Title
A framework for verifying service-oriented software systems using message sequence charts

Permalink
https://escholarship.org/uc/item/02c6356j

Author
Morales-Perea, Ernesto Emiliano

Publication Date
2007

Peer reviewed|Thesis/dissertation
A Framework for Verifying Service-Oriented Software Systems Using Message Sequence Charts

A thesis submitted in partial satisfaction of the requirements for the degree Master of Science in Computer Science by Ernesto Emiliano Morales-Perea

Committee in charge:
Professor Ingolf H. Krüger, Chair
Professor Rajesh Gupta
Professor William Howden

2007
The thesis of Ernesto Emiliano Morales-Perea is approved:

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________

Chair

University of California, San Diego
2007
# TABLE OF CONTENTS

SIGNATURE PAGE ........................................................................................................... iii

TABLE OF CONTENTS...................................................................................................... iv

LIST OF FIGURES........................................................................................................... vi

LIST OF TABLES............................................................................................................... viii

ABSTRACT OF THE THESIS ............................................................................................ ix

Chapter I Introduction......................................................................................................... 1
  I.A Motivation ................................................................................................................ 1
  I.B Problem .................................................................................................................. 2
  I.C Proposed Solution .................................................................................................. 3
  I.D Benefits and Use ..................................................................................................... 4
  I.E Relation to Other Tools and Results ....................................................................... 6
  I.F Organization of Thesis .......................................................................................... 7

Chapter II Background.................................................................................................... 9
  II.A Requirement ......................................................................................................... 9
  II.B Specification ......................................................................................................... 10
  II.C Message Sequence Chart .................................................................................... 11
  II.D Composite MSC .................................................................................................. 12
  II.E Finite State Automaton ......................................................................................... 13
  II.F Büchi Automaton .................................................................................................. 14
  II.G MSCs to Automata ............................................................................................... 15
  II.H Decomposition Property ..................................................................................... 17
  II.I Services and Service-Oriented Systems ............................................................... 18
  II.J System Properties ................................................................................................. 19
  II.K Verification .......................................................................................................... 20

Chapter III MSCCheck User Scenarios ......................................................................... 23
  III.A MSCCheck Features ........................................................................................... 23
  III.B Example System: Central Locking System ........................................................ 24
    III.B.1 Phase 1: Service Design Use Cases .............................................................. 24
    III.B.2 Phase 2: Build by Service Use Cases ............................................................ 27
    III.B.3 Phase 3: Integration Verification Use Cases .................................................. 28
    III.B.4 Phase 4: Release ............................................................................................ 37

Chapter IV MSCCheck Foundations .............................................................................. 38
  IV.A Automaton Synthesis .......................................................................................... 38
    IV.A.1 Global Automaton ......................................................................................... 38
    IV.A.2 Local Automaton .......................................................................................... 39
    IV.A.3 Automaton Composition .............................................................................. 40
  IV.B Compositional Verification Algorithm ................................................................ 41
    IV.B.1 Verifying a Causal MSC Satisfies a Property MSC ...................................... 42
IV.B.2 Verifying an Implementation Satisfies a Causal MSC ........................................45
Chapter V Architecture and Implementation of MSCCheck ........................................50
V.A System Architecture and Packaging ................................................................50
V.A.1 Eclipse ........................................................................................................52
V.B Phase 1: Design Activities ..............................................................................54
V.B.1 M2Code and Implementation Issues .........................................................55
V.B.2 MSCCheck XML Input File Structure and Java Representation ..........55
V.C Verification Activity .......................................................................................59
V.C.1 Finding a Causal MSC ..............................................................................60
V.C.2 Verifying the System Using a Causal MSC ............................................60
V.D Counterexample Analysis Portion ...................................................................70
V.D.1 Defect Format ...........................................................................................70
V.D.2 MSC Rendering Library ............................................................................71
V.D.3 Viewing a Counterexample .......................................................................72
V.E Limitations ....................................................................................................73
Chapter VI Case Study: Center TRACON Automated System .........................75
VI.A Background ...............................................................................................75
VI.B System and Test Setup ..............................................................................77
VI.B.1 System Model .........................................................................................77
VI.B.2 Implementation .......................................................................................82
VI.B.3 Experiment Setup and Results ...............................................................86
Chapter VII Comparison with Related Work ....................................................89
VII.A IIL/Turing Plus and ViP ............................................................................89
VII.B ObjectCheck .............................................................................................90
VII.C JPaX and JMPaX ......................................................................................90
VII.D JAVA-Mac ................................................................................................91
VII.E ASML ..........................................................................................................92
Chapter VIII Concluding Synopsis and Future Directions ..................................94
VIII.A Future Directions .....................................................................................95
Appendix A-1 XML Representation of C Büchi Implementation Component ........97
Bibliography .........................................................................................................98
LIST OF FIGURES

Figure I-1. MSCCheck in the Feature Driven Development Process ........................................ 5
Figure II-1. Visual Representation of a MSC ............................................................................. 12
Figure II-2. Alternative MSC Construct ................................................................................. 13
Figure II-3. FSA of a Door Lock .............................................................................................. 14
Figure II-4. Büchi automaton of a Door Lock ......................................................................... 15
Figure II-5. Global Automaton Corresponding to MSC in Figure II-1 ..................................... 16
Figure II-6. Local Automaton Corresponding to Cmpnt1 in MSC in Figure II-1 ................. 16
Figure II-7. Non-causal MSC ................................................................................................... 17
Figure III-1. Intended Workflow using MSCCheck ................................................................. 23
Figure III-2. High-level MSC for the CLS ............................................................................... 25
Figure III-3. Unlocking MSC “Unlocking” ............................................................................. 25
Figure III-4. Locking MSC “Locking” ..................................................................................... 25
Figure III-5. Locking Property MSC “LockingProperty” ........................................................ 26
Figure III-6. Unlocking Property “UnlockingProperty” .......................................................... 27
Figure III-7. Implementation Automaton for C ........................................................................ 28
Figure III-8. Implementation Automaton for KF .................................................................... 28
Figure III-9. Implementation Automaton for LM ...................................................................... 28
Figure III-10. Implementation Automaton for DB ................................................................. 28
Figure III-11. Compiled MSCCheck Configuration File ......................................................... 29
Figure III-12. MSCCheck Defects View with Violations ........................................................ 30
Figure III-13. Counterexample View for “UnlockingProperty” .............................................. 32
Figure III-14. Query User Presets MSC .................................................................................. 36
Figure III-15. Unlocking MSC joined with the Query User Presets MSC ............................... 36
Figure III-16. Refined C implementation ............................................................................... 36
Figure IV-1: Global Automaton for Unlocking MSC .............................................................. 38
Figure IV-2. Local Automaton for KF Component .................................................................. 40
Figure IV-3. Local Automaton for LM Component ............................................................... 40
Figure IV-4. Local Automaton for C Component .................................................................... 40
Figure IV-5. Local Automaton for DB Component ............................................................... 40
Figure IV-6. Cross Product of LM and KF ............................................................................ 40
Figure IV-7. Verification Algorithm Process .......................................................................... 42
Figure IV-8. Property MSC ..................................................................................................... 43
Figure IV-9. Causal MSC ........................................................................................................ 43
Figure IV-10. Global Automaton of the Property ................................................................... 44
Figure IV-11. Global Automaton of Causal MSC .................................................................... 44
Figure IV-12. Global Automaton of “Unlocking” MSC with an Infinite Loop Construct ....... 49
Figure IV-13. State Space of Cross Product of C implementation and Unlocking Environment .......................................................... 49
Figure V-1. Architectural Overview of MSCCheck ................................................................. 50
Figure V-2. MVC Architecture of MSCCheck ....................................................................... 51
Figure V-3. Informal Grammar of MSCCheck’s Büchi Automaton XML Representation .......... 56
Figure V-4. Informal Grammar of MSCCheck’s MSC XML Representation ........................... 57
Figure V-5. Locking MSC’s XML Representation .......................................................... 58
Figure V-6. Logical Structure of MSCs Using Composition Design Pattern ............... 58
Figure V-7. Verification Activities ............................................................................... 59
Figure V-8. Sample Property Automaton for Locking MSC Displaying Locations and Guarded Commands ................................................................. 65
Figure V-9. Generated Specification Automaton for Locking MSC Composed with the C Component ....................................................................................... 67
Figure V-10. Updated Configuration File for MSCCheck .............................................. 68
Figure V-11. Call to Bogor to model check the BIR code ............................................. 69
Figure V-12. Implementation of the writeCounterExamples Method in the MSCCheckCounterExampleWriter Bogor Extension ......................................................... 70
Figure V-13. MSCCheck Defect for UnlockingProperty .................................................. 71
Figure V-14. MSCCheck Defect Format ..................................................................... 71
Figure V-15. Code that Calls MSCCheck’s CounterExampleEditor from the DefectView ............................................................................................ 73
Figure VI-1. Connect System MSC ............................................................................. 78
Figure VI-2. Initialize System MSC .......................................................................... 79
Figure VI-3. Reinitialize System MSC ..................................................................... 79
Figure VI-4. Update System MSC ............................................................................ 81
Figure VI-5. CTAS System MSC .............................................................................. 82
Figure VI-6. Communications Manager Implementation Automaton ........................ 84
Figure VI-7. Weather-Aware Client Implementation Automaton .............................. 85
LIST OF TABLES

Table III-1. Types of Violations Detected by MSCCheck .................................................. 31
Table III.2: Samples of Issues in question-action pairs from [WK04] Currently Supported by MSCCheck .......................................................................................................................... 35
Table VI-1. Requirements Coverage of “Interactive” Version of CTAS ............................ 76
Table VI-2. Result Times of Methods of Verification of CTAS ........................................ 87
ABSTRACT OF THE THESIS

A Framework for Verifying Service-Oriented Software Systems Using Message Sequence Charts

by

Ernesto Emiliano Morales-Perea

Master of Science in Computer Science

University of California, San Diego, 2007

Professor Ingolf H. Krüger, Chair

The Message Sequence Chart (MSC) is a widely used formalism for specifying behaviors and properties of a component-based system. While they are primarily seen as a design and validation phase artifact in the software development process, their information can play a larger role. In most cases, as the system reaches the implementation phase of development, the gap between the semantics of the implementation and the specification widens. Through component synthesis and verification based on the composition and reuse of MSC artifacts, the system’s implementation and specification defects can be captured early on in the development process; thus, reducing the high cost of uncovering a defect at a latter stage. To demonstrate this we use MSCs to verify the implementation of a system in an integrated development environment using a generic state-of-the-art model checker combined with a verification tool we developed.
Chapter I
Introduction

Today’s society relies heavily on software systems. Software systems are transparently powering and controlling many day-to-day systems ranging from sensor-networks to avionics. However, as the demand for these software systems to control, automate and solve many complex tasks increases, the risk of system failure increases. In many cases, the lives of both businesses and people are at stake.

The scale of the problem of software defects is apparent by their effect on the national economy. In a recent study commissioned by the National Institute of Standards and Technology (NIST), it was found that software bugs cost the U.S. economy about $59.5 billion annually (roughly 0.6 percent of the gross national product) [Gal02]. The reason behind the large figure lied in who was uncovering the majority of bugs in industrial strength software. The study showed that the burden of cost was forced upon the consumer due to the poor quality of the post-production software they purchased. In addition, the study showed that about a third of the national economic cost of software bugs could be eliminated if companies provided better quality assurance of their software through testing.

I.A Motivation

Specifications have long been used as contractual documents between the requirements and implementation phases of software development. Unfortunately, there is no standard notation for specifying a system and its properties. As a result, many heterogeneous, sometimes ad-hoc, specification languages are clustered together to
define the system specification. Some specification languages have been developed to address the issue such as the UML 2.0 [UML07], which attempts to cleanly separate a system’s functional and structural models into a common set of notations. However, due to the informal nature of UML, the system’s semantics are ambiguous to the reader. Moreover, the reuse of the information embedded in these models is lost during latter stages of the system’s development cycle. This removes the greater potential of unraveling defects in both the specification of the system and the system itself. To amplify the precision of discovering design defects and system inconsistencies, and eliminating the possibility of ambiguity, formal notations and methods must be used.

One such formalism is the Message Sequence Chart (MSC) [ITU96]. MSCs describe an abstract view of the interactions between components in a system. They provide a rigorous and formal notation for the sequence of these interactions similar to UML Sequence Diagram models. The result of the formal nature of this particular type of notation is the adequate information to verify, generate and validate the services of the system under development. While MSCs may have a subjective interpretation that is not universally accepted, literature exists that yields formal semantics for MSCs which form the basis for an agreed-upon notation and meaning [FK07] [Kru00].

**I.B Problem**

Maintenance and testing are accounting for roughly three quarters of the cost and time of software production [Lea00]. As one can see from the NIST software study, a large fraction of the time and cost is due to unforeseen defects that are left in the system and later discovered by the end-user. Defects are found very late in the software development cycle, thus imposing higher costs and risks.
Many of the defects in software systems are due to an implementation that deviates from its specification. The cause is usually a developer misunderstanding the lengthy documents passed on to him or her from the designer. Additionally, a defect entering the system can be caused by a faulty specification. Since the design documents are heterogeneous and usually drawn by hand or even natural language literature, the behavior and structure of the system cannot be proven correct with these documents.

As stated earlier, reuse of software artifacts is difficult if not infeasible if the specifications are ambiguous. The time and cost to rewrite properties of a system into reasonable test cases from a written specification may not match the benefit of finding software defects. Moreover, the semantics between the property specifications and the implemented system may not be consistent, and the product of incongruent testing may be false positives.

One of the central issues is the overall lack of tool support for integrating specification and component-based software development with testing and verification. Many software testing tools focus on the functionality of the individual components of a system rather than the services provided by the component interactions that compose the system. Methods such as design by contract [Mey92] are excellent at finding bugs at the component-level, but overlook defects that may occur due to incorrect system-level behaviors. In addition, many of the available tools do not fully utilize formal methods as a verification and validation approach throughout the software development process.

I.C Proposed Solution

Our goal was to capture defects early in the software development process. Through the use of MSCs as the system’s behavioral specification, we can achieve this
using well-known formal methods. The resulting contribution of our research is an Eclipse plug-in called MSCCheck. This plug-in is capable of compositionally verifying a system based on its MSC specification properties.

Our main focus is on the verification of a system at the service-level. A service is any purposeful interaction that occurs between two or more processes or components of a system. This definition of a service is sometimes referred to as a feature. Since the service-level is an abstract high-level view of the system, verification of the functionality of an individual component (unit verification) is out of the scope of our research.

The main goal of our tool is to automate the process of statically verifying a service-based system from design to implementation within an integrated environment using MSCs. This automation consists of transforming MSCs into their automata equivalents, searching their state-space for property violations by composing each component with their specification environment and generating a trace of any violation that occurs for a specific run through that state-space.

**I.D Benefits and Use**

Through MSCCheck’s verification feature, a service-oriented system’s implementation defects may be caught early on during the software development lifecycle. This results in the reduction of high-cost post-production defects. The verification activity is semi-automated allowing for ease of use. Since the design is essentially an assertion to be used to verify the implementation, the byproduct is a congruency of the language and nomenclature during both activities of the lifecycle. The direct benefits of the semi-automation of this activity in the development cycle are two fold in terms of risk. First, the semi-automation reduces the probability of defects being
introduced into the system due to human error. Second, time to develop is faster and more accurate if it supports a well-defined process.

MSCCheck is intended for use within the Feature Driven Development (FDD) Process [PF02]. FDD is a sequential, five activity process composed of planning, designing and building a software system with “client-valued functionality” known as a feature or service driving each progressive iteration of the process. The final activity of the FDD process, building by feature, can incorporate the MSCCheck tool as a subactivity that verifies that a feature is correctly implemented in a system before releasing a new build. Figure I.1 shows how MSCCheck fits into the iterative development process. After the features are defined, designed and implemented, MSCCheck can verify the newly implemented services in relation to the services and properties defined during the first three activities. If all properties of the services pass, the implementation is ready for release. If the properties fail, then the sole component at

Figure I-1. MSCCheck in the Feature Driven Development Process
fault can be refined until it passes the verification test. In some cases the definition of the property may be either incorrect or infeasible and may require some refinement, and thus feedback loops are created back to the design and definition activities. Once new features of the system are added or required in the next build, the iteration commences with MSCCheck as the gateway to a release for any new services introduced to the system.

Another benefit of MSCCheck is its consistent use of MSCs as a communication medium. Many formal verification tools today [Hol91] [KKL01] are powerful and excellent at finding defects in specifications and running systems. However, many programmers find it difficult to understand the symbolic formalisms used in these systems such as PROMELA [Hol91] and Linear Temporal Logic (LTL) [Pnu81]. MSCCheck avoids this problem by strictly using MSCs to convey a “meaning” of the services of a system. Since MSCs dictate a visually tractable layout of the interactions of a system, the reader can immediately understand the intended behavior of a system as well as any behaviors deviating from an expected behavior. MSCCheck utilizes MSCs during counterexample analysis.

In addition, MSCCheck is fully integrated in a popular integrated development environment (IDE). Other tools may be developed for or used in this IDE that can exploit the information generated by MSCCheck or can be extended to replace some of MSCCheck’s functionality.

I.E Relation to Other Tools and Results

Several tools exist today [DL04] [BG97] [XB03] [HR01] [KKL01] [BNS01] that take an implemented software system and verify it using specification properties. The
specification properties are usually in the form of a temporal logic. The implemented system is typically verified holistically such as in [DL04], abstracted to certain levels such as in [BNS01], or decomposed at an object level such as in [XB03]. In relation to these tools, MSCCheck implements a decomposition algorithm at the service-level to verify the implemented software system through usage of Message Sequence Charts as specification properties. A detailed description of these tools and their relationship to MSCCheck is given in Chapter VII.

To illustrate the effectiveness of the semi-automated tool and the efficiency of the decomposition algorithm, we ran MSCCheck on a non-trivial case study, the Center TRACON Automated System [SCS03]. We have reduced the verification time by a factor of 1050 as compared to a manual inspection of the system, and we were able to find defects that escaped during manual inspection.

**I.F Organization of Thesis**

This thesis describes the design and implementation of the MSCCheck plug-in and demonstrates its practical use in a case study. First, we introduce background information that applies directly to MSCCheck in Chapter II. Second, MSCCheck’s features are defined along with end-user scenarios of the tool’s usage for a sample system taken from the automotive domain in Chapter III. The theoretical foundation of MSCCheck’s functionality is explained in Chapter IV. Chapter V explains the software architecture and implementation of the tool, including descriptions of all supporting tools and input and output formats. To demonstrate the effectiveness and performance of the MSCCheck tool, we analyzed the tool using requirements from a case study taken from the avionics domain in Chapter VI. Chapter VII discusses other projects and research
related to the tool and its algorithms. We conclude with final remarks and suggestions for further research and extensions that are applicable to the tool in Chapter VIII.
Chapter II
Background

This chapter presents an overview of the background material pertinent to MSCCheck and its algorithms. MSCCheck uses message sequence charts as a specification to capture the requirements of a service-oriented software system. This specification is in the form of properties that are used to perform a verification of the entire system using a compositional algorithm. The compositional algorithm used by MSCCheck requires translating MSCs into finite state machines, specifically Büchi automata. This algorithm is restricted to one type of MSC. The type of MSCs that accepted by the algorithm must cover the decomposition property. These terms are all introduced formally, along with our interpretations, in the following sections of this chapter.

II.A Requirement

The IEEE defines a requirement as: “(1) A condition or capability needed by a user to solve a problem or achieve an objective. (2) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents. (3) A documented representation of a condition or capability as in (1) or (2) [IEE07].” Requirements can range from the exceptions to the timing of a system. They explicitly define what the system is expected to do, and if used formally, can verify that the system satisfies that expectation.

In the case of the MSCCheck tool, we consider the behaviors of a software system as requirements that must be fulfilled. The behaviors of a system are defined in terms of
services that must occur inside the system and are represented by a series of message-oriented interactions of components.

As an example, a requirement of a lighting system could be “The light shall turn off when the user moves the light switch to the ‘off’ position.” This requirement would be represented as two interactions with the system that included both physically turning off the light and the user moving the light switch. Once captured, it is a requirement of the system to satisfy the dependence of these interactions.

II.B Specification

A specification, as defined by the IEEE, is “a document that specifies, in a complete, precise, verifiable manner, the requirements, design, behavior, or other characteristics of a system or component, and, often, the procedures for determining whether these provisions have been satisfied [IEE07].” In its standard form, a software specification is a document that outlines the requirements of a system and is often referred to as a software requirements specification. The most common use of a specification is as a medium between requirements elicitation phase and implementation or design phase. Other uses of specifications, such as those that conform to some formalism, include automated generation of code and test cases.

Specifications can be separated into two categories: formal and informal. The majority of software specifications in practice are informal. Informal specifications may even take the form of a natural language document. An example of an informal specification would be a Unified Modeling Language (UML) use-case diagram where ambiguity is often seen since it is at a higher level of design. Formal specifications, on the other hand, usually have a mathematical foundation which eliminates the possibilities
of ambiguities and misinterpretations. The power of formal specifications lies in their ability to be used as software analysis artifacts. Some are executable and can provide an abstract simulation of the final system. Examples of formal specifications are statecharts [KGS99], Linear Temporal Logic and xUML [MB02].

The manufacture of components from specifications is not a novel idea. The rapid generation of components from the information in specifications can be seen in such software design paradigms as Generative Programming [CE00] and implemented in process frameworks such as the Rational Unified Process [RUP03].

MSCCheck utilizes the Message Sequence Chart as its formal software requirements specification. The Message Sequence Chart is described in the next section.

II.C Message Sequence Chart

The Message Sequence Chart (MSC) is a popular, formal specification used to document the interactions occurring in a concurrent system [RGG96] [ITU96]. MSC’s main application has been in the documentation of the behavior of telecommunication switching systems.

MSCs are developed during the design phase of the software development lifecycle. They capture the required behavior between a set components or processes in a system. These interactions describe proper behavior that the system is required to exhibit. A single MSC describes the partial-ordering of a system scenario.
A typical MSC contains a set of vertical axes corresponding to the components of the system. These vertical axes are labeled with the name of the corresponding component and are referred to as instances. To indicate a communication relationship between components a horizontal arrow is drawn from the sending instance to the consuming instance. These arrows are known as messages and are labeled with the intended text message. A message sent from an instance not defined in the MSC is known as a message from the environment (Message0 in Figure II-1). The vertical ordering of all messages in an MSC implies the order for that particular system behavior.

For our research, we use MSCs as the requirements specification of the system. An MSC specification defines the required behaviors of the system that must verify the behavior of the implemented system.

As an example consider the MSC in Figure II-1. The ordering of the sends and receipts of messages in this interaction must be followed. Message1 must be sent immediately following the receipt of Message0, Message5 must eventually be sent following the receipt of Message1, and etc. are all requirements that are embedded inside the semantics of this MSC.

**II.D Composite MSC**
The MSCs described thus far are defined as basic MSCs (BMSC) and fall under a larger set of MSCs known as Composite MSCs (C-MSCs). These MSCs extend the BMSC language by adding functionality such as alternatives, joins and loops. C-MSCs are essentially cascading constructs that wrap around basic MSCs.

Alternative MSCs are choices of interaction flow. The construct label found on the diagram is “ALT.” Based on a choice, the flow of interaction between components would either be the top or bottom half of the construct in the diagram (Figure II-2). The join operator is constructed with the label “JOIN.” The interactions on the top and bottom half of the construct are joined/connected on common messages. The messages following the common messages are interleaved up until the next common message. A loop on a message sequence chart is noted by the “Loop <*” label and contains a full construct that is not partitioned into an upper and lower half. The loop construct signifies repetitive interactions between components. The interested reader can find a formal translation of C-MSCs to their respective automaton in [FK07].

![Figure II-2. Alternative MSC Construct](image)

**II.E Finite State Automaton**

Finite State Automata (FSA) model the behavior of a system using states and transitions between those states. Each state of a FSA corresponds to a unique stage or
point of the system that is arrived by past behaviors that have occurred in the system. In order to proceed to a new or previous FSA state, an event must occur that directly changes the system state. To model proper execution of the system, accept states are defined. If a finite sequence of events occurs that is explicitly defined in the FSA transitional structure and the final event drives the FSA into an accept state, then the sequence of events is considered as successfully completing for the given system.

![Figure II-3. FSA of a Door Lock](image)

Figure II-3 gives an example of a FSA describing a door system that can be in two states, *Locked* and *Unlocked*. The initial state is represented by the blackened circle and the system begins in the *Unlocked* state. In order for the system to move to a new state, the system must go through a “Lock the door” event where it can then persist in the *Locked* state. For this example, if *Unlocked* were considered an acceptance state, then a proper execution of the door system must always terminate in the *Unlocked* position. Proper execution of the door system would include a transitional sequence such as “Lock the Door”, “Unlock the Door”, “Lock the Door”, “Unlock the Door.” An improper execution of the system would be to “Lock the Door” and termination of the system in the *Locked* state.

### II.F Büchi Automaton

A Büchi automaton is syntactically similar to a FSA. Semantically a Büchi automaton differs from a FSA in its acceptance conditions. While a FSA accepts finite runs of a system, a Büchi automaton can accept infinite executions of a system. A Büchi
automaton must have a set of acceptance states. For proper acceptance of the modeled system, a system’s accept state must be visited infinitely often. For our purposes, since there are no “end” states in a Büchi automaton, finite runs of the system are considered accepted if the execution path terminates in an accept state.

Consider an enhanced version of the Door system’s automaton representation in Figure II-4. If the automaton’s accept state is “Unlocked” then an infinite execution of the system is recognizable if it continuously visits the “Unlocked” state. This includes runs where the “Unlock the Door” transition is infinitely executed in the “Unlocked” state.

II.G MSCs to Automata

Current research has shown that the feasibility of MSCs in the automation of software analysis is dependent on the synthesis of its state-based counterpart. As a result, a number of methods have been introduced to translate MSCs into finite-state systems [FK07] [DH99] [MKS00] [AEY00] [KGS99].
MSCCheck’s implementation input is in the format of automata created from the synthesis algorithm in [FK07]. This algorithm is described in more detail in Chapter IV. The algorithm follows by translating each message into two transitions between three states in the resulting automaton. Sent messages are denoted by a “!” while received messages are prefixed with “?”. Figure II-5 displays the global automaton translated from Figure II-1. An important item of information is lost when translating an MSC into an automaton. Components are decoupled from their messages and ownership of each message is lost. MSCCheck assumes a universal definition of a message and unique ownership by a component so that there is no ambiguity in regards to ownership of a message.
There are two types of MSC to automaton translations that should be addressed: global and local. For our purposes, the global and local automaton translation strictly follows the definition in [FK07]. For global translation, the MSC is taken as a whole. Each CMSC construct is evaluated until a BMSC is located. The BMSC is then evaluated and each send and receive message is paired as in Figure II-1. A construct specific transformation is applied to the resulting automaton for each construct that is visited. The final global automaton reflects the entire interaction sequence (i.e. sends with corresponding receives) view of the MSC. On the other hand, the local translation’s view is on each component’s interaction with the environment of the MSC. Component specific messages are captured in the resulting automaton that will not have a common send or receive message. A transformation similar to the global transformation is applied to the local messages of a component for any constructs that are encountered on the component’s lifeline. An example of a local transformation of the Cmpnt1 component can be found in Figure II-6. For a formal description of the translations refer to [FK07].

II.H Decomposition Property

An MSC is said to satisfy the decomposition property if its global and local languages are harmonious. In order for an MSC to satisfy the decomposition property it must fall under the restriction of causality. The basic MSC found in Figure II-1 satisfies
the decomposition property because of its local and global duality. In fact, all basic MSCs satisfy the decomposition property [FK07]. The MSC in Figure II-7 is an example of an MSC that violates the decomposition property, because the global and local semantics do not agree. The global language has Message2 dependent on Message0 and Message3 dependent on Message1. Locally, Cmpnt0 and Cmpnt2 can send either of their respective messages but are unknowledgeable of the dependency relationship that occurs globally. This relationship is only implied globally.

While non-causal MSCs are indeed valid in an MSC specification, the decomposition property is important in compositional verification of MSCs, because it localizes erratic behavior to individual components. Thus, the remaining specification MSCs in this paper are restricted to causal MSCs.

II.1 Services and Service-Oriented Systems

The traditional view of a service is seen as a stateless function that accepts an incoming request and returns a result or response through a well-defined interface. For our research, we adopt the definition of a service found in [EHK07]. We view a service as defined by the “interaction among the entities involved in establishing the service.” By this definition, services are composed of stateless functions and represent partial behaviors of the whole system. In this regard, a service is defined by its intended purpose. This view of a software service has also been cited as a feature in some of the literature [JZ98].

According to our view of a service, the service can be treated as the principal system design and implementation entity for a service-oriented system. The service-
oriented system notion is popular today due to its ability to improve the coupling of software components in distributed systems. Components have particular roles and duties assigned to them, and requests for services from these components can be implemented through a message passing mechanism [KM05].

Many service-oriented architectures exist today such as .NET, JavaOne and CORBA. Architectures such as these provide an excellent implementation infrastructure for the service-oriented approach, but fail to provide a modeling abstraction of the system. Through our chosen system design, a user can use MSCs to provide this abstraction and then automatically map the semantic information from the MSC specification over a service-oriented architecture.

II.J System Properties

A property of a system is a characteristic of a system. Properties are usually separated into two classes: liveness and safety [Lam77].

A liveness property is a property of the system that asserts that “something good happens.” Liveness properties are violated when infinite traces of the system’s execution never see a “good thing” occur. The focus of liveness is on the future of a computation from one or more particular events. An example of a liveness property found in the MSC in Figure II-1 could be that “Message3 must be sent subsequently after Message2 is received.” Therefore, if Message3 is never sent in the future, following a received Message2, the property is violated.

The second class of properties is known as safety. Safety properties assert that “something bad does not happen.” These types of properties are flagged when a finite trace of the system’s execution results in a “bad thing” occurring. While liveness
properties are more focused on the future of a computation, safety properties delve into the history of a computation up to one or more particular events. For the sake of example, assume that the MSC in Figure II-1 is the only MSC describing this “Message0 service” of the system. A safety property that could be insinuated from the MSC would be “the send of Message3 is only followed immediately after the receipt of Message2.” Therefore, if a send of Message3 is followed immediately after the receipt of Message1 or any other message in the system, the safety violation would be raised. The violation will also be raised when Message3 is the first message in the system (i.e. before Message0 occurs). As one can see, this differs from the liveness property in the previous example which only asserts that Message3 should occur sometime in the future after the receipt of Message2.

II.K Verification

Verification is “confirmation, by examination and provisions of objective evidence, that specified requirements have been fulfilled. In design and development, verification concerns the process of examining the result of a given activity to determine conformity with the stated requirement for that activity [IEE07b].” In other words, software verification is the process of checking that the system model correctly implements the designer’s conceptual model of the system. The goal of verification is to prove that the actual system model meets desired properties or behavior. Verification does not intend to find all defects or anomalies in a system, but rather increase the confidence that such a system is correct.

The most familiar type of software verification is static analysis. Static analysis can further be divided into two categories: abstract interpretation and model checking.
Since our research is focused on model checking, we will only give a brief background of model checking.

Model checking [CE81] is a method for formally verifying finite-state software systems. The state-space of the model is expressed as either a computation tree or a finite-state machine. Properties of the state-space are represented as terms in either temporal logics or Büchi automata. Efficient symbolic algorithms are then used to traverse the state-space to verify that these properties are never violated. Examples of such tools that implement these algorithms are SMV [McM93] and Spin [Hol91].

One of the on-going problems of model checking is the state-space explosion problem [Val98]. This problem occurs when the set of states that the system can reach are very large or grow exponentially as new components are introduced to the system.

One approach towards circumventing the state-space explosion is through composition [Dij65]. Through composition we can reason about a system’s environment based on the reasoning of its individual parts rather than of the system as a whole as in a divide-and-conquer strategy. In the case of verification of a component-based system, the compositional verification of individual components can reduce the state-space explosion problem from the system-level to the component level. Every component is verified individually, and the system can be considered completely verified when all components are verified.

The technique used for compositional verification in our system follows the assumption-commitment paradigm [BK98], sometimes referred to as rely-guarantee. This paradigm sets a certain number of conditions on the environment of a component to
guarantee the proper behavior of that component. All unnecessary environmental behaviors are abstracted away from that component when it is verified.

Since MSCs give the information needed for understanding a component’s interaction with its environment, we can use MSCs to compositionally verify a component. As stated earlier however, the MSCs must satisfy the decomposition property in order for the MSCs local semantics to coincide with its global semantics. The algorithm for decomposing a causal MSC and verifying its components is given in Chapter IV.
Chapter III
MSCCheck User Scenarios

The purpose of this chapter is to demonstrate the use of the MSCCheck tool from
the perspective of the end-user. The use of MSCCheck follows the design and
implementation phases of development with a feedback loop to each of these phases.
Figure III-1 gives a view of this cycle. We follow through the intended workflow by
visiting each phase using an example taken from the automotive domain. Focus will be
placed primarily on the “Check” phase and MSCCheck’s role during this phase. A
detailed description of the semantics of inputs and outputs, algorithms and
implementation decisions are covered in the following chapters.

Figure III-1. Intended Workflow using MSCCheck

III.A MSCCheck Features

MSCCheck takes an implementation of a software system in the form of automata
in addition to the system’s specification in the form of MSCs. The specification is
composed of both a causal description and properties of the system. The user picks the
causal specification documents that cover the properties she or he wants to check against
the implementation. Based on those choices, MSCCheck checks each component’s
implementation against the properties using the specification document, and if any erratic behaviors are located, the user will be presented with MSC traces that capture the erratic behaviors. The following example system will showcase these features of the tool with relation to the feature driven development process.

**III.B Example System: Central Locking System**

As an example of the use of MSCCheck, consider the Central Locking System (CLS) of a car. Conceptually, this system provides two services for the car: locking and unlocking. The CLS is composed of a number of interacting components:

- A key fob (KF) that enables the user to send a signal to the car with the command to unlock or lock the doors.
- A controller (C) that controls all incoming messages from the user and relays the appropriate calls to other components to provide the service.
- A lock manager (LM) that controls the locks of the entire car.
- A database (DB) that loads car preferences for a driver.

When a car is unlocking, the user presses the unlock button on the KF. A signal is sent to the C that instructs the LM to open all doors of the car. When a car is locking, the same type of scenario occurs. The user presses the lock button on the KF. The C then commands the LM to lock all doors. These two services/features of locking and unlocking will be designed and verified throughout the rest of the chapter.

**III.B.1 Phase 1: Service Design Use Cases**
The first phase requires the use of an MSC diagramming tool to build the specification model of the two services. For this example, we will use M2Code [Moo06] which is described further in Chapter V. The following use cases should occur when the designer is using the service definitions given to him or her from the feature set for creating the specification of the system. This is synonymous with the “Design by Feature” activity of the FDD process given in Figure I-1.

**Step 1.** Designer draws several causal “system” MSCs in M2Code that describe the intended system behavior. This includes the HMSCs that join all the system MSCs together.

![High-level MSC for the CLS](image)

**Figure III-2.** High-level MSC for the CLS

![Unlocking MSC “Unlocking”](image)

**Figure III-3.** Unlocking MSC “Unlocking”

![Locking MSC “Locking”](image)

**Figure III-4.** Locking MSC “Locking”
The services of locking and unlocking can easily be specified as MSCs. Figure III-2 gives a High-level MSC (HMSC) of the CLS. On the left half of the figure are two MSCs, “Unlocking” (Figure III-3) and “Unlocking-2,” that are joined to perform the unlocking service that follows the scenario of a user wanting to unlock the car doors from his or her key fob. On the right of the HMSC are two MSCS joined to perform the locking service. The “Unlocking-2” and “Locking-2” MSCs are extension placeholders for other MSCS that can occur during the “Unlocking” and “Locking” service respectively. Currently the MSCs are empty. Note that these MSCs have component states symbolized by the diamond shaped mark at the beginning and end of the lifeline of each component. States of components are ignored by MSCCheck, and interactions are assumed to occur at any state of each component.

**Step 2.** Designer draws several “property” MSCs in M2Code describing system properties.

![Figure III-5. Locking Property MSC “LockingProperty”](image)

Figure III-5. Locking Property MSC “LockingProperty”
The designer specifies the type of properties the system should have. In our case we have two properties checking the locking and unlocking services respectively. The property in Figure III-5 could be read as “Ensure there’s always a close_ok message received by the C component following an unlck received by the C component.” Simply stated, there should be a close_ok message between two unlck messages. The property in Figure III-6 follows in the same manner. Also note that property MSCs can be non-causal.

**Step 3.** Designer clicks “Export to MSCCheck…” on M2Code which creates files for each property and system MSC.

**III.B.2 Phase 2: Build by Service Use Cases**

**Step 4.** Using the services specification, the implementer creates an implementation of the system.

The implementer uses the MSCs from M2Code to create an implementation for each component. An implementation automaton for each component is synthesized from the actual implementation. This should be accomplished by utilizing an outside tool
whose final output should be converted to MSCCheck’s automaton input format. Figures III-7, III-8, III-9 and III-10 give the implementation automaton for the C, KF, LM and DB components respectively.

![Figure III-7. Implementation Automaton for C](image)

![Figure III-8. Implementation Automaton for KF](image)

![Figure III-9. Implementation Automaton for LM](image)

![Figure III-10. Implementation Automaton for DB](image)

The semantic and implementation details regarding the automata in this phase are given in Chapter IV.

III.B.3 Phase 3: Integration Verification Use Cases

The verification phase runs the automata through a model checker using the compositional verification algorithm described in the following chapter. This phase requires the user to interface with MSCCheck inside the Eclipse IDE [ECL07].

**Step 5.** Designer creates a new MSCCheck configuration.
The configuration file holds essential information for the verification and generation of the system under development. The designer creates a “Simple project” in Eclipse and uses the “MSCCheck Configuration Wizard” to create the configuration file.

The user clicks “Browse…” to select the Eclipse folder containing the implementation, properties, and system folders. The user then clicks “Compile System” which brings up a screen similar to Figure III-11. On the left a listing of the implementation components is viewable. Clicking on any of the component names brings up a view of the automaton that represents the component’s implementation. On the right a listing of all the MSC properties found in the property folder are given. Alongside each property is a listing of potential causal system MSCs that the user can choose as a best suited verification MSC. If a potential causal MSC is missing for a property it is unverifiable and thus grayed out from selection. Above the component and property listing is a parameter for the maximum amount of violations allowable per a given component property coupling (defaulted as “3”).
**Step 6.** User selects components and properties to be verified and clicks "Verify System."

Inside the configuration editor, the user selects the components and properties to be verified. In order for the system to be considered fully verified, all components must be selected. Once the user clicks “Verify System,” the system is checked for any inconsistencies and property violations.

**Step 7.** User views listing of violated properties (probable defects).

![Figure III-12. MSCCheck Defects View with Violations](image)

The designer notices that MSCCheck has completed verification, and realizes an Eclipse View entitled “MSCCheck Defects” has been filled with a description of the violations (see Figure III-12). This view is a table consisting of three columns of information. The first column contains the name of the property that has been violated. Icons next to the property indicate the type of violation or warning. The second column lists the causal system MSC that was used to raise the violation, and the third lists the violating component.

There are two types of violations that the MSCCheck Defects View will display. The first type is a no causal MSC found warning. This is created when the “Compile” button is pressed and no causal system MSC can be found for a specific property. This
syntax error is denoted by a yellow warning symbol. The second type of violation is a property violation raised during a model check of the system. Figure III-12 shows that the offending component of the “LockingProperty” (Figure III-5) and “UnlockingProperty” (Figure III-6) property MSCs is the C component and the causal MSCs’ that were used to verify the component were “Locking” and “Unlocking” respectively.

<table>
<thead>
<tr>
<th>Violation</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Causal MSC</td>
<td>Warning</td>
</tr>
<tr>
<td>Premature Send</td>
<td>Defect (Safety)</td>
</tr>
<tr>
<td>No Message Occurred</td>
<td>Defect (Liveness)</td>
</tr>
<tr>
<td>Extra Behavior</td>
<td>Defect (Safety/Liveness)</td>
</tr>
</tbody>
</table>

These violations lead up to a discussion on the violations that MSCCheck is able to detect. A table of these violations is given in Table III-1. The first type of violation, Non-Causal MSC, is raised if a non-causal MSC could not be found that could potentially cover verification for a specific property. The second type of violation is a premature send. If a component sends a message during a system state where it should be performing some other behavior (idling, sending or receiving another type of message), then a safety violation occurs. The third violation occurs when a component fails to send a message to the environment and stays in idle. The final type of violation is any extra behavior that a component may be performing that violates the boundaries of a property. This is a hybrid of the previous two types of defects. During these types of violations, the component has a behavior consisting of alternative events that interact with the environment as “outside” messages. These outside messages are not defined in the chart and will lead up to a livelock, a deadlock, or a premature send performed by the system.
Step 8. User clicks the “UnlockingProperty” violation from “Defects View” and views the counterexamples.

Figure III-13. Counterexample View for “UnlockingProperty”

The counterexample editor displays three artifacts to the user given by separate tabs in the bottom of the Counter Example Editor. The first is the counterexample artifact. This is a view of the interaction between the component that violated the property and its system environment that was produced by the causal MSC. There is the possibility of finding several counterexamples for one component. In our example’s case, the “UnlockingProperty” states that an open_ok message should always occur between two unlck messages. The counterexample in Figure III-13 shows three components. The C and the [ENVIRONMENT] components are the two components that are interacting with each other where C is the component under verification and the environment are those components that were extracted from the causal MSC. The
[ERROR] component is a placeholder component for messages that are sent but never are received during the trace. In this particular counterexample, an open_ok message is received by the [ERROR] component. This is prefixed by a lambda transition or message marked with “[L].” This message is ignored by the verification engine, but offers the user a more defined trace of the execution between the component under verification and its environment. The defect is a result of one of two scenarios. The C component is missing behavior (missing an open_ok received) or it has extra behavior (the fourth type of violation) following the send of an “open” message that sets it in a state where it cannot receive the open_ok message. In theoretical terms, a liveness property is violated because the [ENVIRONMENT] component never proceeds after a close_ok is sent. The violating message is marked with an “[ERROR]” tag and ends at the [ERROR] component. The extra open_ok message is a result of the [ENVIRONMENT] component being “stuck” in its state and should be ignored by the observer. The remaining two artifacts in the counterexample view are the property MSC that was violated and the causal MSC used during verification.

**Step 9.** User decides on best solution for counterexample.

Recognition of the cause of the violation is placed on the user. It is important to check the property MSC, causal MSC and component’s implementation to determine which is incorrect or incomplete. The user looks at the C component’s implementation in Figure III-7 and views the causal MSC in Figure III-3. Comparing them with the counterexample and following the “refine” feedback loop from the check phase in the MSCCheck workflow, the user begins to strategize a solution to the problem. The implementation of the C has extra behavior that allows it to communicate with the DB.
This behavior is unaccounted for in the specification. The user decides to explore solutions to the problem being the causal MSC used to describe the property.

A number of solutions have been given for arriving at a complete scenario void of violations/incompleteness such as those in this example. In [WK04] the authors describe a methodology for refining scenario-based requirements specifications such as MSCs to arrive at a more complete specification. Issues are addressed in the form of “question-action” pairs, where the question is asked, and, based on the answer found under the action, the action is either taken or not. Most of the actions require the introduction of new structural notation into the specification documents. The authors argue that nominal scenarios are commonplace for the majority of scenario-based documents in a specification. In order to arrive at a more complete scenario-based specification, one must consider the optional, alternative and less common behaviors. These are arrived at by using the nominal scenarios as a starting point and following the “question-action” pairs to aid the designer/developer through completing the specification. MSCCheck complements this strategy if an implementation already exists for system. MSCCheck will raise violations of the specification when alternative behaviors are detected from inserting a component into a specification. The user simply needs to understand the counterexample to determine that extra behaviors exist in the system that are correctly created by a component and that the specification is incomplete in that it does not capture these extra behaviors. The “question-action” pairs currently supported by MSCCheck are given in Table III-2.
### Table III.2: Samples of Issues in question-action pairs from [WK04] Currently Supported by MSCCheck

<table>
<thead>
<tr>
<th>Issue</th>
<th>Question</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Could <em>message</em> be replaced with another message without changing the behavior of the scenario?</td>
<td>If yes: replace <em>message</em> with a combined fragment with interaction operator <code>alt</code> and the two alternative messages as operands.</td>
</tr>
<tr>
<td>2.3</td>
<td>Is the message a choice point? – i.e., could an alternative message have appeared at this point that would change the following behavior?</td>
<td>If yes: encapsulate the existing and new behaviors in operands of a combined fragment with an <code>alt</code> operator, and introduce guards for the operands if necessary.</td>
</tr>
<tr>
<td>2.6</td>
<td>Does <em>message</em> have a guard?</td>
<td>If yes: introduce alternatives when the guard is not satisfied (using <code>alt</code>).</td>
</tr>
<tr>
<td>2.7</td>
<td>Can <em>message</em> fail?</td>
<td>If yes: capture the failure handler as a separate interaction diagram referred to (using <code>ref</code>) in the main diagram and use an <code>alt</code> operator to capture the alternative when the message fails.</td>
</tr>
<tr>
<td>2.11</td>
<td>Does <em>message</em> really depend on all its predecessors?</td>
<td>If no: extract the dependent messages into a separate scenario.</td>
</tr>
</tbody>
</table>

The user refers to the full table of issues and arrives at relevant questions (Table III-2). After answering each of the questions and referring to the implementation of the C component, the user notices the extra behavior found in the C component’s implementation for interaction with the DB component. Upon this realization, the user decides that Issue 2.11 is the most relevant and that the querying of the database for user presets is essentially another service of the system that requires its own MSC. The semantics of the Unlocking service is now changed and calls upon the “Query User Presets” service after a successful unlock.
Step 10. Designer refines models from “Phase 1.”

To form a more complete scenario, the “Query User Presets” is added to the feature repository, and the “Unlocking” MSC is updated by joining the “Query User Presets” service on the “open_ok” message. The designer refines the system specification in M2Code and arrives at the MSCs given in Figure III-14 and Figure III-15.

![Figure III-14. Query User Presets MSC](image)
![Figure III-15. Unlocking MSC joined with the Query User Presets MSC](image)

Step 11. Implementer refines implementation created in “Phase 2” and verifies C’s implementation with the new specification.

After review of the updated specification, the implementer refines the implementation of the C component explicitly separating the Unlocking service from the Query User Presets service and arrives at an implementation found in Figure III-16.

![Figure III-16. Refined C implementation](image)
III.B.4 Phase 4: Release

Once the system and specification have been thoroughly verified for defects, the release manager checks the new component code into the final release. Any new services are then chosen from the features list and a new iteration of designing and verifying takes effect.
Chapter IV
MSCCheck Foundations

The purpose of this chapter is to explain the underlying algorithms that drive MSCCheck and their relationship to the Integration Verification Phase example in the MSCCheck User Scenarios.

IV.A Automaton Synthesis

Automaton synthesis is essential to the compositional verification used in MSCCheck. Each system MSC must be decomposed into local automata for each component in the entire system. The local automaton of a component in a MSC represents its local language for that MSC. All local automata are then composed using a cross product operation that yields the global automaton.

IV.A.1 Global Automaton

As an example of the synthesis of automaton, reconsider the Unlocking MSC given in Figure III-3. Following the MSC to automaton synthesis algorithm described in [FK07], we arrive at the global automaton in Figure IV-1. This automaton displays the sequential information of all the message exchanges between components in the MSC and represents the global language of the Unlocking MSC.

Each state in the global automaton is represented by a set of active events. The set of active events is a set of component events that are capable of reacting at this
particular state. For instance in the global automaton for the Unlocking MSC, state 1 is composed of the active event set \{!\text{unlck}, ?\text{unlck}, ?\text{open}\} and state 2 is composed of the active event set \{?\text{unlck}, ?\text{open}\}. State 1 has three components, Key Fob (KF), Control (C) and Lock Manager (LM) with the !\text{unlck}, ?\text{unlck} and ?\text{open} events (respective to each component) actively waiting to react. By State 2, the KF component has exhausted all of its events and only events from C and LM are active. Send messages are always ready to react and need not wait, while receive events need to wait for their common send message to be sent before it can react.

**IV.A.2 Local Automaton**

The local automata are generated as a vertically filtered or projected version of a global MSC. Only the messages associated with a role or component are considered for this type of automaton. Local automata for the C, LM and KF components of the Unlocking MSC are given in Figure IV-2, VI-3 and VI-4 respectively. This was generated by proceeding down the axis of the each component’s instance in the Unlocking MSC and marking every outgoing and incoming message as a transition in the final automaton. The states of each component are composed of the active events associated only with that component and not at a global scale. In addition to the components in the global automaton, since we are taking the local automaton from the global space, each component known to the system’s local automaton must be created in order to have a complete system. Therefore, for this example the Database (DB) component will have a local automaton associated with it. Essentially all components
unassociated with a global automaton will be represented as one state that continuously
listens for outside messages (messages unassociated with the MSC).

![Figures IV-2 to IV-5: Local Automata for KF, LM, C, and DB Components]

**IV.A.3 Automaton Composition**

To compose a global automaton from local automata, the cross product of all the
local automata is performed. The cross product is used to “glue together” the local
automata of an MSC by ordering send messages before their associated receive messages.
Essentially the cross product between two local automata is accomplished by starting at
the initial states of the automata and proceeding by visiting each state in the two local
automata in parallel. The active events in the states will determine which events can react
by following a simple rule: “If the event is a send event, a receive event with no common
send event, or a receive event whose common send event has already occurred, the event
can proceed.”

![Figure IV-6: Cross Product of LM and KF]
As an example, consider the cross product of LM and KF (Figure IV-6) taken from the local automaton of the two components (Figure IV-2, IV-3). The two local automata contain no common messages; therefore, all events are reactive at every state. State 4 in the cross product is the composition of States 0 and 0 in the two respective local automata. Its active events are \( ?\text{open},!\text{unlck} \) which are both reactive. This is shown by the transitions to States 0 and 1 in the cross product automaton. The composition of active event states continues until all events are exhausted. Once completed, the cross product automaton is cross produced with the C and DB component’s local automaton which will then manufacture the global automaton found in Figure IV-1.

For a more formal description of component synthesis including construct transformations see [FK07].

**IV.B Compositional Verification Algorithm**

Following the generation of the three inputs, MSCCheck is used to verify that the property automata are never violated in the state-space of the system. This section explains the compositional verification algorithm.
To verify that a system is correct it must be proven that the interactions of the synthesized automaton are contained inside the interactions of the MSC specification. Therefore, two statements must be proven. First, the interactions of a system MSC must be contained inside the interactions of a property MSC. Secondly, every component in the system MSC must have its implementation inserted into the environment of the system MSC. This is accomplished by performing a buffer cross product of the component's implementation automaton with the remaining local automata of the components in the system MSC, referred to as the environment of the component. To be considered fully verified, the interaction patterns of this generated automaton must be contained inside the interactions of the original system MSC. For a formal proof of this process refer to [FK07]. Figure IV-7 gives an overview of the compositional verification process.

IV.B.1 Verifying a Causal MSC Satisfies a Property MSC
As stated earlier, the first step of the compositional verification algorithm is to find a causal system MSC that satisfies the non-causal property MSC that we want to check against the implementation. MSCCheck is able to detect system MSCs that satisfy causality and are good candidates for satisfying a property MSC. However, it is up to the user to choose if a system MSC accurately represents a property MSC. In the second step of the verification process, the selected causal MSC will be decomposed by component to verify the implementation.

Figure IV-8. Property MSC

Figure IV-9. Causal MSC

MSCCheck cannot verify that a causal MSC satisfies a property MSC due to non-determinism in the language of the model checker it uses. If at any point there is a non-deterministic choice in the property and the model checker chooses incorrectly, the model checker flags it as an error and does not rollback to the other choices.
As an example, consider the sample property in Figure IV-8. In natural language this property means “C1 and C2 communicate message1 infinitely OR C1 and C2 communicate message1 finitely.” A causal MSC that expresses this property is shown in Figure IV-9. Translating these MSCs to their associated global automata yields the automata in Figures IV-10 and IV-11. Visually, it is clear that in Figure IV-11 every choice point is accepted by the property. The traces starting at state 1 in the causal MSC’s automaton are accepted starting at state 1 in the property automaton. The traces starting at state 6 in the causal MSC’s automaton are accepted by the choice at state 4. The empty trace starting and ending at state 5 in the causal MSC is accepted at state 3 in the property automaton. However, because of the guard conditions of transitions of a property in our input language to the model checker, all traces of the causal msc’s global automaton run for all choice points that are true of the property automaton. This will create false positives as counterexamples during the verification process. For example, the traces of the causal MSC’s global automaton starting at state 1 are finite. Following in lock-step with the property, the causal MSC’s global automaton reaches state 3 and performs a ![message1] transition. During this transition, the property automaton has three choices starting at states 1, 3 and 4. Since this is a non-deterministic decision, two choices are possible. Both choices are taken, with taking the transition from state 1 to
state 2 being the correct one and taking the transition from state 4 to state 5 being the incorrect one. A false positive is created when the causal MSC’s automaton terminates at state 5 and the property terminates at state 4 raising a liveness property violation.

Although MSCCheck does not automatically verify that a causal MSC satisfies a property due to the problem of non-determinism, it gives a list of probable candidates to the user. The user can manually inspect the list and choose the best candidate for use during the verification of the implementation step.

**IV.B.2 Verifying an Implementation Satisfies a Causal MSC**

Using a causal MSC found in the first step of the compositional verification algorithm and each component’s implementation as input, the algorithm first extracts the local automaton of each component in the causal MSC. This includes automaton for components that are part of the system’s implementation but do not provide any service in the causal MSC. The algorithm then inserts each implementation component into the environment created by the causal MSC. This yields all behaviors the component will perform when interacting with its environment. The causal MSC is then used as a property that the resulting automaton must satisfy. Once all components of the implementation undergo a buffer cross product with their respective environment and no violations occur for any component when checked against the original MSC, then the system can be considered verified for the original non-causal property.

**IV.B.2.a Buffer Cross Product of an Implementation and its Environment**

In order to perform the verification of the implementation using the causal MSC that the user selects in the first step of the process, we must visit the buffer cross product
operation. This operation ensures that the common receive events of components always follow their common send events sometime in the future. It deviates from the cross product operator in that an implementation’s ordering of messages and the number of messages it can send. Theoretically, an implementation component can send an infinite amount of messages and the environment can infinitely wait to receive them. A buffer is introduced between the executions of the two automata and ensures fairness in consumption of the messages. Each send message that is intended to be received by the opposite component is stored in the buffer where the receiving component can later consume it.

IV.B.2.b Locating Deadlock and Livelock

In addition to temporal property violations, there are deadlock and livelock violations that can occur during the verification step. Both types of violations occur during the buffer cross product operation when certain conditions are raised.

A deadlock occurs when the implementation and the environment automaton are unable to continue progress in communicating with each other. The environment and implementation components have active events in their respective states, but those events not reacting due to one of two conditions. The first possible condition occurs when the active events in the component’s current state are receive messages that contain no send messages ready to be consumed in the buffer. The second possible condition occurs when the current state consists of only send messages that the buffer is blocking because it is full. When the current states of the implementation and environment meet these conditions, the system is considered deadlocked and a violation is raised.
A livelock occurs when we arrive at a point in the execution of the buffer cross product operation that one component’s active event state can no longer progress, but the opposing component is progressing as usual. A simple test for this is to find cycles in the buffer cross product operation’s resulting automaton where each state in the cycle has one component continually “starved” of progress, but the other is not. While performing the buffer cross product operation, the current buffer cross product state is analyzed for this condition. If it satisfies this condition, it is marked as a possible livelock state. The buffer cross product state that the previous livelocked state has progressed to is then analyzed for the livelock condition. If the state of the implementation or environment automaton that was under consideration for deadlock is able to progress then it is considered to have removed the livelock. In this case, all states that were marked as livelock candidates and progress to this state are recursively removed from the set of livelock states. In the case of the buffer cross product state that was progressed from a livelock state is still livelocked, the state is marked as livelocked. If the next progression of the buffer cross product operation is into a buffer cross product state that is already marked as livelocked, then the system is considered to be livelocked for that particular behavior of the system.

IV.B.2.c Instance of Deadlock Using the CLS Example

As an example, reconsider the “UnlockingProperty” property MSC in Figure III-6. Searching for an MSC that has its interactions contained within this property would produce the “Unlocking” MSC in Figure III-3. This satisfies the first half of the compositional verification algorithm. The second half requires a buffer cross product of
the implementation automaton of every component in the system with its environment (the remaining local automata of the components in the MSC). The “Unlocking” MSC has three components: KF, C and LM. If we started the verification process with the C component, we would first take the local automaton of KF and LM from this MSC and perform a cross product operation on all 3 components. This yields the environment automaton for C (Figure IV-6). We then would take the implementation automaton of C (Figure III-7) and attempt a buffer cross product with the environment automaton. The resulting buffer cross product automaton (Figure IV-13) is checked for containment with Unlocking’s global automaton (Figure IV-12). The “Unlocking” MSC is translated into an infinite looping MSC to match the original “UnlockingProperty” which is an infinite looping MSC. As seen from the counterexample produced in the previous chapter, the buffer cross product automaton of C and the environment produced from the “Unlocking” MSC does not satisfy the “Unlocking” MSC’s property. This is easily noticeable by examination of the buffer product automaton in Figure IV-13. Starting from state 8 and ending in state 0, the implementation and environment progressed as intended. At state 0 there was no progress from the two automata. It does not satisfy the liveness property of the system that states that the C component should always be receiving and open_ok message between the receipts of two unlock messages. An open_ok message is never received by the implementation automaton because of the deadlock that occurs in state 4 of the implementation automaton.
Figure IV-12. Global Automaton of “Unlocking” MSC with an Infinite Loop Construct

Figure IV-13. State Space of Cross Product of C implementation and Unlocking Environment
Chapter V
Architecture and Implementation of MSCCheck

The purpose of this chapter is to describe the tools that the MSCCheck plug-in depends on for proper execution and MSCCheck’s design details. The chapter follows through each phase of the intended life cycle, examining the design details of MSCCheck, relationships to third-party applications and input and output file formats used during that phase.

V.A System Architecture and Packaging

Figure V-1. Architectural Overview of MSCCheck

Figure V-1 shows the high-level architecture of MSCCheck and its inputs. MSCCheck resides completely within the Eclipse framework and has its own extensions.
for functionality such as viewing counterexamples and external calls to plug-ins such as Bogor, the model checker used to verify the MSCs, and third-party applications such as GraphViz for viewing state models [GRA07].

![MVC Architecture of MSCCheck](image)

**Figure V-2. MVC Architecture of MSCCheck**

Logically, MSCCheck uses the Model-View-Controller architectural pattern (Figure V-2). The model consists of Java classes that provide means of encapsulating objects such as the MSCs, defects and automata. The controller layer provides an interface between Eclipse-specific classes and the model. Controller components include a verifier that controls the inputs and outputs while executing the compositional verification algorithm and a code generator that creates the input code for the model checker. The top-most layer, the view, includes components that allow the user to update configuration files, interact with the verification process and view automata, defects and MSCs. The MVC pattern was primarily used as a framework for developing MSCCheck to allow for modular development and separate testing of the controller and model layers.
where the majority of complexity exists. As a consequence, the Eclipse-specific view layer contains a number of components that are responsible for translating unrecognized Eclipse data (i.e. MSCs and automata) into an Eclipse recognizable format.

The Java code of MSCCheck is organized into four primary packages. These four packages are:

- **MSCCheck.editors**
  - Eclipse user interface extensions that enable the user to view counterexample traces and edit configuration files.

- **MSCCheck.utilities**
  - interfaces and classes that provide mechanisms such as verification and objects for model data encapsulation.

- **MSCCheck.views**
  - Eclipse user interface extensions that enable the user to view list of counterexamples.

- **MSCCheck.wizards**
  - Eclipse user interface extensions that enable the user to create a new MSCCheck configuration.

The **MSCCheckPlugin.java** file contains the entry point for Eclipse to load MSCCheck as a separate plug-in. Since Eclipse plays an important role in MSCCheck, it is necessary to give the reader a synopsis of the tool.

V.A.1 Eclipse
Integrated Development Environments (IDEs) are commonly used to shorten development time by providing an integrated toolset for common programming tasks. Tools typically present in an IDE include syntax-driven editors, source code browsers, GUI builders, compilers/linkers and debuggers. Often, however, such tool sets are too generic for the programming task at hand: increasingly, development environments need to take the application domain and the corresponding design knowledge available into account to provide adequate support for the developer. Because most commercially available IDEs are bound to one particular programming language, operating system, or tool suite, their flexibility regarding extensions and customization is limited. Eclipse [ECL07] is an open-source project that addresses this deficit by providing a flexible framework for integrating highly customized development tools into one generic and extensible IDE. These tools, known as plug-ins, can access all the information available in Eclipse to support advanced development tasks such as verification and validation.

Eclipse is an open source, cross platform compatible IDE that is commercially funded by IBM. In its plainest form, Eclipse is a framework capable of being extended through the use of user designed plug-ins. Due to this fact, it is known as an “Integrated Everything Environment” (IEE). The proper integration of this IEE’s plug-ins is directly proportional to the enhancement of its power.

Eclipse comes pre-bundled with plug-ins for Java development. By exploiting the information generated in these plug-ins, extremely powerful plug-ins can be created and seamlessly integrated into the IEE. In our case, we have written MSCCheck by creating our own plug-in that extends several plug-ins that communicate with each other and cooperate with other Eclipse plug-ins to provide verification of service-oriented systems.
V.A.1.a Eclipse’s Plug-in Architecture

A plug-in is the means by which users add new tools to the Eclipse environment. It can be anything simple such as a help system, or anything complex such as a testing tool. Complex plug-ins consist of many small plug-ins that are integrated to compose the plug-in’s full functionality. This integration is formed through XML manifest files that describe the relationships between plug-ins. These manifest files consist of a list of a plug-in’s extension points. It is through extension points that Eclipse is able to associate plug-ins and extend them to communicate with each other through a common interface. Plug-ins are activated during runtime, so the memory footprint is not very large.

V.A.1.b Eclipse’s Plug-in Development Environment

To aid in the rapid development and deployment of plug-ins, Eclipse contains a Plug-in Development Environment (PDE) plug-in. This plug-in contains generic code for wizards and editors that set up the base code for developing, debugging and deploying plug-ins into Eclipse. The PDE was used in our project to develop and test MSCCheck. It also contains a manifest file editor that allows the user to visually describe the relationship between plug-ins and their extension points.

V.B Phase 1: Design Activities

The design phase consists of using M2Code or another system design tool. MSCCheck was designed to be flexible to integrate with a MSC design tool such as M2Code by providing a mechanism of input in XML format. The MSC design tool is intended to produce the MSC properties and specification (causal MSCs) of the system.
The implementation of each component is also intended to be synthesized from an outside tool.

V.B.1 M2Code and Implementation Issues

M2Code [Moo06] is an extension for Microsoft Visio that enables the user to visually create MSCs. Once graphed, the MSC interactions can be translated into automata. Currently, M2Code has two outputs that have a foundation on roles. The concept of a role in M2Code is seen as the part that a component plays in the performance of a service. Therefore, each component is mapped to a role that it plays in each service. The first output of M2Code is a model of the structure of the relationships between roles known as a role domain model. The second is an automaton for each role defined. In order for the M2Code specification to be used in the verification process by MSCCheck, it must be extended to contain a mechanism for producing an XML formatted document for each MSC in the M2Code Visio file. Since MSCCheck is ignorant of the concept of a role, each role can be treated as a component. M2Code currently views all MSCs embedded in a document as system MSCs. In other words, if a property MSC were to be created inside the project they would be regarded as behaviors of the system rather than properties of the system. Thus, in addition to XML output capabilities, a mechanism must be implemented inside of M2Code that marks a distinction between system and property MSCs.

V.B.2 MSCCheck XML Input File Structure and Java Representation

To allow for flexibility of future change, MSCCheck uses an ad-hoc XML-based input to describe the structure of the required implementation automaton and system and
property MSCs. The two types of input are distinguished by their file extension and directory location. The format of the extension nomenclature is: for,

- Implementation Automaton: <component name>.msccheck-buchi
- System and Property MSCs: <MSC name>.msccheck-msc

To parse and read all XML files, MSCCheck uses the SAX reader [SAX07] with the Crimson XML parser [CRI07]. The classes used in generating a Java automaton model are found in the MSCCheck.utilities.model.automata package and the classes used for generating MSC models are found in MSCCheck.utilities.model.msc.

V.B.2.a Implementation Automata

\[
\text{BÜCHI AUTOMATON := \langle Büchi \rangle INISTATES ACCEPTSTATES TRANSITIONSCONSTRUCT \langle Büchi \rangle} \\
\text{INISTATES := \langle initialstates \rangle ID \langle initialstates \rangle} \\
\text{ACCEPTSTATES := \langle acceptstates \rangle ID \langle acceptstates \rangle} \\
\text{ID := \langle id \rangle \langle NUMBER \rangle} \\
\text{TRANSITIONSCONSTRUCT := \langle transitions \rangle STATE \langle transitions \rangle} \\
\text{STATE := \langle state \rangle \langle id \rangle \langle NUMBER \rangle \langle transition \rangle \langle dest \rangle} \\
\text{TRANSITION := \langle transition \rangle \langle SYMBOL \rangle \langle DESTINATION \rangle} \\
\text{SYMBOL := \langle symbol \rangle \{'\?\' | '!\}' \langle TEXT \rangle} \\
\text{DESTINATION := \langle dest \rangle \langle NUMBER \rangle} \\
\text{NUMBER := any Java supported integer} \\
\text{TEXT := any non-whitespace characters}
\]

Figure V-3. Informal Grammar of MSCCheck’s Büchi Automaton XML Representation

Component implementations are represented by Büchi automata. MSCCheck expects an XML formatted document that follows the grammar given in Figure V-3. The document consists of the automaton’s initial states, acceptance states and a list of states that compose the automaton. Each state has a list of outgoing transitions associated with
it. An XML implementation representation of the C component, from the Central
Locking System example used in previous chapters, is given in Appendix A-1.

The Büchi Java representation is designed as a set of Transition objects, where a
transition is composed of two Büchi states (a source and destination) and a symbol. A
symbol can either be a lambda transition (no relevant event occurs) or a send or receive
message. Each state is designed as a set of active events or messages that can occur at
that particular state. In the case of implementation Büchi automaton, each state is
considered unique by what events can occur in that component at that particular point in
time.

V.B.2.b MSCs

Figure V-4. Informal Grammar of MSCCheck’s MSC XML Representation

System and property MSCs that are input into MSCCheck are expected to follow
the informal grammar given in Figure V-4. MSCs are represented using a recursive
definition of a construct if the MSC is a composite. Nevertheless, a composite should decompose into several Basic MSCs. Each Basic MSC is composed of several components, and every component is composed of several events (send and receive messages). The Locking MSC’s (Figure III-4) XML representation can be found in Figure V-5.

```xml
<MessageSequenceChart name="Locking">
  <construct type="infiniteLoop">
    <basic>
      <component name="KF">
        <event type="send">lck</event>
      </component>
      <component name="C">
        <event type="receive">lck</event>
        <event type="send">close</event>
        <event type="receive">close_ok</event>
      </component>
      <component name="LM">
        <event type="receive">close</event>
        <event type="send">close_ok</event>
      </component>
    </basic>
  </construct>
</MessageSequenceChart>
```

**Figure V-5. Locking MSC’s XML Representation**

The model representation of the MSC follows the composite design pattern. At the top-level is a `MessageSequenceChart` class that acts as a proxy layer above the abstraction of a composite MSC. A composite MSC is treated as a generalization of all
MSCs, where constructs such as alternative, sequence and infinite loop are defined. This composite pattern allows the flexibility to add any new constructs to MSCCheck with minimal effort. A class diagram representing the structure of MSCs in the model layer is given in Figure V-6.

**V.C Verification Activity**

![Figure V-7. Verification Activities](image)

The verification activity is composed of several activities with two that require focus. Firstly, MSCCheck needs to provide the user with a number of causal MSC candidates for a given property MSC. It iterates through the property automata and decides which causal MSCs describe the property best. If none are found, a causal MSC warning is created. If a causal MSC is found and decided upon by the user, MSCCheck performs a buffer product on the automata as defined the compositional verification algorithm in Chapter IV. The second main activity is the model checking step. MSCCheck must verify that the property holds true for the system. These activities are accomplished with the aid of a model checking tool, Bogor. MSCCheck translates the buffer product result and property into Bogor’s input format. Bogor finds any violations in the input file and reports them back to MSCCheck.
V.C.1 Finding a Causal MSC

In order for a property MSC to be utilized in verifying the system during verification activities, a list of replacement candidate causal MSCs needs to be offered to the user. To accomplish this, MSCCheck executes a causality check on those MSCs found in the pool of system MSCs that contain a subset of the messages found in the property MSC. Each model representation of every type of MSC is responsible for implementing its determination of causality. Generally stated, every composite MSC is considered causal if all of its composing basic MSCs are causal and the operations (construct) that are performed between them do not break that causality. For a more formal definition of causality checks one can refer to [FK07].

V.C.2 Verifying the System Using a Causal MSC

Once the user has a set of properties that he or she finds a corresponding causal system MSC for, MSCCheck can compositionally verify the system. This activity includes the sub-activities of decomposing the causal MSC, performing a buffer product, translating the result into an input for the model checker and synthesizing any defects found.

The most important sub-activity in regards to the architecture of MSCCheck is the model checking step. In order to understand the role of the model checker in regard to the verification algorithm and process, a brief description of the model checker is necessary.

V.C.2.a Bogor
Bogor [RDH03] (pronounced “bo-gore”) is a model checker plug-in for Eclipse that MSCCheck uses to perform verification of the implementation using an MSC specification. Bogor adopts the Eclipse idea of building a structurally modular framework and allowing the user to choose what functionality and information is needed to perform a specific task during the model check. Bogor is extensible in that it allows the user to use or build outside plug-ins for mechanisms such as state-space storage, state-space reduction and the state-space traversal strategy. The primary goal of Bogor is to give the user flexibility in choosing model checking strategies specific to their domain. This is a novelty in the area of model checking where traditional model checkers are known to have ad-hoc, non-customizable methods of performing model checking.

Bogor’s input language is the Bandera Intermediate Representation (BIR) (pronounced “beer”). The designers’ intentions were for a language that could support the translation of multiple abstractions of Java code, but still keep domain-specific knowledge of Java such as thread creation and heap state. In parallel, the BIR provides flexibility to abstract other languages such as CORBA and C# and has the capability of representing transitional systems such as state machines. Additionally, the BIR can be extended to provide additional expression and action constructs specific to the domain. This is accomplished by attaching new plug-ins to Bogor. The BIR follows a guarded command syntax similar to Promela where actions (commands) are taken dependent on the Boolean evaluation of expressions (guards).

The architecture of Bogor is separated into three layers: a Front-End, Interpretative Components and Model Checking Components. The Front-End layer is responsible for synthesizing and providing an in-memory representation of the model in
the BIR file. In similar fashion to a compiler, it contains a lexical analyzer component which feeds tokens to a parser that stores the BIR model in an Abstract Syntax Tree. The AST is passed through a well-formedness checker that builds a symbol table of the BIR model. The symbol table contains such information as indices for global variables and a mapping of thread names to their thread ids. In order to interpret the AST from the Front-End layer, Bogor’s architecture contains an Interpretative Components layer that is responsible for providing data structures that represent BIR states and values. This layer includes defining how a type’s operators affect the state of a model and how to transition to the next state in the model. The third layer is responsible for the model checking process and is composed of three important components. The searching component contains the algorithm necessary to explore the state-space of the BIR model. This component is dependent on two other important components that are responsible for transition strategy and backtracking from a trace.

While Bogor is intended to be an open system, where the user can create the algorithms to use for a particular model check, it is packaged with several ready-made algorithms to model check a state-space. The majority of extensions to Bogor employ a version of a depth-first search (DFS) of a system’s state-space. Bogor performs model checking on the BIR code by performing safety property checks in the form of invariants or assertions during the DFS exploration. For model checking specifications, Bogor has a number of extensions that are capable of checking the state-space by running properties in parallel with the DFS. These extensions include sets, Büchi automata, LTL and regular expressions.
Choosing Bogor over other model checkers such as SPIN and SMV was due to Bogor’s extensibility. We wanted MSCCheck to fit seamlessly into Eclipse, and Bogor provided a flexible framework in which to do so. It provided a method for easily modeling transition systems, while offering several methods to check properties on the model. The extension that MSCCheck exploits is the Büchi extension in order to capture defects and output a counterexample’s trace of the violation that could be easily parsed.

**V.C.2.b Performing a Buffer Product on the System**

Bogor provides several methods through its extensions to arrive at a cross product of two components or threads running in parallel. There is a channel mechanism similar to Promela channels in SPIN and a Set extension that could essentially be treated as channel. These extensions were experimented with as a means for generating a product, however, at the time of development these extensions were at beta stages and problems arrived when the product of an implementation and its environment would reach a non-deterministic choice. If a wrong choice was traversed in Bogor’s state-space there would be methods of ignoring the choice and backtracking to the choice point, but the checking property would still raise a violation creating a false positive. To overcome this obstacle, we decided to perform the buffer product entirely in the model layer where we had better control over decision points. The outcome could then be translated into Bogor’s transition system model and run in parallel with Bogor’s Büchi property checking extension.

The buffer product is implemented as a separate component whose inputs are an implementation automaton of the component to be verified and an environment automaton extracted from the causal system MSC. These two automaton run in parallel
with each other, ensuring that one automaton only send a message that is common between them when the other is in a state where it is expecting to receive it. Once the buffer product result is created, it is translated into Bogor’s input format, BIR, where it can be model checked against the property represented by the global Büchi automaton of the causal MSC. This is accomplished through Bogor’s Büchi property extension.

**V.C.2.c Büchi Property Extension and BIR Input**

The Büchi property extension for Bogor allows model checking of a BIR specification through product checking of finite state automata. The Büchi property automaton runs in parallel with the buffer product automaton during a state-space exploration. Using a Nested-Depth First Search (NDFS) of the buffer product automaton and augmenting the property automaton to the current state allows Bogor to perform Büchi checking. The NDFS will look at the label of the current state of the Büchi automaton. If it is marked as a non-accepting state, it will recursively call the NDFS method to check if there is a trace that revisits this state. If it arrives back at this state, then a counterexample is produced by Bogor.

The Java Emitter Templates (JET) tool [ECL07b] was used to create the BIR code for input into Bogor. JET is a plug-in that “generates code that generates code.” A user can write a script similar to Java Server Pages (JSP) format and generate Java code that can generate any programming language the user desires. Using the wrapper design pattern, the buffer product automaton and the property automaton are translated to their respective BIR transitional model equivalent by processing it through the JET script.
function FSASpec() {
    loc start0: do {} goto init;

    loc endstate:
        when (sysend == false &&
        (event == "!lck" || event == "!close_ok" ||
         event == "?close_ok" || event == "!close" ||
         event == "?close" ||
         event == "?lck" || false ))
        do {} goto bad$trap;

    loc bad$trap:
        when true do { event := ";" } goto bad$trap;

    loc init:
        when event == "!lck" do { event := ";" } goto loc6;
        /* Bad Transitions */
        when (event == "!close_ok" ||
             event == "?close_ok" ||
             event == "!close" ||
             event == "?close" ||
             event == "?lck" ||
             sysend || livelock)
        do { event := "" } goto bad$trap;

    loc loc0:
        when event == "!close" do { event := "" } goto loc2;
        /* Bad Transitions */
        ...

    loc loc6:
        when event == "?lck" do { event := "" } goto loc0;
        /* Bad Transitions */
        ...
}

Figure V-8. Sample Property Automaton for Locking MSC Displaying Locations and Guarded Commands

A thread and function are generated in the BIR code. The thread, called the SYSTEM thread, is representative of the buffer product automaton and the function represents the causal MSC that it must be contained in. Each state in the generated property function is mapped to a BIR “location,” and each transition in the automaton is treated as a BIR guarded command. A location will only transition to another location when the guard is true. If the guard condition returns true, then it is true that the SYSTEM thread made a transition that matched the guard condition that the property automaton was waiting on. To capture any safety violations a special trap state is introduced into the property function. If the SYSTEM thread performs a transition that is out of sequence with the property function, then it will be caught in a trap state and a safety violation will be
generated. Otherwise the transition is “recognized” (i.e. it has a transition in proper progressive order defined in the automaton) to the property automaton and the property function and system thread will progress to their next progressive state. A sample containing the first transitions of the Locking MSC in BIR Büchi syntax is given in Figure V-8.

The guard conditions in the property function are activated by a variable shared between the system thread and property thread called event. The variable is updated by the system thread to represent a transition in the system automaton. The property function reads this variable to satisfy its guarded conditions. Another important variable is the sysend variable. This variable signifies that the system thread has terminated unexpectedly and will move the property function into a trap state. Other variables are named after the components in the system thread to initialize the property thread in the right start state (based on its active events).

The buffer product automaton is translated to the system thread portion of the BIR code. Similar to the property function, its states are translated to BIR locations and transitions follow guarded commands. However, since all transitions are essentially active, no guards are necessary and all commands are issued upon arrival to a location. Figure V-9 shows the BIR representation of the buffer product of the C component and its environment from the Locking MSC (Figure IV-13). Following the events leading from loc2 to loc1 puts the system in a deadlock. The Büchi property function running in parallel with the buffer product will recognize the deadlock and send itself into the trap state, thus instructing Bogor to generate a defect.
Main thread `SYSTEM()` {
  loc init: live {
    do {
      LM := "?close";
      C := "?lck";
      KF := "!lck";
    } goto loc0;
  }
  loc end$state: 
    do { sysend:=true; event := ""; } goto end$state;
  loc loc0: live {
    do {
      event := "!lck";
      statement := 0;
    } goto loc5;
  }
  loc loc1: live {
    do ( sysend := true; ) goto end$state;
  }
  loc loc2: live {
    do {
      event := "!close_ok";
      statement := 1;
    } goto loc1;
  }
  loc loc3: live {
    do {
      event := "!close";
      statement := 2;
    } goto loc4;
  }
  loc loc5: live {
    do {
      event := "?lck";
      statement := 3;
    } goto loc3;
  }
  loc loc4: live {
    do {
      event := "?close";
      statement := 4;
    } goto loc2;
  }
}

Figure V-9. Generated Specification Automaton for Locking MSC Composed with the C Component
**V.C.2.d Bogor Configuration File for MSCCheck**

```java
edu.ksu.cis.projects.bogor.module.IActionTaker=
  edu.ksu.cis.projects.bogor.module.DefaultActionTaker
edu.ksu.cis.projects.bogor.module.ICounterExampleWriter=
  edu.ucsd.sosa.MSCCheck.utilities.verifier.MSCCheckCounterExampleWriter
edu.ksu.cis.projects.bogor.module.IExpEvaluator=
  edu.ksu.cis.projects.bogor.module.DefaultExpEvaluator
edu.ksu.cis.projects.bogor.module.ISchedulingStrategist=
  edu.ksu.cis.projects.bogor.module.DefaultSchedulingStrategist
edu.ksu.cis.projects.bogor.module.ISearcher=
  edu.ksu.cis.projects.bogor.module.property.buechi.NestedFSASearcher
edu.ksu.cis.projects.bogor.module.IStateManager=
  edu.ksu.cis.projects.bogor.module.DefaultStateManager
edu.ksu.cis.projects.bogor.module.IStateFactory.stateAugmenters=
  edu.ksu.cis.projects.bogor.module.property.fsa.FSAStateAugmenter
edu.ksu.cis.projects.bogor.module.IValueFactory=
  edu.ksu.cis.projects.bogor.module.value.DefaultValueFactory
fsaFunctionId=FSASpec
edu.ksu.cis.projects.bogor.module.ISearcher.maxErrors=3
```

**Figure V-10. Updated Configuration File for MSCCheck**

Bogor uses a configuration file that contains parameters that point to the various plug-in extensions that contain the implementation of the functionality Bogor should perform when executing a model check. By default, Bogor connects to its standard extensions. Therefore, some parameters need to be updated and added to point to the Bogor Büchi Property and MSCCheckCounterExampleWriter extension plug-ins. Figure V-10 shows the updated and additional parameters that are required to run Bogor correctly with MSCCheck. This file is stored in the edu.ucsd.sosa.MSCCheck.utilities.verifier package and is loaded at runtime, thus the user need not attend to it. Figure V-11 displays how MSCCheck calls Bogor to initiate the verification process for a specific BIR file.
ByteArrayOutputStream bogorOutput = new ByteArrayOutputStream();
PrintWriter pw = new PrintWriter(bogorOutput);
IProgressManager pm = new AbstractProgressManager();
Properties configuration = new Properties();
configuration.load(MSCVerifier.class.getResourceAsStream(CONFIG_NAME));
configuration.put(ERROR_KEY, Integer.toString(maxErrors));
Bogor.modelCheck(birFilename, pw, pm, new DefaultBogorConfigurationUI(), configuration);

Figure V-11. Call to Bogor to model check the BIR code

V.C.2.e Writing a Counterexample Defect

The current counterexample writing mechanism in Bogor creates a symbolic trail of the counterexamples from the BIR file that was model checked and contained the defect. This counterexample file is specifically meant to be read by Bogor to simulate the user through the original BIR code. The transitional information was reduced to symbolic information. Since our intention was to make the BIR specification language transparent to the user and provide counterexamples in MSC format, the counterexample writing mechanism needed a new counterexample writing plug-in.

Figure V-12 shows the code for the writeCounterExamples() function in the MSCCheckCounterExampleWriter extension. To capture the messages and components that label a transition, two variables are set that store the message labels in the SYSTEM() thread in the BIR file. When a counterexample is found the symbolic error trace information that Bogor usually collects for storage by default is simulated over the model of the system instead. At every step of the simulation a snapshot of the state is taken. The state information is then queried to extract the value of one of the message variables. This process is continued until the end of the error trace is reached. Each value of the message variable is then stored in a vector that is passed to MSCCheck for conversion to a basic MSC.
for (int i = 0; i < errorCount; i++) {
    currErrorSchedules = errorSchedulesList.get(i);
    IBogorConfiguration sim = getGuidedSimulator(symbolTable,
        getCounterExampleGenerationConfiguration(),
        this.bc.getClass(),
        currErrorSchedules,
        currErrorSchedules.size(),
        false).getBogorConfiguration();

    ISearcher searcher = sim.getSearcher();
    searcher.initialize();
    searcher.step();
    int[] threadids = searcher.getState().getThreadIds();
    int systemtid = threadids[0];
    ObjectIntTable<String> gblidxtbl = symbolTable.getGlobalIndexTable();
    int msgidx = gblidxtbl.get("event");
    List<Msg> ctrxtrace = new ArrayList<Msg>();
    for (int j = 0; j < currErrorSchedules.size(); j++) {
        IState state = searcher.getState();
        String msgstr = state.getGlobalValue(msgidx).toString();
        if (msgstr.length() > 0) {
            Msg m = new Msg(msgstr);
            ctrxtrace.add(m);
        }
        searcher.step();
    }
    sim.dispose();
    sim = null;
    System.gc();
    traces.add(convertTraceToBMSC(ctrxtrace, commons, lambdas));
}

Figure V-12. Implementation of the writeCounterExamples Method in the
MSCCheckCounterExampleWriter Bogor Extension

V.D Counterexample Analysis Portion

The counterexample analysis portion exploits two of Eclipse’s extension
mechanisms: views and editors. There were also several plug-ins used from Eclipse’s
Tools Project [ECL07b] that aided in developing and deploying our tool.

V.D.1 Defect Format

When a defect is found, a counterexample file is produced. This file contains
information regarding the following:

- type of defect (syntax or behavioral)
- name and trace of the MSC that was used in the compositional verification
- name of the component in the implementation that was found to be defective (if the violation was not caused by the specification MSC)
- name and trace of the violated property
- a trace of the counterexample

```xml
<defect>
  <type>MSC</type>
  <mscUsed>UNLCK-1</mscUsed>
  <defectComponent>LM</defectComponent>
  <property>UnlockingProperty</property>
  <mscTrace>
    <event>KF:!unlck</event>
    <event>C:?unlck</event>
    <event>LM?:open</event>
    <event>LM!:open_ok</event>
    <event>C:?open_ok</event>
  </mscTrace>
  <propertyTrace>
    <event>KF:!unlck</event>
    <event>C:?unlck</event>
    <event>LM!:open_ok</event>
    <event>C:?open_ok</event>
  </propertyTrace>
  <counterexampleTrace>
    <event>KF:!unlck</event>
    <event>C?:unlck</event>
    <event>LM!:open</event>
    <event>LM!:open_ok</event>
    <event>C!:handle_ID</event>
    <event>LM!:open_ok</event>
  </counterexampleTrace>
</defect>
```

**Figure V-13. MSCCheck Defect for UnlockingProperty**

**Figure V-14. MSCCheck Defect Format**

Figure V-13 gives an example of the contents of an MSCCheck defect file taken from the counterexample produced in Figure III-13. Figure V-14 gives the typical structure of an MSCCheck defect file.

V.D.2 MSC Rendering Library
The Graphical Editing Framework (GEF) [ECL07b], a subproject of Eclipse’s Tools Project, is composed of tools for building and editing drawings. It separates diagrams through use of the Model-View-Controller design pattern. In GEF’s three-layered architecture, the GEF toolset provides functionality for the control layer. The view layer is rendered by a Standard Widget Toolkit (SWT)-based plug-in known as Draw2D. This plug-in is used to render MSCCheck’s MSCs in Eclipse.

The edu.ucsd.sosa.MSCCheck.utilities.view.sd package contains an simple library that we developed to diagram sequence interactions between instances. We chose to develop a new rendering package because it gives us the flexibility of controlling the extended notation of MSCs or any customizations to the rendering. In addition, if need be, the package can be extended to have a GEF control interface. Therefore, future considerations of MSCCheck could be integrated into Eclipse in full, and the reliance on outside tools for modeling of MSCs is eliminated.

V.D.3 Viewing a Counterexample

When a counterexample is created it is stored in the “defects” directory of the Eclipse project the MSCCheck configuration file is stored in. The DefectView extends the ViewPart package of Eclipse. It shows a table outline of the newly created defects. The CounterExampleEditor for viewing the defect files extends the MultiPageEditorPart class of Eclipse editors. The DefectView calls this editor with the code shown in Figure V-15. Once the CounterExampleEditor is loaded, the MSC rendering library is used on each of the three traces in the defect file.
V.E Limitations

In creating a tool built around a framework for verifying and generating MSCs, we were limited by the expressiveness of our inputs and the completeness of the compositional verification algorithm. Unfortunately these limitations place a bound on the effectiveness of MSCCheck. However, these limitations should be possible to emend as MSCCheck and the underlying fundamentals mature.

Causality

As stated in Chapter II, the specification is limited to causal MSCs due to the necessary decomposition property exploited by the compositional verification algorithm.

Incompleteness

The authors of [FK07] state that the compositional verification algorithm is an incomplete proof. There exist systems that are correct where a property cannot be checked with a MSC specification. The authors give an example where an MSC specifies two components having three alternative behaviors. Either a single message may be sent from component A to component B, component B may send a message to component A, or they both can send their messages at the same time. If the implementation is such that the components implement a different scheduling mechanism for these messages, one component may be implemented such that it does not satisfy one alternative. This is due to the same message labels being reused in the third scenario where both messages are sent simultaneously.
Globally Unique Messages

Every message in the system is a unique interaction between two components. One message is meant to be sent by one component and received by one component. This eliminates the possibility of broadcast messages where one component sends a message and multiple components receive that message. This also eliminates the possibility of multiple instances of the same component in the system.

Triggers

Triggered messages are left out of the verification process. This class of messages is indeed necessary for describing situations where interrupting the system is critical such as an alarm that immediately invokes a service. They are also essential for nominal services such as terminating a service that is currently in progress.

Nature of MSC Properties

When verifying properties within MSCCheck they are considered as “should always occur” in the system as opposed to “if this occurs then this should occur.” For example, if there was a property MSC that had messages m1, m2 and m3 in an ordered sequence, then this sequence should always occur in the system. Every message in the property is seen as causal. It defines the sequence as m1 caused m2 which caused m3. However, if we wanted to define the property as “if m1 is followed by m2, then m3 should occur,” then this is infeasible in MSCCheck.

Cross-platform Compatibility

The last minor issue is that MSCCheck was designed for exclusive use on Windows machine configurations despite its use of Eclipse and Java.
Chapter VI
Case Study: Center TRACON Automated System

VI.A Background

In order to explore and demonstrate the scalability of verifying system properties with MSCCheck, we evaluate a larger and more complex system. The Center TRACON Automated System (CTAS) [SCS03] is composed of a number of tools designed to help air traffic controllers manage the increasingly complex air traffic at large airports. The project began in 1991 and prototypes are now deployed at Denver and Dallas/Fort Worth airports. Studies of CTAS have been made by researchers over several years and still continue today in the areas of scenario-based modeling and state-machine synthesis [DAG03].

CTAS is a distributed system that communicates using sockets. Using a requirements document [SCS03b] for CTAS that describes processes and the interactions between them we can model a version of the CTAS. The two processes (components) described in the requirements document are a Communications Manager (CM), weather-aware clients (AwareClient), weather-unaware clients, RAPS clients and PGUI clients. The CM is centralized and controls all weather related interactions between all weather-aware clients. A weather-aware client requires weather updates from the CM for internal processing. RAPS clients and PGUI clients are both weather-aware clients. The key difference is that RAPS clients contain lower level components that require weather-
related information and may request the CM to reinitialize. Weather unaware clients do not rely on weather related information from the CM.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Rationale</th>
<th>Level of Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Assumptions are covered in the interactions.</td>
<td>Full</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Source of forecast definition, no interactions defined.</td>
<td>None</td>
</tr>
<tr>
<td>2.2</td>
<td>Mode types definition, no interactions defined.</td>
<td>None</td>
</tr>
<tr>
<td>2.3</td>
<td>Weather calculations, no interactions defined.</td>
<td>None</td>
</tr>
<tr>
<td>2.4</td>
<td>_statuses for the weather cycles captured in messages.</td>
<td>Partial</td>
</tr>
<tr>
<td>2.5</td>
<td>_statuses for the weather clients captured in messages.</td>
<td>Partial</td>
</tr>
<tr>
<td>2.6</td>
<td>Interactions for the pre-initialize phase are covered with extra messages to satisfy causality.</td>
<td>Full</td>
</tr>
<tr>
<td>2.7</td>
<td>Interactions for re-initializing are covered with extra messages to satisfy causality.</td>
<td>Full</td>
</tr>
<tr>
<td>2.8.3-2.8.6</td>
<td>Interactions for initializing are covered with extra messages to satisfy causality.</td>
<td>Full</td>
</tr>
<tr>
<td>2.8.7,2.8.9</td>
<td>Interactions for post-initializing are covered with extra messages to satisfy causality.</td>
<td>Full</td>
</tr>
<tr>
<td>2.8.1,2.8.2.2.9</td>
<td>Interactions for weather update checking are covered with extra messages to satisfy causality.</td>
<td>Full</td>
</tr>
<tr>
<td>2.8.10</td>
<td>Interactions for pre-updating are covered with extra messages to satisfy causality.</td>
<td>Full</td>
</tr>
<tr>
<td>2.8.11-2.8.13</td>
<td>Interactions for updating are covered with extra messages to satisfy causality.</td>
<td>Full</td>
</tr>
<tr>
<td>2.8.14-2.8.16</td>
<td>Interactions for post-updating are covered with extra messages to satisfy causality.</td>
<td>Full</td>
</tr>
<tr>
<td>2.8.17-2.8.19</td>
<td>Interactions for post-reverting are covered with extra messages to satisfy causality.</td>
<td>Full</td>
</tr>
<tr>
<td>2.10</td>
<td>Specific to the CM only and no interactions.</td>
<td>None</td>
</tr>
<tr>
<td>2.11</td>
<td>Details about the weather panel and no interactions.</td>
<td>None</td>
</tr>
<tr>
<td>2.12-2.13</td>
<td>Interactions for AwareClients with extra messages to satisfy causality.</td>
<td>Full</td>
</tr>
</tbody>
</table>

There are several requirements for states and modes which this case study version will not cover. However, we are primarily interested in the requirements that capture the
interactions between components. If a state is required to distinguish an interaction the state name is appended to the message name. These requirements will help in demonstrating the effectiveness of MSCCheck. A table of requirements coverage is given in Table VI-1. Note that the original system is intended for multiple clients receiving updated weather information broadcasted from the Communications Manager. We tailored this requirement for MSCCheck which does not support the notion of broadcasting messages and limited the interactions between the Communications Manager and one weather-aware client.

VI.B System and Test Setup

Based on the requirements document coverage, several services were specified and two component implementations developed.

VI.B.1 System Model

The system model is defined as causal system MSCs that support the two base services of the system, Client Initialization and Update Weather.

VI.B.1.a Connect

The connect service specifies the interaction that must occur for a weather aware client, such as the RAPS or PGUI clients to connect to the Communications Manager. The client sends a CONNECT message to the manager and this triggers the Initialize service. Figure VI-1 displays the Connect System MSC. The “REF” construct is a reference operation. In this case, the Initialize System MSC is referenced.
VI.B.1.b Initialize

The initialize service specifies the interaction that must occur when a weather aware client wishes to connect successfully to the communications manager or when a weather aware client such as the PGUI client needs to reinitialize its subcomponents successfully. The sequence begins with a pre-initialization step, an update of the client’s status to initializing and an attempt for the weather-aware client to get new weather information from the communication manager. If the information is unsuccessfully handled on the client’s end, the client notifies the manager of this and the manager disconnects the client. If the information is successfully handled by the client, the client goes through post-initialization and is instructed to use the new weather. If the use weather instruction fails, the client is disconnected by the communication manager. If the use weather instruction succeeds, then the manager relays other weather related data to the client. Figure VI-2 displays the MSC that describes the Initialization service.

VI.B.1.c Reinitialize

Re-initialization occurs when the PGUI client attempts to restart its subcomponents. It is triggered by CTAS_WTHR_REINITIALIZED message coming from the weather aware client to the communications manager. The initialization service
occurs immediately following the receipt of this message by the communication manager.

The Reinitialize System MSC is given in Figure VI-3.

Figure VI-2. Initialize System MSC

Figure VI-3. Reinitialize System MSC
VI.B.1.d Update

The update service is the second prime feature of CTAS. It involves similar contained in the Initialize service except that clients are already connected to the communication manager and they need to be updated with new weather information. The interaction begins with the manager switching to a pre-updating state and notifying all connected clients that a weather update is available. The manager then attempts to provide the clients with new weather information. If the update fails, the clients are instructed to use the weather information that it was originally functioning with. The clients are then instructed to use either the old or new weather information. If the use weather instruction fails on the client side, it is instructed to disconnect from the manager. If the use weather instruction succeeds on the client side, it continues to update with other weather related information. Figure VI-4 displays the Update System MSC that describes this service.
VI.B.1.e CTAS

The CTAS service describes the entire CTAS system in terms of how the services should occur sequentially. The first service that should occur is the connect service. If the connect service is successful, the client can either receive a weather update or reinitialize its subcomponents. Due to the lack of expressiveness in the MSCs that MSCCheck recognizes and to maintain causality in the system MSC, one of the requirements had to be altered so that an aware client could only perform the connect service whether the update or reinitialize service fails or not. Figure VI-5 displays our limitation in an infinite loop view of the CTAS system MSC.
VI.B.2 Implementation

Based on the states described in the requirements document and the system MSCs described in the previous section, an implementation for both the communications manager and weather-aware client were developed. The Manager and AwareClient states of pre-initializing, initializing, post-initializing, done, pre-updating, post-updating and
post-reverting are all captured in the composition of several states of the implementations. Figures VI-6 and VI-7 display the implementations of the Manager and AwareClient respectively.
Figure VI-6. Communications Manager Implementation Automaton
Figure VI-7. Weather-Aware Client Implementation Automaton
VI.B.3 Experiment Setup and Results

To test the impact of MSCCheck, we performed verification on our version of CTAS manually and compared the times and results with performing the verification using MSCCheck. We used the CTAS System MSC as our property MSC and causal system MSC for our experiment. Since this MSC covers all intended properties of the system, it verifies that all requirements covered under the MSC specification will be covered. In addition, it yields a complex property that requires several automata translations and is non-trivial in verifying manually.

For our two methods of verifying CTAS, we separate results into two steps: Cross Product Synthesis and Verification. The Cross Product Synthesis step involves two sub-steps. The first step is translation of the CTAS system MSC into an implementation component automaton environment. Taking that environment and performing a buffer cross product is our second step. Note that the Cross Product Synthesis step does not include proving that CTAS is a causal MSC which MSCCheck has already done for us. By manual method, this involves us taking a print out of the CTAS system MSC, reviewing the translation algorithms in [FK07] for each MSC construct and sequencing each resulting “sub-automaton” from the translation to create the environment automaton on paper. Using a print out of the component implementation, we then perform a buffer product with that environment automaton on paper. This step also includes translating the CTAS system MSC into its global automaton equivalent in preparation for the second step of the verification algorithm. For the Cross Product Synthesis step, MSCCheck loads the implementation automata and CTAS system MSC into memory, performs a
buffer cross product and writes the output to a file in BIR format. The second step that we measure is the Verification step which includes comparing the resulting cross product automaton with the global automaton of the CTAS System MSC. Manually, this process involves visiting each state in each of the handcrafted automaton and ensuring that the cross product automaton does not create any safety or liveness violations. If there is a violation the state is marked and a trace up to that state is recorded on paper in the form of state numbers. MSCCheck calls the Bogor API on the generated BIR files from step one, and writes a counterexample file for each component under verification for this step.

Table VI-2. Result Times of Methods of Verification of CTAS

<table>
<thead>
<tr>
<th>Method</th>
<th>Cross Product Synthesis</th>
<th>Verification (&lt; 4 errors)</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual</td>
<td>3 hrs 15 mins</td>
<td>50 mins</td>
<td>4 hrs 05 mins</td>
</tr>
<tr>
<td>MSCCheck</td>
<td>1.093 secs</td>
<td>13.282 secs</td>
<td>14.375 secs</td>
</tr>
</tbody>
</table>

Each verification method’s time is given in Table VI-2. The global automaton for the CTAS System MSC resulted in 63 states. After a cross product of the environment automaton (32 states) with the Manager component (28 states), buffer cross product automaton produced 68 states. The AwareClient produced the same total state numbers for the implementation and environment and a total of 72 states for the buffer cross product automaton. Manual verification of the system produced 0 counterexamples for the AwareClient and 2 for the Manager. MSCCheck was profiled using Java’s `System.currentTimeMillis()` method call. The times captured in Table VI-2 are an average of 5 runs of the system, which ranged between 1 and 1.24 seconds for the Cross Product Synthesis step and 12 to 15 seconds for the Verification step. Verification not only successfully completed more efficiently than the manual, but also discovered 2
deadlocks deep in the AwareClient’s implementation. Specifically, when an AwareClient updated successfully, its environment would be caught in a waiting state, waiting for a new connection from an outside client. This is purely a specification defect. A client is not required to connect again after an update. From the Manager component’s perspective, an AwareClient can connect in this state, but is not required to. The CTAS System MSC needs to be refined to address this problem. This defect was unrealized during the manual method’s verification step due to user error.

While MSCCheck was better at finding defects more effectively and efficiently, there were limitations in use of the tool. Discovering the cause of the deadlocks for the Manager was trivial since the counterexample trace was simple. One could easily follow the trace alongside the implementation automaton and pinpoint exactly where and why the defect was produced. For a longer trace such as those produced by the AwareClient, however, discovery of the root cause of the defect was not as trivial. We observed that for this specific case it was beneficial to review the manual verification method’s outputs. Following the cross product algorithm in lock-step and following the states of the environment and implementation automata is important in locating the exact messages that they deadlocked on.
Chapter VII
Comparison with Related Work

VII.A IIL/Turing Plus and ViP

Several projects similar to MSCCheck can be found at Queen’s University [DL04]. The authors recognize the semantic gap between the artifacts used during verification and those used during validation. To alleviate this problem, they developed a framework around implicit invocation (publish-subscribe) systems using the Implicit Invocation Language (IIL) as their formalism. Temporal properties can be defined within this specification. The IIL undergoes a number of transformations for verification and validation. It is first translated into an XML representation and then into a finite state machine to be model checked against the temporal properties using Cadence SMV. Then, to provide for simulation and validation, the IIL is translated into a modified version of the executable language Turing Plus. These transformations are accomplished through a source to source transformational tool called TXL. This framework’s approach is similar in using a modified version of an implementation of a message-based system, and using a formalism to verify a set of properties of that system. The approach differs in that during the verification step the implemented system is taken as a whole, rather than decomposed.

Other researchers within the same team are experimenting with a tool called ViP [ZBC04] that takes C\C++ code and temporal logic and translates them into Verisoft-compatible [BG97] C code. It then uses Verisoft to model-check run-time traces of the
system. The ViP provides the same kind of high-level component/service analysis as with MSCCheck, and uses Verisoft for counter-example analysis during verification failure. While the ViP looks very promising, the counterexample generation and analysis is presented to the user in terms of the compiled source code. MSCCheck presents counterexamples in the original specification’s notation. ViP is also restricted to C/C++ code, whereas MSCCheck’s implementation automaton is flexible for any language.

VII.B ObjectCheck

In the area of model-checking in component-based systems, researchers at the University of Texas at Austin have been working on the verification of systems using xUML specifications using a compositional reasoning approach [XB03]. Specifically, a component is verified if its subcomponents are verified based on environmental assumptions. xUML specifications are translated into an automaton language S/R. They are then verified using the COSPAN model checker. Using the ObjectCheck tool [XLB02], the user can visually step through the xUML specification when a counterexample is produced.

ObjectCheck and MSCCheck’s approaches intersect on compositional reasoning about a component’s environment to verify the entire system. The major deviation of the verification approach used by ObjectCheck and MSCCheck is that ObjectCheck’s verification is performed at the component-level. Our approach allows for properties to be provided at the service-level, allowing for a better overall view of the system’s functionality rather than single interactions between components.

VII.C JPaX and JMPaX
In the area of run-time verification, the Java PathExplorer (JPaX) project [HR01] is a run-time verification tool that monitors execution of a Java application and checks it against user-defined temporal logic specifications. Monitoring is achieved by instrumentation using their home developed mechanism JSpy [GH03]. JPaX is used to verify two facets of the running application, temporal logic properties and concurrency. The former is accomplished through the use of a future and past time logic written in the Maude rewriting system. In addition, equipped with a set of concurrency analysis algorithms, JPaX is able to locate race conditions and deadlocks from an arbitrary execution trace.

JPaX has been extended several times. Researchers at University of Illinois – Urbana are focusing on one such extension known as Java Multipath Explorer (JMPaX). JMPaX [SRA03] is able to predict possible safety violations based on one execution of the system regardless of the production of a counterexample. It accomplishes this by exploiting causal dependencies similar to MSCCheck.

The work done on JPaX and JMPaX is similar to ours in using a formal specification to catch safety and liveness violations in an implementation. Both JPaX and JMPaX are powerful tools, but the specification is written in past time temporal logic which is usually hard to understand visually at design time and during a counterexample trace.

VII.D JAVA-Mac

Another run-time verifier similar to MSCCheck is the Java-Mac Project [KKL01] by researchers at UPENN provides run-time verification of Java applications based on the ability to extract low-level implementation events and map them to high-level...
requirements in a specification. These specifications are separated as two different scripts for low-level and high-level events, PEDL and MEDL respectively. The architecture follows closely to that of JPaX. An event stream is monitored by the event-recognizer, and low-level events are assembled according to the MEDL script and sent as high-level events to the run-time checker. Verification takes place inside the run-time checker by responding to any properties the high-level events may steer away from. A newer version of the Java-Mac system responds to these alarms by bringing the system back into a safe state. This is similar to MSCCheck monitoring a static state space for event-based/service-based property violations. However, Java-Mac performs an exhaustive search of the run-time state space and does not abstract an implementation model as MSCCheck requires.

Java-Mac provides variable monitoring and condition checks as part of the properties of the system to be verified which is very valuable. However, Java-Mac uses an ad-hoc specification language in its PEDL and MEDL scripts and only checks those events that were defined in these specifications. This limits Java-Mac in that it cannot recognize any unexpected run-time behaviors.

**VII.E ASML**

Abstract State Machine Language (ASML) [BS01] [BNS01] is a project by Microsoft Research that takes an abstract view of the system written as a specification in ASML and runs this model in parallel with the running system. At runtime a proxy is set up between a client component and its server component and checks are done at the interface level within the proxy. ASML is very scalable in that it can be applied to virtually any programming language for checking model conformance of an
implementation similar to MSCCheck. Use of ASML is most similar to MSCCheck when using it to check against scenario-oriented models. Both tools take an abstracted implementation in the form of a finite state machine and perform property checking on it. ASML allows the user to specify which traces are of interest. Therefore, if the user is interested in checking if the state space contains behaviors that should not occur he or she can specify it as a property. The resulting scenario will be generated as a finite state machine. This is demonstrated in [BGG03] using the CTAS case study. ASML differentiates from MSCCheck in that it is not entirely service-based, and provides support for data assertions. It also uses a customized specification language to define properties.
Chapter VIII
Concluding Synopsis and Future Directions

As the need for more complex systems rises, software complexity to control and support these systems increases. In service-oriented software systems the complexity of the system exists at the service-level. New services and features are added to the system to support the evolving requirements of the system. This introduces new, unexpected and expensive defects into systems that are often left uncaught. In addition, locating such defects is difficult and expensive without the proper tool support and methodology.

We have created a tool that supports the discovery of defects in service-oriented software systems using service-based semantics. The tool fully utilizes MSCs to verify that an implemented system satisfies its specification. Additionally, the tool uses a compositional verification approach to efficiently verify the system. The user can specify what liveness or safety properties he or she wishes to check in the implemented system in the form of MSCs as a specification. By utilizing the compositional verification approach and a state-of-the-art model checker, the tool checks that the specified services in the implemented system are correct, and if not, offers the user a trace leading up to the discovered defect in a MSC format. This aids the user in discerning the cause and eventual resolution of the defect.

The CTAS case study demonstrates the feasibility and scalability of using the tool to verify a complex, service-oriented software system. It also demonstrates the benefits of using an automated approach over manually verifying the system. We were able to
verify the implemented system using a complete specification as a property to check the system. Experimentation with the case study also yielded some of the limitations of the tool. While the counterexample reports generated by the tool are comprehensive, counterexample analysis of the reports leaves more to be explained. Furthermore, using small properties without the supporting causal properties as a property check is not possible. Despite these limitations, we were able to achieve our goal of quickly and inexpensively locating defects in the system using a service-based specification.

VIII.A Future Directions

The tool we created uses MSCs to verify a state-based implementation. MSCs have a rich and diverse language that the tool does not support currently. This language set includes High-level MSCs, parallelism, join and preemption. These are all essential constructs that will augment the expressiveness of the specification and allow for a larger set of properties to be checked.

In addition to supporting other constructs, further research must be done in definition of a supporting toolset. MSC design can be accomplished using a tool such as M2Code [Moo06] and extra functionality must be added to support the design of system properties and conversion of the MSCs in an MSCCheck compatible format. Furthermore, a tool to support the creation of the implementation automata is necessary. Abstraction of an implemented system has been accomplished by tools such as Bandera [DHJ01] and Verisoft [BG97]. Utilizing this feature of these two tools, and tailoring the feature for our use would yield the implementation model that we desire.

Another future direction is in the area of counterexample analysis. Currently, MSCCheck will create a counterexample trace and it is left to the user to determine the
cause. Providing the user with a more comprehensive counterexample in the form of a service-composed approximation of the trace would be of benefit to user. This would allow them to compare the trace to the specification and locate more quickly where in the specification the defect occurred, and determine if the defect exists in the specification or in the implemented component.
Appendix A-1
XML Representation of C Büchi Implementation Component

```xml
<Buchi>
  <initialstates>
    <id>1</id>
  </initialstates>
  <acceptstates>
    <id>1</id>
  </acceptstates>
  <transitions>
    <state id="1">
      <transition>
        <symbol>?unlck</symbol>
        <dest>5</dest>
      </transition>
      <transition>
        <symbol>?lck</symbol>
        <dest>3</dest>
      </transition>
    </state>
    <state id="5">
      <transition>
        <symbol>!open</symbol>
        <dest>2</dest>
      </transition>
    </state>
    <state id="4">
      <transition>
        <symbol>?ID_OK</symbol>
        <dest>0</dest>
      </transition>
    </state>
    <state id="2">
      <transition>
        <symbol>!handleID</symbol>
        <dest>4</dest>
      </transition>
    </state>
    <state id="0">
      <transition>
        <symbol>?open_ok</symbol>
        <dest>1</dest>
      </transition>
    </state>
    <state id="3">
      <transition>
        <symbol>!close</symbol>
        <dest>1</dest>
      </transition>
    </state>
  </transitions>
</Buchi>
```
Bibliography


