element included into the output model
-->
```

```
<!ELEMENT property (\#PCDATA)>
<!ATTLIST property
   name                CDATA  \#IMPLIED>
```

Appendix C

Set and Counter Example

C.1 Summary of Cadence SMV Results

Each model was model checked a Sun Fire Midframe 4800 Server with the following specifications:

- *Processor architecture:* SPARC V9
- *Number of processors:* 4
- *Processor speed:* 750MHz
- *Main Memory:* 8 GB

For each model, a table of results is provided for:

1. *State count results:* provides details regarding the model produced for the specified property in the specified system. The number of states reached and the BDD size of the reached states is recorded to provide information on the size of

the model. The user time required to generate the model as well as the number of iterations provide the time required for generating the model.

2. *Analysis results:* provides details on the verification of the specified property in the specified model. The first column of data in this table gives the result of the verification: "true" or "false". The number of BDD nodes allocated is provided as a measure of memory and size constraints. The user time required as well as the number of verification iterations are also provided.

3. *Counter-example results:* not all properties will have information displayed in this table. All properties verified to be "true" and all properties verified as "false" but have no counter-example will appear in the table as "-". For properties with counter examples the user time and the number of iterations needed to generate the example are given.

C.1.1 gk_SC_immediate SMV Results

Property	**Number of States Reached**	**Reached States BDD Size**	**State Count Time for User (sec.)**	**Number of State Count Iterations**
AlwaysCatchesUp	8218	638	0.22	13
SomethingInterestingHappens	8218	676	0.18	13
GlobalDataCheck	-	616	0.11	1

Table C.1: State Count Results of gk_SC_immediate

Property	Analysis Result	BDD Nodes Allocated	Verification Time for User (sec.)	Number of Verification Iterations
AlwaysCatchesUp	True	93543	3.52	577
SomethingInterestingHappens	True	138025	5.15	374
GlobalDataCheck	True	13249	0.14	1

Table C.2: Analysis Results of gk_SC_immediate

C.1.2 gk_SC_random SMV Results

Property	Number of States Reached	Reached States BDD Size	State Count Time for User(sec.)	Number of State Count Iterations
AlwaysCatchesUp	1.8967e+06	1409	1.22	26
SomethingInterestingHappens	1.8967e+06	2054	2.01	26
GlobalDataCheck	-	589	0.09	1

Table C.3: State Count Results of gk_SC_random

Property	**Analysis Result**	**BDD Nodes Allocated**	**Verification Time for User (sec.)**	**Number of Verification Iterations**
AlwaysCatchesUp	False	6241704	1725.38	1259
SomethingInterestingHappens	True	322070	15.69	417
GlobalDataCheck	True	10191	0.14	1

Table C.4: Analysis Results of gk_SC_random

Property	**Counter-Example?**	**Counter-Example Time for User (sec.)**	**Number of Counter-Example Iterations**
AlwaysCatchesUp	Yes	311.49	240
SomethingInterestingHappens	N/A	-	-
GlobalDataCheck	N/A	-	-

Table C.5: Counter-Example Results of gk_SC_random

C.1.3 mod_SC_1_immediate SMV Results

Property	**Number of States Reached**	**Reached States BDD Size**	**State Count Time for User (sec.)**	**Number of State Count Iterations**
AlwaysCatchesUp	413540	1258	0.25	14
SomethingInterestingHappens	413540	1357	0.25	14
GlobalDataCheck	-	740	0.09	1

Table C.6: State Count Results of mod_SC_1_immediate

Property	**Analysis Result**	**BDD Nodes Allocated**	**Verification Time for User (sec.)**	**Number of Verification Iterations**
AlwaysCatchesUp	True	182768	6.49	482
SomethingInterestingHappens	True	225715	24.22	462
GlobalDataCheck	True	12439	0.16	1

Table C.7: Analysis Results of mod_SC_1_immediate

C.1.4 mod_SC_1_random SMV Results

Property	Number of States Reached	Reached States BDD Size	State Count Time for User (sec.)	Number of State Count Iterations
AlwaysCatchesUp	3.73396e+06	2735	2.38	26
SomethingInterestingHappens	3.73396e+06	4146	4.16	26
GlobalDataCheck	-	590	0.07	1

Table C.8: State Count Results of mod_SC_1_random

Property	Analysis Result	BDD Nodes Allocated	Verification Time for User (sec.)	Number of Verification Iterations
AlwaysCatchesUp	False	10249747	3306.71	1525
SomethingInterestingHappens	True	528145	44.13	511
GlobalDataCheck	True	10067	0.14	1

Table C.9: Analysis Results of mod_SC_1_random

Property	**Counter-Example?**	**Counter-Example Time for User (sec.)**	**Number of Counter-Example Iterations**
AlwaysCatchesUp	Yes	562.29	260
SomethingInterestingHappens	N/A	-	-
GlobalDataCheck	N/A	-	-

Table C.10: Counter-Example Results of mod_SC_1_random

C.1.5 mod_SC_2 SMV Results

Property	**Number of States Reached**	**Reached States BDD Size**	**State Count Time for User (sec.)**	**Number of State Count Iterations**
AlwaysCatchesUp	1.08721e+09	4937	3.1, 0.08	22
SomethingInterestingHappens	1.08721e+09	5891	2.4, 0.04	22
GlobalDataCheck	-	711	0.09, 0.04	1(?)

Table C.11: State Count Results of mod_SC_2

Property	Analysis Result	BDD Nodes Allocated	Verification Time for User (sec.)	Number of Verification Iterations
AlwaysCatchesUp	False	4620582	1895.58	836
SomethingInterestingHappens	True	397565	34.49	566
GlobalDataCheck	True	10907	0.14	1

Table C.12: Analysis Results of mod_SC_2

Property	Counter-Example?	Counter-Example Time for User (sec.)	Number of Counter-Example Iterations
AlwaysCatchesUp	Yes	176.42	95
SomethingInterestingHappens	N/A	-	-
GlobalDataCheck	N/A	-	-

Table C.13: Counter-Example Results of mod_SC_2

Appendix D

Active Badge Location System

D.1 Summary of Cadence SMV Results

Each model was model checked on a Sun Fire Midframe 4800 Server with the following specifications:

- *Processor architecture:* SPARC V9
- *Number of processors:* 4
- *Processor speed:* 750MHz
- *Main Memory:* 8 GB

For each model, a table of results is provided for:

1. *State count results:* provides details regarding the model produced for the specified property in the specified system. The number of states reached and the BDD size of the reached states is recorded to provide information on the size of

the model. The user time required to generate the model as well as the number of iterations provide the time required for generating the model.

2. *Analysis results:* provides details on the verification of the specified property in the specified model. The first column of data in this table gives the result of the verification: "true" or "false". The number of BDD nodes allocated is provided as a measure of memory and size constraints. The user time required as well as the number of verification iterations are also provided.

3. *Counter-example results:* not all properties will have information displayed in this table. All properties verified to be "true" and all properties verified as "false" but have no counter-example will appear in the table as "-". For properties with counter examples the user time and the number of iterations needed to generate the example are given.

D.1.1 poll_ABLS SMV Results

Property	**Number of States Reached**	**Reached States BDD Size**	**State-Count Time for User (sec.)**	**Number of State-Count Iterations**
PollEventAlwaysAnnounced	532491	9384	1.08	20
PollEventAlwaysEventually-Announced	532657	9383	0.851224	20
Person2CanBeLocated	1.75104e+07	19727	5.50	20
OnceLocatedPerson2AlwaysLocated	1.75104e+07	19727	5.48	20

Table D.1: State Count Results of poll_ABLS

Property	**Analysis Result**	**BDD Nodes Allocated**	**Verification Time for User (sec.)**	**Number of Verification Iterations**
PollEventAlwaysAnnounced	False	4454750	415.568	2071
PollEventAlwaysEventually-Announced	True	927599	25.056	297
Person2CanBeLocated	False	6950592	3224.49	2127
OnceLocatedPerson2AlwaysLocated	False	12853802	7091.17	2727

Table D.2: Analysis Results of poll_ABLS

Property	**Counter-Example?**	**Counter-Example Time for User (sec.)**	**Number of Counter-Example Iterations**
PollEventAlwaysAnnounced	Yes	10.945	103
PollEventAlwaysEventually-Announced	N/A	-	-
Person2CanBeLocated	Yes	83.831	103
OnceLocatedPerson2AlwaysLocated	Yes	151.92	103

Table D.3: Counter-Example Results of poll_ABLS

D.1.2 find_ABLS_immediate

Property	**Number of States Reached**	**Reached States BDD Size**	**State Count Time for User (sec.)**	**Number of State Count Iterations**
FindCorrectnessImmediately	64739	2325	0.140202	20
FindCorrectnessInNextState	64739	2325	0.170245	20
FindCorrectnessInTwoStates	64739	2325	0.190274	20
FindCorrectnessInThreeStates	64739	2325	0.230331	20

Table D.4: State Count Results of find_ABLS_immediate

Property	**Analysis Result**	**BDD Nodes Allocated**	**Verification Time for User (sec.)**	**Number of Verification Iterations**
FindCorrectnessImmediately	False	473201	13.8499	491
FindCorrectnessInNextState	True	420350	18.7169	518
FindCorrectnessInTwoStates	False	910188	31.3551	546
FindCorrectnessInThreeStates	False	1549930	41.8001	572

Table D.5: Analysis Results of find_ABLS_immediate

Property	Counter-Example?	Counter-Example Time for User (sec.)	Number of Counter-Example Iterations
FindCorrectnessImmediately	Yes	4.4164	149
FindCorrectnessInNextState	N/A	-	-
FindCorrectnessInTwoStates	Yes	8.2318	149
FindCorrectnessInThreeStates	Yes	12.488	154

Table D.6: Counter-Example Results of find_ABLS_immediate

D.1.3 find_ABLS_random

Property	Number of States Reached	Reached States BDD Size	State Count Time for User (sec.)	Number of State CountIterations
FindCorrectnessImmediately	107370	2303	0.34049	19
FindCorrectnessInNextState	107370	2303	0.480691	19
FindCorrectnessInTwoStates	107370	2303	0.350504	19
FindCorrectnessInThreeStates	107370	2303	0.330475	19

Table D.7: State Count Results of find_ABLS_random

Property	Analysis Result	BDD Nodes Allocated	Verification Time for User (sec.)	Number of Verification Iterations
FindCorrectnessImmediately	False	1129526	19.8686	399
FindCorrectnessInNextState	False	1410473	32.3365	427
FindCorrectnessInTwoStates	False	3253647	91.7419	449
FindCorrectnessInThreeStates	False	6416698	167.861	483

Table D.8: Analysis Results of find_ABLS_random

Property	Counter-Example?	Counter-Example Time for User (sec.)	Number of Counter-Example Iterations
FindCorrectnessImmediately	Yes	15.6925	193
FindCorrectnessInNextState	Yes	19.9687	160
FindCorrectnessInTwoStates	Yes	53.3871	175
FindCorrectnessInThreeStates	Yes	99.023	194

Table D.9: Counter-Example Results of find_ABLS_random

D.1.4 history_ABLS_queue_1

There are two types of component announced events in the history_ABLS_queue_1 system: `HistoryResult` events and `History` events. The priority queue delivery policy gives high priority to `HistoryResult` events and low priority to `History` events. Recall, that environment events such as `EnvHistory` are not affected by the queue and always use immediate delivery.

Property	**Number of States Reached**	**Reached States BDD Size**	**State Count User Time (sec.)**	**Number of State Count Iterations**
HistoryCorrectnessImmediately	535587	6534	0.811166	19
HistoryCorrectnessInNextState	535587	6534	0.791139	19
HistoryCorrectnessInTwoStates	535587	6534	0.791138	19
HistoryCorrectnessInThreeStates	535587	6534	0.881267	19

Table D.10: State Count Results of history_ABLS_queue_1

Property	**Analysis Result**	**BDD Nodes Allocated**	**Verification Time for User (sec.)**	**Number of Verification Iterations**
HistoryCorrectnessImmediately	False	107519	32.2263	456
HistoryCorrectnessInNextState	True	397315	33.7686	516
HistoryCorrectnessInTwoStates	False	1278179	39.5669	575
HistoryCorrectnessInThreeStates	False	1372085	49.8517	636

Table D.11: Analysis Results of history_ABLS_queue_1

Property	**Counter-Example?**	**Counter-Example Time for User (sec.)**	**Number of Counter-Example Iterations**
HistoryCorrectnessImmediately	Yes	17.1046	152
HistoryCorrectnessInNextState	N/A	-	-
HistoryCorrectnessInTwoStates	Yes	19.9387	152
HistoryCorrectnessInThreeStates	Yes	20.3392	152

Table D.12: Counter-Example Results of history_ABLS_queue_1

D.1.5 history_ABLS_queue_2

There are two types of component announced events in the history_ABLS_queue_1 system: `HistoryResult` events and `History` events. The priority queue delivery policy gives high priority to `History` events and low priority to `HistoryResult` events. Recall, that environment events such as `EnvHistory` are not affected by the queue and always use immediate delivery.

Property	**Number of States Reached**	**Reached States BDD Size**	**State Count Time for User (sec.)**	**Number of State Count Iterations**
HistoryCorrectnessImmediately	600146	7851	1.0515	21
HistoryCorrectnessInNextState	600146	7851	0.981411	21
HistoryCorrectnessInTwoStates	600146	7851	1.0415	21
HistoryCorrectnessInThreeStates	600146	7851	1.00144	21

Table D.13: State Count Results of history_ABLS_queue_2

Property	Analysis Result	BDD Nodes Allocated	Verification Time for User (sec.)	Number of Verification Iterations
HistoryCorrectnessImmediately	False	1064096	45.6957	578
HistoryCorrectnessInNextState	False	1076887	47.5584	650
HistoryCorrectnessInTwoStates	False	1181674	71.4928	724
HistoryCorrectnessInThreeStates	False	1386455	70.7618	796

Table D.14: Analysis Results of history_ABLS_queue_2

Property	Counter-Example?	Counter-Example Time for User (sec.)	Number of Counter-Example Iterations
HistoryCorrectnessImmediately	Yes	14.8413	159
HistoryCorrectnessInNextState	Yes	13.249	152
HistoryCorrectnessInTwoStates	Yes	16.5538	159
HistoryCorrectnessInThreeStates	Yes	18.2161	159

Table D.15: Counter-Example Results of history_ABLS_queue_2

D.1.6 look_ABLS

Property	Number of States Reached	Reached States BDD Size	State Count Time for User (sec.)	Number of State Count Iterations
LookReachability	14268	2660	0.420603	21
LookLiveness	14268	2660	0.410590	21
LookEventuallyAlwaysAnnounced	14268	2660	0.380547	21
LookCorrectness	2.41356e+07	4759	0.390562	22

Table D.16: State Count Results of look_ABLS

Property	Analysis Result	BDD Nodes Allocated	Verification Time for User (sec.)	Number of Verification Iterations
LookReachability	False	375544	13.2390	566
LookLiveness	False	283240	1.83264	120
LookEventuallyAlwaysAnnounced	False	424068	29.8629	1051
LookCorrectness	True	296553	11.7269	472

Table D.17: Analysis Results of look_ABLS

Property	Counter-Example?	Counter-Example Time for User (sec.)	Number of Counter-Example Iterations
LookReachability	Yes	0.1002	57
LookLiveness	Yes	0.17024	57
LookEventuallyAlwaysAnnounced	Yes	3.8956	85
LookCorrectness	N/A	-	-

Table D.18: Counter-Example Results of look_ABLS

D.1.7 with_ABLS

Property	Number of States Reached	Reached States BDD Size	State Count Time for User (sec.)	Number of State Count Iterations
WithReachability	15293	3536	0.460662	21
WithLiveness	15293	3536	0.510734	21
WithEventuallyAlwaysAnnounced	15293	3536	0.430619	21
WithCorrectness	3.13687e+06	3835	0.370533	21

Table D.19: State Count Results of with_ABLS

Property	Analysis Result	BDD Nodes Allocated	Verification Time for User (sec.)	Number of Verification Iterations
WithReachability	False	312616	15.6725	566
WithLiveness	False	222538	2.14308	120
WithEventuallyAlwaysAnnounced	False	467734	34.3594	1051
WithCorrectness	True	242522	14.4608	471

Table D.20: Analysis Results of with_ABLS

Property	Counter-Example?	Counter-Example Time for User (sec.)	Number of Counter-Example Iterations
WithReachability	Yes	0.1803	25
WithLiveness	Yes	0.06009	25
WithEventuallyAlwaysAnnounced	Yes	0.0701	25
WithCorrectness	N/A	-	-

Table D.21: Counter-Example Results of with_ABLS

D.1.8 find_look_ABLS

Property	Number of States Reached	Reached States BDD Size	State Count Time for User (sec.)	Number of State Count Iterations
FindLookCorrectness	3.00022e+06	18975	6.5394	11

Table D.22: State Count Results of find_look_ABLS

Property	Analysis Result	BDD Nodes Allocated	Verification Time for User (sec.)	Number of Verification Iterations
FindLookCorrectness	True	2991643	251.972	400

Table D.23: Analysis Results of find_look_ABLS

Appendix E

Unmanned Vehicle Control System Example

E.1 Summary of Cadence SMV Results

Each model was model checked on a Sun Fire Midframe 4800 Server with the following specifications:

- *Processor architecture:* SPARC V9
- *Number of processors:* 4
- *Processor speed:* 750MHz
- *Main Memory:* 8 GB

For each model, a table of results is provided for:

1. *State count results:* provides details regarding the model produced for the specified property in the specified system. The number of states reached and the

BDD size of the reached states is recorded to provide information on the size of the model. The user time required to generate the model as well as the number of iterations provide the time required for generating the model.

2. *Analysis results:* provides details on the verification of the specified property in the specified model. The first column of data in this table gives the result of the verification: "true" or "false". The number of BDD nodes allocated is provided as a measure of memory and size constraints. The user time required as well as the number of verification iterations are also provided.

3. *Counter-example results:* not all properties will have information displayed in this table. All properties verified to be "true" and all properties verified as "false" but have no counter-example will appear in the table as "-". For properties with counter examples the user time and the number of iterations needed to generate the example are given.

E.1.1 UVCS SMV Results: Rule Version Synchronization

Property	**Number of States Reached**	**Reached States BDD Size**	**State Count Time for User (sec.)**	**Number of State Count Iterations**
ruleVersionConsistent	5.46601e+021	20048	1.12161	15
ruleVersionCurrentand- ConsistentAlways	5.46601e+021	21289	1.1116	

Table E.1: State Count Results of Rule Version Synchronization Properties

Property	Analysis Result	BDD Nodes Allocated	Verification Time for User (sec.)	Number of Verification Iterations
ruleVersionConsistent	True	383055	2.82406	273
ruleVersionCurrentand- ConsistentAlways	False	601626	3.7554	273

Table E.2: Analysis Results of Rule Version Synchronization Properties

Property	Counter-Example?	Counter-Example Time for User (sec.)	Number of Counter-Example Iterations
ruleVersionConsistent	N/A	-	-
ruleVersionCurrentand- ConsistentAlways	Yes	1.56225	96

Table E.3: Counter-Example Results of Rule Version Synchronization Properties

E.1.2 UVCS SMV Results: Regional Vehicle Transfer

Property	Number of States Reached	Reached States BDD Size	State Count Time for User (sec.)	Number of State Count Iterations
regionalMovement	6073	245	0.14	17
validXYMovementBetweenRegions	6073	212	0.14	17
validXYMovementWithInARegion	6073	212	0.14	17
validVehicleInfoReceived	66027	2304	0.38	26

Table E.4: State Count Results of Regional Vehicle Transfer Properties

Property	Analysis Result	BDD Nodes Allocated	Verification Time for User (sec.)	Number of Verification Iterations
regionalMovement	True	16808	0.24	73
validXYMovementBetweenRegions	True	17120	0.23	73
validXYMovementWithInARegion	True	13820	0.24	73
validVehicleInfoReceived	True	268532	4.96	276

Table E.5: Analysis Results of Regional Vehicle Transfer Properties

E.1.3 UVCS SMV Results: Collison Avoidance

Property	Number of States Reached	Reached States BDD Size	State Count Time for User (sec.)	Number of State Count Iterations
collisionAvoided	2.69211e+24	4943	19	0.84121

Table E.6: State Count Results of Collison Avoidance Properties

Property	Analysis Result	BDD Nodes Allocated	Verification Time for User (sec.)	Number of Verification Iterations
collisionAvoided	True	241746	1.94279	151

Table E.7: Analysis Results of Collison Avoidance Properties

Appendix F

Variations on Event Delivery Policies

This appendix contains example delivery policies that supplement the material discussed in Chapter 4. Each event delivery policy is explained with the use of events that contain no event data. This is done for simplicity, and in an effort to place the focus on the delivery policy, not on the events being delivered.

Delivery Policy Categorization:	*Level 1*		
Event Types:	x	y	z
Event Priorities:	$p(x) = 1$	$p(y) = 2$	$p(z) = 3$
Number of Events Pending:	$n(x) = 4$	$n(y) = 2$	$n(z) = 2$

Description:

A mutually exclusive priority queue. Consider a system with the above mentioned event types. The delivery policy for such a system uses a priority queue based on event type where no two events have equivalent priorities. The event delivery policy assumes that at most one event will be delivered in each state in the SMV model.

Propositional Logic Representation:

$((pending_x > 0) \Rightarrow deliver_x' \wedge \neg deliver_y' \wedge \neg deliver_z')),$
$((pending_y > 0) \Rightarrow (\neg deliver_x' \wedge deliver_y' \wedge \neg deliver_z')),$
$((pending_z > 0) \Rightarrow (\neg deliver_x' \wedge \neg deliver_y' \wedge deliver_z'))$

SMV Notation:

```
case {
 pending_x > 0:  deliver_x := 1; deliver_y := 0; deliver_z := 0;
 pending_y > 0:  deliver_x := 0; deliver_y := 1; deliver_z := 0;
 pending_z > 0:  deliver_x := 0; deliver_y := 0; deliver_z := 1;
 default:  deliver_x := 0; deliver_y := 0; deliver_z := 0;
}
```

Sample Output:

If no other events are announced, the order of delivery of events should be:

$$x, x, x, x, y, y, z, z$$

If an x is announced in the middle of the y's delivery then order should be:

$$x, x, x, x, y, x, y, z, z$$

Table F.1: A Mutually Exclusive Priority Queue Based Delivery Policy

Delivery Policy Categorization:	*Level 2*		
Event Types:	x	y	z
Event Priorities:	$p(x) = 1$	$p(y) = 2$	$p(z) = 3$
Description:			
Environment event dependent priority queue. Consider a system with the above mentioned event types and number of events pending. In the delivery policy if an `envImmediate` event is not pending then priority queue based on event type with random delay is used. Otherwise, the event dispatcher will use immediate delivery for all events. Therefore there are two delivery sub-policies for this system: • A priority-based policy and an immediate delivery policy. The priority queue based on event type with random delay is used when `envImmediate` is not announced. In this policy only one event is announced per state. • An immediate delivery is used in a state where `envImmediate` is announced. In this policy multiple events can be delivered in a single state.			
Propositional Logic Representation:			
$((pending_envImmediate > 0)$ $\Rightarrow deliver_x' \wedge deliver_y' \wedge deliver_z')),$ $((\neg(pending_envImmediate > 0) \wedge (pending_x > 0))$ $\Rightarrow ((deliver_x' \vee \neg deliver_x') \wedge \neg deliver_y' \wedge \neg deliver_z')),$ $((\neg(pending_envImmediate > 0) \wedge (pending_y > 0))$ $\Rightarrow (\neg deliver_x' \wedge (deliver_y' \vee \neg deliver_y') \wedge \neg deliver_z')),$ $((\neg(pending_envImmediate > 0) \wedge (pending_z > 0))$ $\Rightarrow ((\neg deliver_x') \wedge \neg deliver_y' \wedge (deliver_z' \vee \neg deliver_z'))$			

Table F.2: (Part A) An Environment Event Dependent Delivery Policy

SMV Notation:

```
if (pending_envImmediate > 0)
 case {
 pending_x > 0:  deliver_x := 1;
 default:  deliver_x :=0;
 };
 case {
 pending_y > 0:  deliver_y := 1;
 default:  deliver_y :=0;
 };
 case {
 pending_z > 0:  deliver_z := 1;
 default:  deliver_z :=0;
 };
 next(pending_envImmediate) := 0;
}
else {
 case {
 pending_x > 0:  deliver_x := {0,1}; deliver_y := 0; deliver_z := 0;
 pending_y > 0:  deliver_x := 0; deliver_y := {0,1}; deliver_z := 0;
 pending_z > 0:  deliver_x := 0; deliver_y := 0; deliver_z := {0,1};
 default:  deliver_x := 0; deliver_y := 0; deliver_z := 0;
 }
 next(pending_envImmediate) := 0;
}
```

Table F.3: (Part B) An Environment Event Dependent Delivery Policy

Vita

Name	Jeremy Scott Bradbury
Place and year of birth	Grand Falls, Newfoundland, Canada, 1978
Education	Queen's University, 2000-2002
	Mount Allison University, 1996-2000
Experience	Teaching assistant, Department of Computing and Information Science, Queen's University, 2000-2002
	Teaching assistant, Department of Mathematics and Computer Science, Mount Allison University, 1999-2000
	Research assistant, Department of Mathematics and Computer Science, Mount Allison University, 1998-2000
Awards	NSERC PGS A, 2000
	University Leadership Certificate, 1999-2000
	Mount Allison Computer Science Departmental Award, 1999-2000
	Percy Simpson Bailey Scholarship, 1999-2000
	Dean's Honour List, 1996-2000
	NSERC USRA, Summer 1999