INCREMENTAL VALIDATION OF FORMAL SPECIFICATIONS

A Thesis

Presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Paul Corwin

May 2009

COMMITTEE MEMBERSHIP

TITLE:	Incremental Validation of Formal Specifications
AUTHOR:	Paul Corwin
DATE SUBMITTED:	May 2009
COMMITTEE CHAIR:	Gene Fisher, Ph.D.
COMMITTEE MEMBER:	David Janzen, Ph.D.
COMMITTEE MEMBER:	Clark Turner, J.D., Ph.D.

Abstract

Incremental Validation of Formal Specifications

by

Paul Corwin

This thesis presents a tool for the mechanical validation of formal software specifications. The tool is based on a novel approach to incremental validation. In this approach, small-scale aspects of a specification are validated, as part of the stepwise refinement of a formal model.

The incremental validation technique can be considered a form of "lightweight" model checking. This is in contrast to a "heavyweight" approach, wherein an entire large-scale model is validated en masse.

The validation tool is part of a formal modeling and specification language (FMSL), used in software engineering instruction. A lightweight, incremental approach to validation is beneficial in this context. Such an approach can be used to elucidate specification concepts in a step-by-step manner. A heavy-weight approach to model checking is more difficult to use in this way.

The FMSL model checker has itself been validated by evaluating portions of a medium-scale specification example. The example has been used in software engineering courses for a number of years, but has heretofore been validated only by human inspection. Evidence for the utility of the validation tool is provided by its performance during the example validation. In particular, use of the tool led to the discovery of a specification flaw that had gone undiscovered by manual validation alone.

Acknowledgements

Thanks very much to Dr. Gene Fisher for giving me guidance and support above anything I ever could have expected and for handing off to me C code that not only worked well but also was entertaining to digest. Thanks to Dr. Clark Turner for sharing with me his passion for software engineering and for serving on my committee. Thanks to Dr. David Janzen for serving on my committee and for being flexible as my defense approached.

Contents

List of Tables

List of Figures

Chapter 1 Introduction

Software engineering is an error-prone and expensive process. Errors can originate in any software engineering phase, and there are a variety of ways to prevent the errors. A well-accepted premise of software engineering is that early detection of errors is beneficial. That is, detecting an error early in the development process is likely to limit the impact of the error, compared to detecting the same error later in the process [13]. In one form or another, early error detection is an aspect of most modern software engineering processes.

This thesis focuses on enabling early error detection during a formal specification phase of software development. The thesis presents a tool-supported technique to validate formal specifications in a straightforward manner that naturally fits into an incremental software development process.

The incremental validation capabilities are provided as part of a Formal Modeling and Specification Language (FMSL). FMSL is comparable to other modern specification languages, such as Z [23] and OCL [63]. The primary contribution of this thesis is the introduction of executability to an FMSL specification. This is provided by a functional interpreter, comparable to that provided by such languages as Lisp [61] and ML [51]. In addition to standard functional evaluation, the FMSL interpreter can execute Boolean expressions containing universal and existential quantifiers, including unbounded quantification.

FMSL is used primarily as a vehicle to teach formal methods to software engineering students. It is currently used by Professor Gene Fisher in software engineering courses at California Polytechnic State University, San Luis Obispo (Cal Poly). Wider distribution of FMSL is planned for the fourth quarter of 2009, via hosting at sourceforge.net, and a dedicated website.

1.1 Description of the Problem

The specific problem addressed in this thesis is how to validate a formal model-based specification. Model behavior is defined with Boolean preconditions and postconditions on model operations. In this context, the problem of validating the specification becomes a problem of Boolean expression evaluation, as is done commonly with interpreted programming languages. The problem of evaluating quantifier expressions is of particular interest in this thesis. This problem is generally not addressed in programming language interpreters. The more general problem discussed in this thesis is how formal methods can be used effectively, particularly in an instructional setting.

Creating a software solution can be a difficult and complex process. There are many ways that people try to improve the software development process: refine the requirements gathering process, improve specifications, create more rigorous test disciplines, select suitable and effective implementation methodologies, etc.

Formal methods and models can be used effectively to describe and analyze a system prior to concrete implementation. Key here is their pre-implementation use.

This can help expose errors, misunderstood properties, and improperly stated behaviors that otherwise might have been overlooked. Gause and Weinberg warn [32] humans are not especially good at seeing what we've overlooked and formal methods and models effectively force the issue. The formal model, which serves to accurately and precisely describe a system, has a utility that is limited by its correctness. That being the case, some consider "analysis of models [to be] a particularly rewarding investment, often exposing problems that can cost much more if not discovered until later" [42]. In this vein, there is a need for tools and methods that help detect errors and increase confidence in formal models.

1.2 Overview of the Solution

This thesis' aims are twofold: (1) to provide a means to validate formal specifications in a straightforward manner that naturally fits into the software development process and (2) to demonstrate how this can be applied practically in an instructional context, within which step-by-step understanding of a specification is an important goal.

Prior to beginning work on this thesis, FMSL existed as a predicative specification language with a formal semantics that supported describing a system comprised of objects and operations. FMSL has a type checker that provides mechanized static analysis of a model. The type checker performs syntactic and semantic analysis comparable to that performed by compilers for strongly-typed programming languages. FMSL also has a documentation generator. This aids in the manual human analysis of a model.

The work of this thesis is to add executability to the FMSL analyzer. This provides the means to execute the operational components of a specification directly. The form of execution focused on particularly is called *operation validation*. The foundation of operation validation is a standard functional interpreter for FMSL. Such an interpreter is comparable to that for interpretable programming languages, including Lisp [61], ML [51], Python [49], and many others.

There is a fundamental difference between a predicative specification written in FMSL and a program written in an interpreted programming language. In the specification, operational behavior is expressed as Boolean predicates that must be true before and after an operation executes. I.e., these are the preconditions and postconditions. The operation itself is not defined with an executable body, as in a programming language. Therefore, what it means to execute a predicate-defined operation can be characterized as follows:

(1) supply inputs and expected outputs for an operation;
(2) evaluate the operation precondition on the given inputs;
(3) if the precondition is true, then evaluate the operation postcondition on the given inputs and outputs;
(4) if the postcondition is true, then the specification is valid for the given set of input/output values.

These steps constitute an operation validation. The solution presented in this thesis defines and implements the means to perform such validations in FMSL.

1.3 Outline of the Thesis

What follows in Chapter 2 is a description of background and related work, which covers formal methods, model checking, and related existing modeling and specification languages. Chapter 3 describes scenarios of system use while Chapter 4 provides an overview of the system design. Chapter 5 discusses the functional interpreter implementation details and Chapter 6 discusses quantifier execution. Chapter 7 concludes with a summary of contributions and lists potential future work.

Chapter 2 Background and Related Work

This chapter provides a background discussion of formal methods and related topics. The subject of "lightweight" formal methods is introduced, with a discussion of how the work of the thesis fits into this category. The related work section provides a survey of relevant specification languages and model checkers. It compares and contrasts the related work on model checking to the approach presented in the thesis.

2.1 Formal Methods

For decades, formal methods have been promoted by researchers as an important part of a rigorous software engineering process. Glass explains that "a formal method of software development is a process for developing software that exploits the power of mathematical notation and mathematical proofs" [33]. Formal methods can be used to express software properties from high-level to low-level. At a high level, a formal model can be used to evaluate whether a system specification satisfies certain properties or meets certain behavioral constraints. At a lower implementation level, formal methods can be used to "formalize, debug, and prove the correctness of algorithms and protocols" [37].

Despite researchers' best efforts, critics contend that formal methods have played a small and insignificant roll in the software engineering process over the last 30 years [33]. In further support of this notion, Heitmeyer points out that "the use of formal methods in practical software development is rare" [37] while Bowen and Hinchey explain that "few people understand exactly what formal methods are or how they are applied" [16].

The critics claim that using formal methods has a high barrier of entry, especially since many formal methods techniques are "difficult to understand and apply" [37] and employ notation that requires significant mathematical expertise [47]. Some argue that formal methods approaches are impractical at best, and there is no compelling reason to incorporate them into their software engineering processes [33]. While that may be true in some cases, formal methods proponents counter-argue that "formal methods are usually the only practical means of demonstrating absence of undesired behavior" [45]. Whether practical or impractical, difficult or easy to understand, if a method helps expose errors, then people likely will consider that method useful. For example, Kurshan observes: "show a designer a bug in the design, and she immediately understands the value of your tool, although she may have little idea how the bug was discovered" [46].

2.1.1 Beneficial Uses of Formal Methods

While general arguments about formal methods continue, there are some demonstrably beneficial uses for formal methods throughout the software engineering process. Formal methods can be used effectively during requirements development, specification, design, and implementation phases [5, 6]. Formal methods employ notations with a well-defined structure, which can be used to present requirements. As observed by Agerholm and Larsen, presenting requirements in formal notations can make "reviewing and inspection easier and therefore useful in locating errors" [1]. Some have found it useful to involve formal methods during the requirements engineering stages, where the formality prompts the engineers to raise questions and "improve the overall quality of the existing specifications" [26]. Although there is more cost associated with formally defining and maintaining a system in multiple notations, experience has shown that early modeling can prove beneficial [26]. On the other hand, when formal methods are not used during pre-implementation stages, design inadequacies only can be exposed once programmers begin building code [41] – a time when it's been shown that design errors are relatively more expensive to fix.

In addition to contributing to more firm and complete requirements and a better system design, formal methods also help people to better understand a system. Users who employ formal methods at early stages are forced to seriously consider fundamental design questions, and formal models can succinctly separate concerns and effectively express system properties [42]. Particularly when dealing with complex systems, the abstraction capabilities of many formal methods often prove to be rather helpful. The formal methods can serve to describe a system in an abstract fashion such that the complexities are masked and so the users acquire a better understanding of the system [1].

2.1.2 Formal Methods for System Parts

Formal methods need not be applied across an entire system. As Bowen and Hinchey advise, “There are occasions in which formal methods are in a sense ‘overkill’, but in other situations they are very desirable” [16]. Agerholm et al. conclude that sometimes “only parts of the systems would benefit from a formal model” [1]. Others have seen positive effects of taking a minimalist approach to formal methods. Easterbrook et al. observed that they could better handle effects of changing requirements by modeling only the specific properties of interest [26]. It may require consideration to determine where formal methods use might be most advantageous to use [47].

The benefits of code reuse are well accepted. A benefit of using formal methods is the potential for model reuse. Once system parts have been formalized into a model then those model parts can be reused [45], for example in later projects. That formal methods are reusable is a major benefit, but formal methods also promote code reuse, in particular when the code has an accompanying, succinct description of guarantees and assumptions then it’s easier to effectively re-use that code [42]. Formal methods and models are appropriate tools to describe those guarantees and assumptions.

2.1.3 Cost Effectiveness

While model and code reuse can contribute cost savings to a software project, a common myth surrounding formal methods use is that they’re just too expensive –

cost- and time-wise – to be viable in industry. Empirical evidence shows that, indeed, use of formal methods early on in development adds up-front costs; however, often the effort is recovered later [47]. Also, although using formal methods usually requires that the users know some formalized notations, to train employees in formal methods topics does not cost more than typical on-the-job, high-tech training [47].

2.2 Model Checkers and Theorem Provers

Once a formal model is in place, it may be a worthwhile exercise to determine whether the model is correct. That in mind, much research has gone into developing model checkers. According to Chan et al. [18], model checking is a "formal verification technique based on state exploration." Model checking algorithms "exhaustively explore the state space to determine whether the system satisfies a property." Kuhn et al. add that model checking often involves providing a counter-example to prove that a property does not hold under certain conditions [45], although failure to discover a counterexample does not necessarily prove correctness [40].

Confidence in a formal model is important because "an incorrect model can be worse than no model at all" [42]. Jackson et al. recommend developing a formal model of a system so long as it can be shown that the model describes the system [42]. Another approach to building confidence in a model involves use of theorem provers. Rather than search for counter-examples, theorem provers "assist the user in constructing proofs, generally to show that the specification has desired properties such as absence of deadlock or various security properties" [45]. Although theorem proving technology has been around for decades, it has not been accepted broadly. Some reasons for not being accepted may be that theorem proving tools may require expert users and an application cycle involving theorem provers is "generally slower than a normal product design cycle" [46].

Model checking also can bring to the surface hard-to-find design errors [19], and it does so in a fashion that Kurshan [46] claims actually accelerates the development process thus "significantly decreasing the time to market." For maximum benefit, Kurshan also recommends that model checking be introduced early on, i.e., "at the same time that the first behavioral models are written."

While there are several approaches to model checking, many agree that those model checkers that enable automatic verification are most desirable [36]. That makes sense not just for convenience reasons, but also for cost benefits as Beizer [11] reports that automated testing can reduce the cost of both software development and maintenance.

2.2.1 Model Checking Challenges

Although there are many benefits that come along with model checking, there also are some challenges. Model checking tends to require specialized expertise, and when it's performed by hand then it can be very time consuming or even error-prone [9]. Experts are often needed because model languages can be rather difficult to learn [37]. These specialized experts may be called upon to translate a system into the model checking tool or language and then to interpret the results [9]. Given that experts may be involved and that this process can be time consuming, model checking can be costly [18] despite the overall savings it may offer.

Another problem people encounter when trying to work with model checkers is the state explosion problem. The concept of state explosion is that there can be so many variables that the model "explodes" in size exponentially to a point that the computing resources cannot cycle through or perhaps determine the state space in the given time constraints [18, 52]. Since most models represent some abstraction of the expected implementation, though, the model state space can be somewhat smaller than the system's state space [25]. Myers et al. point out that when attempting to model check, the engineers ought to keep abstraction in mind when modeling a system to help avoid the state explosion problem [52]. To deal with the state space explosion problem others try to work with a flavor of model checking called symbolic model checking. Symbolic model checkers visit a set of states at a time, and the efficiency of this method "relies on succinct representations and efficient manipulations of … predicates" [18].

All this considered, scalability with model checking remains a challenge [2]. Despite the difficulties that may come with performing model checking, model checking activities can help people better understand a system and specification [18]. If model checking exposes an error, the users should keep in mind that the error could indicate a problem with the specification, model, claim, or even developer understanding [52]. That in mind, it's better to discover these sorts of errors earlier rather than later.

2.3 Lightweight Formal Methods

Model checking is generally considered to be a “heavyweight” formal method. The goal of model checking is to fully verify the correctness of a model, specified in a fully formal notation.

In contrast to the heavy-weight approach, “lightweight” formal methods employ techniques that fall short of complete verification. A lightweight method may use a fully formal notation, but not conduct a complete proof, or not specify fully all aspects of a system [20, 38]. A total proof of correctness may not possible in all cases [45].

A lightweight formal method may not even use a fully formal notation. For example, Easterbrook et al. [26] describe lightweight methods as involving “partial analysis on partial specifications, without a commitment to developing … complete, consistent formal specifications.” Such methods do not require that the user be trained in advanced mathematics or be skilled at developing sophisticated proof strategies [37].

Simulation is another example of a lightweight formal methods technique that animates or “electrifies” a model by examining a small subspace of possible states and transitions [42]. Especially when building a model incrementally, simulation may immediately expose easy-to-make mistakes [42]. Having this model available for early simulation also provides the users the convenient ability to test functional requirements of interest [24]. Not only does the process of simulation make the model creation experience “more compelling,” but Jackson et al. also find that “a model that has been simulated is much less likely to contain egregious flaws” [42].

While heavy-duty formal methods do have a use – in especially interesting or critical software components – Jackson explains that lightweight formal methods can be more practical [41]. Dwyer et al. [25] concur that in some cases it is just impractical to use heavy-duty formal methods – e.g., model checking – on large code bases. These points of view together suggest that people should evaluate where it makes sense to use formal methods, as researchers explain that to reap significant benefits checking an entire specification is not necessary [18] and “not everything should be formalized” [21].

2.3.1 Lightweight Formal Methods and Test-Driven Development

Simulation also lends itself to integration with a project’s test philosophy. Since simulation involves examining a small subspace of states, that subspace can be created by executing parts of a specification against a set of test inputs. These relevant test inputs or test cases can have a longer-lasting benefit since they can be reused at any later point in development, to test the actual implementation. This early creation of test cases may fit in well with the philosophy of test-driven development, which calls for programmers to write low-level functional tests before beginning the implementation [7, 10, 28]. Erdogmus [28] found that following this “test-first” philosophy seems to improve productivity. Janzen et al. [43] observed that “test-first programmers are more likely to write software in more and smaller units that are less complex and more highly tested."

The better end-product software may be a result of the developers' increased understanding of the system. Myers et al. explained that "if you run simple claims early on and then gradually increase the complexity of your claims to explore intricacies of the system behavior then you have a basis of understanding both the model and the system" [52]. This improved understanding can help developers to more easily spot errors or problems, and it can improve customer-developer communication [45]. It would seem that early simulation and test-first together are a synergistic combination, and since testing costs typically make up a significant portion of overall software labor costs [11] then this synergy should be friendly on the budget.

The ultimate synergy between formal specification and test-driven development may come with the wider-scale adoption of automated test generation tools, such as Korat [17] and the commercial product JTest [54]. With such tools, unit test cases are generated automatically from specified preconditions and postconditions. In this way, a specification-driven methodology automatically becomes a test-driven methodology. If a tool does not generate a sufficient set of tests, then manual test creation supplements the generated cases. The formal specification can be used synergistically to guide manual test creation, based on the many years of research in specification-based testing [57].

2.3.2 Lightweight Formal Methods and UML

The Unified Modeling Language is generally not regarded as formal, since it lacks a fully formal semantics. However, there has been a significant amount of work on integrating formal methods into UML. The Object Constraint Language (OCL) is part of UML itself, and is discussed further in Section 2.4.4 of this thesis. Several formalized versions of UML have been used in conjunction with the specification of software security [4]. For general-purpose use, UML-B is an integration of UML and the B formal specification language [59].

UML-based formal methods are arguably all lightweight. Each uses a subset of UML as the basis for formalization. In this way, some but not all properties of a complete UML specification can be treated formally.

2.3.3 Cost Effectiveness

On the topic of costs, the choice to utilize lightweight methods may be both practical and cost-effective [37]. Jackson agrees that "a small amount of modeling and analysis during the initial determination of requirements, specifications, or program design costs only a tiny fraction of the price tag of checking all the code but provides a large part of the benefit gained from an exhaustive analysis" [41]. This relatively low-cost investment provides reasonable coverage of test cases against a model, and yields an increased confidence in the model's correctness [18]. If more assurance is needed after utilizing lightweight formal methods, model checking can be used in selected, particularly critical aspects of the model [25].

For all the above reasons, lightweight formal methods may be attractive to industry. Since lightweight formal methods provide something of an incremental change to existing software processes – rather than a revolutionary change – they may be more likely to be seriously considered, particularly in large organizations where it is difficult to push against process inertia [21, 47].

2.4 Model Checking Tools and Formal Specification Languages

Many automated model checking tools and formal specification languages exist. Each has a set of characteristics that make it suited to particular types of use. In kind, FMSL has its own characteristics and potential uses. What follows is a brief survey of some existing model checkers and formal specification languages: VeriSoft, SMV, JML and Korat, UML/OCL, OOSPEC, and Aslantest.

2.4.1 VeriSoft

VeriSoft, developed at Bell Laboratories, is a "general-purpose 'model checker'" [19] tool that explores the state spaces of a concurrent system in order to detect potential problems such as deadlocks (when each system process' next operation is blocking) and violations of user-specified assertions [34]. Rather than analyzing a separate system model, VeriSoft directly analyzes the actual system implementation. VeriSoft performs system analysis through a scheduler that controls relevant processes on a system by controlling and observing visible operations, which are operations that facilitate inter-process communication. Through system re-initialization and the ability to suspend and resume processes, VeriSoft can explore transitions been system states and report back the sequence of states that led to a system problem. VeriSoft offers an automatic state space exploration mode and a manual mode where the user can explore specific paths between system states.

VeriSoft assumes that a system is deterministic, i.e., it performs the same sequence of execution steps for the same data inputs. The authors of VeriSoft recognized that the environment in which a system operates can add elements of non-determinism to the system's execution, and so they implemented a mechanism that allows the user to optionally hook a user-defined environment implementation together with VeriSoft. While optional, this hook mechanism enhances VeriSoft's utility since it can enable the user to run VeriSoft through a more realistic collection of state spaces.

2.4.2 Symbolic Model Verifier

The Symbolic Model Verifier (SMV) tool checks finite state machine representations of systems that range from synchronous to asynchronous and from detailed to abstract [50, 52]. This experimental SMV tool accepts as inputs a system model description and a set of expected properties of the system, expressed in computational tree logic (CTL). The SMV input language that describes the model has a formal semantics and includes support for modular descriptions and re-usable components. The data types available to SMV are finite data types (Booleans, scalars, fixed arrays, and static structured data types). The expected properties are checked against the model using an ordered binary decision diagram (OBDD). A diagram-based algorithm is used to determine whether the CTL property specifications are satisfied in the model. If it discovers that some part of the specification is false, the SMV model checker attempts to produce and output a counterexample to prove that the model is not correct. McMillan [50] suggests that SMV is a tool intended to facilitate experimentation with symbolic model checking techniques as applicable to hardware verification.

To speed up the model checking process, Myers et al. [52] created a GUI-based SMV prototype tool that allows the user to input a visual representation of the model and conveniently enter in properties to check against the model. Their initial version has limited functionality that translates visual state diagram models into SMV input language code, but they describe their ideal version as something that allows the user to model complete, complex systems.

2.4.3 JML and Korat

The Java Modeling Language (JML) [48] is a behavioral interface specification language that is intended to be used for specifying Java modules by describing preconditions, postconditions, and intermixed assertions. Leavens et al. [48] created JML with the additional goals that it be "readily understandable" by Java developers and that the language be "capable of being given a rigorous, formal semantics, and must also be amenable to tool support." Rudimentary uses of JML include placing Boolean precondition (keyword: `requires`) and postcondition (keyword: `ensures`) specifications in comments above Java method declarations within .java source files, although JML specifications can exist in standalone specification files as well.

An example tool built on JML is Korat, a "framework for automated testing of Java programs" [17]. Korat is novel in that it works by first generating the set of all non-isomorphic inputs, bounded by a given size, that satisfy the Boolean `requires` precondition specified in JML. Korat uses the JML tool-set to generate a test oracle from the Boolean `ensures` JML postcondition in combination with the generated inputs. Finally, Korat executes the method on all these generated test inputs and evaluates the method outputs against the test oracle, and Korat reports any postcondition violations as counterexamples [3, 17].

2.4.4 UML and OCL

The Unified Modeling Language (UML) [14] is a visual language that facilitates the description or modeling of software designs and patterns, and it has become the "de facto standard for modeling software applications" [56]. A UML model generally consists of one or more diagrams and "provides a more compact code description than an ordinary programming language does" [58].

Although UML typically is not thought of as an executable language, there are some subsets of UML that can be rendered executable. These subsets consist of one or more of the following forms of UML elements: class diagrams, StateChart diagrams, activity diagrams, sequence diagrams, and the Object Constraint Language (OCL) [58]. Bouquet et al. [15] have isolated such a subset of UML 2.1 and clarified the semantics of the subset to make it interpretable by model-based testing tools.

When modeling operations in UML, preconditions and postconditions can be described using pseudocode, OCL, or plain English text [56]. OCL is a language with syntax and keywords, and although it cannot modify the model it can be used to describe preconditions, postconditions, and invariants. Within these descriptions OCL syntax includes support for basic scalar types, conditionals, a let construct for improved expressiveness, and universal and existential quantifiers.

There are mixed opinions of OCL. Some critics claim that OCL expressions are "unnecessarily hard" to read or write [40, 62] yet they concede it is more easily used by non-mathematicians compared to some other modeling languages [40]. Also, OCL is not a standalone language since it always must be accompanied by a UML diagram [40, 62]. Still, Kuhn et al. suggest that the combination of UML with OCL is formal enough that the combination can "provide a rigorous system specification" and could be used by model checkers [45].

2.4.5 OOSPEC

OOSPEC [55] is an executable "model-based specification language and development system" intended to be used to introduce formal methods and specifications to undergraduate students. OOSPEC has an object-oriented form with concepts of classes, inheritance, instances, and objects and it supports “high level” structures like sets and sequences. In OOSPEC, operations are specified completely through preconditions and postconditions described in a predicate calculus notation that allows for sequential, conditional, and iterative evaluation. Paryavi et al. [55] also provide a graphical user interface environment prototype that allows for “creation and evaluation of partial and full specifications.”

2.4.6 ASLAN and Aslantest

ASLAN [8] is a formal specification language that takes the state-based approach to describing systems. ASLAN supports identifiers, lists, sets, types, conditional statements, quantification, constraints, and invariants. All these together enable the ASLAN user to specify a system in terms of a collection of states and definitions of state transitions with specific entry and exit criteria (similar to preconditions and postconditions).

Aslantest [24] is a symbolic executor tool that animates and tests Aslan formal specifications to give the user assurance that the model satisfies functional requirements. Aslantest provides the user with two approaches of animating specifications: individual test case evaluation and symbolic execution. The individual test case evaluation allows for testing specific examples that the user considers to be important, while the second approach – symbolic execution – is a method that enables the user to establish proofs about the model since the results consist of symbolic values and constants.

The Aslantest tool provides the user an interface to conveniently navigate through the specification animation process. The tool allows the user to enter in Aslantest commands interactively, but a sequence of commands also can be read from a text file. With the tool, the user can:

- execute state transitions one at a time or in sequence
- get debug information about the current state
- save the state or restore a state
- add assertions

2.5 Empirical Successes with Formal Methods

Through research and industry experiments, researchers have tried to gather information to evaluate whether formal methods really are useful. The following sub-sections summarize several industry and university experiments, all of which conclude that formal methods are beneficial.

2.5.1 BASE: A Trusted Gateway

Larsen et al. conducted an experiment at British Aerospace Systems and Equipment Ltd. (BASE) to determine the cost and quality effects of utilizing formal methods during development of a system [47]. BASE had a need for a "trusted gateway," and so they created two teams of similarly qualified engineers to develop the system independently. One team followed conventional methods and the other was encouraged to use formal specification wherever the team deemed it appropriate. Throughout development both of these teams were monitored to observe engineering methods, communications with the customer, and other development activities.

After reviewing the customer requirements, both teams were given the opportunity to ask the customer for additional detail. Larsen et al. observed that the formal methods team not only asked more questions – 60 vs. 40 – but their questions focused heavily on the data and exceptional conditions, which is a sensible emphasis when developing a security-critical system. Also, the formal methods team's modeling of the system shed light on an exceptional condition that was not initially called out in the original requirements. The conventional methods team did not catch the potential occurrence of the exceptional condition, and they later had to develop a patch to their software.

Once the teams finished initial implementations of their trusted gateway software, Larsen et al. tested the systems using the identical user interface that was provided to both teams. The trusted gateway systems were run against their separately developed test suites and then run against each other's test suites. The conventional methods team's software failed some of the formal methods team's tests, which included testing of the exceptional condition mentioned above. The trusted gateways also were benchmarked for performance and the formal methods team's software performed fourteen times faster during normal operation, although it took longer to initialize (which was an acceptable trade-off given the requirements). Lastly, the overall effort spent by both teams was roughly equivalent, which ran counter to some criticisms of formal methods that claim formal methods are prohibitively expensive for use in industry.

2.5.2 Miami University of Ohio: OOD Course

Sobel et al. conducted an experiment to judge the effects of integration of formal methods techniques into an undergraduate software engineering curriculum [60]. The experiment sought to evaluate students' potential for learning formal methods and to increase their complex problem solving skills. To carry out the experiment Sobel et al. worked with two separate classes broken into teams of students for an Object Oriented Design (OOD) course: one control group of thirteen teams that had taken the university's normal curriculum and one formal methods group of six teams that had taken two semesters of formal methods courses. The teams' workflow on a common elevator project was monitored to observe design and implementation efforts and methods. All teams were asked to provide executable source code for this project and all teams were encouraged, but not required, to submit a UML diagram of their system design. The formal methods group was additionally asked to submit a formal specification – a first order logic description of preconditions, postconditions, and invariants – of their system.

The experiment showed that the formal methods teams generally followed a more rigorous design process. For example, none of the thirteen control teams submitted a UML diagram of their design (in fact, no design artifacts could be found) whereas three (out of six) of the formal methods teams submitted UML diagrams of their design and four of the formal methods teams submitted a formal specification. Although some of the formal methods teams used symbols incorrectly (for example, they interchanged existential and universal quantifiers), their system description demonstrated a good understanding of the system behavior. In all, the formal methods teams had relatively better designs.

The formal methods teams' implementations had a significantly better test success rate compared to the control group teams' submissions: 100% correctness vs. 45.5% correctness. Of the thirteen control teams, two did not provide any submission at all. Overall, the formal methods teams' source code was less complex while the control teams' source code was more complex and offered poorer, more tightly coupled solutions. Sobel et al. were surprised that the various teams across the control and formal methods groups produced solutions with counts of source lines of code that were not significantly different, but the benefits of formal methods training were clear: 100% of the students trained in formal methods techniques produced correct solutions compared to only 45.5% of the control teams' students.

2.5.3 NASA: Lightweight Formal Methods

At the National Aeronautics and Space Administration (NASA), many engineering practices rely on informal processes – such as inspection – and generally do not employ careful requirements engineering in critical areas [26]. Easterbrook et al. set out to observe the effects of implementing lightweight formal methods in several NASA programs to evaluate whether their incorporation into existing engineering practices might yield increased safety or reduced cost. Their approach involved assigning formal methods experts the task of incorporating formal methods techniques early on in the requirements phases of three new space systems where many of the requirements were still volatile. In these three cases they followed a common approach that involved unambiguously re-stating requirements, identifying and correcting inconsistencies, testing the requirements, and finally discussing the results with the requirements' authors.

Ultimately the authors of [26] did not perform an extensive analysis on the cost benefits of formal methods in their studies, but they concluded that application of formal methods early on added value since their use helped detect errors and clarify requirements. Examples of the many types of requirements problems that formal methods helped uncover include: ambiguities, inconsistencies, missing assumptions, missing preconditions, traceability problems, logic errors, missing requirements, inadequate requirements, and incorrect expression of timing requirements. Easterbook et al. also observed that the development team was much more receptive to working through these errors discovered through the use of formal methods, since these techniques were applied so early on in the process.

2.6 The Work of this Thesis in the Spectrum of Formal Methods

While some of the aforementioned languages and tools may have similarly positive impacts on a software project, FMSL's qualities and characteristics distinguish it from other formal methods languages and tools. The remainder of this sub-section summarizes some of these differences.

Whereas Verisoft [19] is a model checking tool that analyzes a system implementation, FMSL is suitable for pre-implementation formal modeling, which can be beneficial since “verification at early stages is more likely to be tractable” [18]. Vaziri and Jackson assert that it is “near impossible to get a system right by fudging late in the day, so early investment in modelling and analysis will be essential” [62].

The FMSL language itself provides formal methods capabilities in a practical and balanced fashion. For example, like the SMV input language [50], FMSL has a formal semantics – a must for modeling languages [42] – and it exhibits a natural language expressiveness that should be “familiar to the user” [37]. These qualities could make FMSL appealing to non-software professionals [36] and engineers [47]. Unlike SMV, which is a heavier-weight model checking tool that uses a diagram-based algorithm to search for counterexamples to prove that a model is not correct, FMSL provides users with a lighter-weight approach that does employ exhaustive model checking algorithms.

FMSL specifications do not lend themselves to any specific implementation programming languages, whereas JML [48] is intended to be used for specifying Java modules. A separate GUI front-end to facilitate specification validation could be an effective companion tool for FMSL (see Section 7.2.3), and JML also can be integrated with other tools. Korat [17], which automatically generates test cases for JML specifications, is one such tool.

Unlike some other languages – for example, OCL – FMSL does not fall into the category of being so implementation-oriented that it's not "well-suited for conceptual modeling" [62]. Another difference between OCL and FMSL is that, as mentioned in a preceding sub-section, OCL must be accompanied by a visual UML diagram [40, 62]. While visual representations of FMSL specifications may have some utility, they are not required.

OOSPEC [55] and FMSL share some common qualities: both are used to introduce formal methods and specifications to undergraduate students, both have an object-oriented form, and both support operation specification through precondition and postcondition definition. FMSL's combination of a functional interpreter and a means to execute preconditions and postconditions may make FMSL useful and appealing to software engineering students, who expect an executable specification language [55]. One major difference between them is their respective styles of specification expression. Specifically, while FMSL draws heavily from functional programming languages, OOSPEC draws strongly from languages that utilize set-theoretic notation – VDM [12] and Z [23].

ASLAN [8] and FMSL support similar similar features like identifiers, lists, types, and quantification. While ASLAN users specify systems in terms of states and state transitions, FMSL users specify systems using objects and operations with constraints. State transition entry and exit criteria constraints in ASLAN are comparable to operation preconditions and postconditions in FMSL. While both Aslantest [24], a tool that executes ASLAN specifications, and FMSL support execution through individual test cases, Aslantest also supports symbolic execution.

FMSL and Aslantest also handle quantifier execution differently, and those differences are discussed in Chapter 6.

With its particular set of qualities and characteristics, FMSL is designed to be easy to understand. According to Sobel and Clarkson, even those who do not fully understand a formal modeling language (or formal method) still can gain some benefit from using it [60]. In addition to the academic benefits, while Jackson cautions that "as in a building, when the software's foundation is unsound, the resulting structure is unstable" [41], using FMSL to describe and validate a model may increase the likelihood that the model will serve as better foundation for the software that implements the model. The FMSL modifications for this thesis aim to transform FMSL into a more effective and useful tool that fits well with lightweight formal methods techniques, and so its use could be introduced incrementally.

Chapter 3 Demonstration of Tool Capabilities

FMSL specifications consist primarily of object and operation definitions. The following is a simple illustrative example.

```
object PersonList
    components: Person*;
    description: (*
        A PersonList contains zero or more Person records.
    *);
end PersonList;

object Person
    components: firstName:Name and lastName:Name and age:Age;
    description: (*
        A Person has a first name, last name, and age.
    *);
end Person;

object Name = string;
object Age = integer;

operation Add
    inputs: p:Person, pl:PersonList;
    outputs: pl':PersonList;
    precondition: not (p in pl);
    postcondition: p in pl';
    description: (*
        Add a person to a list, if that person is not already in the
        list.
    *);
end Add;
```

Figure 3.1: Sample FMSL specification

This example illustrates the two primary forms of definition in FMSL: objects and operations. Objects have components, which are defined in terms of other objects. Object definitions "bottom out" in one of the built-in primitive types of `integer`, `real`, `string`, or `boolean`.

Operations have inputs, outputs, preconditions, and postconditions. The types of inputs and outputs are the names of defined objects. Preconditions and postconditions are boolean expressions. Other notational features worthy of explanation are the following:

- '(*' and '*)' are used to enclose comments
- `Name` and `Age` use an optional short form of object definition; it can be useful for objects of simple scalar types, with no description
- the `in` operator is built-in; it tests for list membership
- any identifier can have an apostrophe character as a suffix; this is purely a lexical form, in that a trailing apostrophe is a legal character in an identifier; it is used most often in operation outputs when the type of an input and output object are the same; e.g., the `Add` input list is named `pl` and the output list is `pl'`, read "*pl prime*"

A complete discussion of FMSL syntax and semantics is given in its reference manual [29]. This thesis will only use a subset of its features, specifically those features that are germane to the topic of specification validation.

Given a specification such as the example above, a basic question is this: "How does one validate that it is correct?" Firstly, static correctness can be validated using the FMSL type checker, which performs syntactic and semantic analysis comparable to that performed by a programming language compiler. A particularly useful part of static analysis is completeness checking. For example, if the specifier left out the definitions of the `Name` and `Age` objects, the checker would flag the error in the definition of the `Person` object that uses `Name` and `Age`.

The focus of this thesis is determining the dynamic correctness of a specification. For an operation, this fundamentally requires some means of evaluation. In the example at hand, the `Add` operation could be evaluated in the manner shown in Figure 3.2:

```
(*
 * Sample person, an empty person list, and a one-person list
 *)
value p:Person = {"Arnold", "Schwarzenegger", 61};
value pl:PersonList = [];
value pl':PersonList = [p];

> Add(p, pl);                      -- invoke the Add operation
```

Figure 3.2: Person definitions with Add

The following aspects of notation warrant brief explanation:

- a `value` declaration defines a constant value of some type of object
- tuple values are enclosed in curly braces; a tuple is an object defined with `and`ed components
- list values are enclosed in square brackets; a list is an object defined with `*` components
- point-to-end-of-line comments are defined with '`--`'

- expression evaluations are preceded with the prompt character '>'; these are typically entered in the top-level of a conversational interpreter, but may be included within a specification file; the important point is that the '>' prompting character distinguishes an expression to be evaluated from a specification declaration, in this and all subsequent examples.
- an operation is invoked in the way standard to most programming languages, with the operation name followed by a parenthesized list of actual parameters

So, the question at hand is "*What value does the invocation of* `Add(p, pl)` *produce?*" Since the `Add` operation has no defining expression, the value of invoking `Add(p, pl)` is `nil`, where `nil` is the empty value for any type of object. `Nil` is in fact is result of evaluating `Add` for any inputs, given that `Add` is defined only with a precondition and postcondition.

The precondition and postcondition for `Add` define a behavior. However, they do so in a declarative and analytic form, not a constructive form. It is possible to define FMSL operations constructively, but that is not the point here. What is desired is a way to validate `Add`'s precondition and postcondition, given a particular set of inputs and expected outputs.

One way to do this is to extract the precondition and postcondition expression, and evaluate them individually. For example, given the preceding value declarations, the precondition expression could be tested with logic expressions such as those shown in Figure 3.3.

```
> p in pl;                        -- should be false
> not (p in pl);                  -- should be true
> not (p in pl');                 -- should be false
```

Figure 3.3: Precondition logic expressions

The postcondition expression could be tested as in Figure 3.4:

```
> p in pl';                       -- should be true
> not (p in pl');                 -- should be false
```

Figure 3.4: Postcondition logic expressions

These are clearly rudimentary expressions. The point is that the logic of preconditions and postconditions can be dynamically validated by plugging in various values and examining the results. The work of this thesis has included the implementation of this form of expression evaluation in FMSL. This form of evaluation supports the notion cited earlier from Myers [52]: "if you run simple claims early, ... then you have a basis for understanding both the model and the system."

While isolated evaluation of boolean expressions can be helpful, it would be even handier to invoke an operation with sample input and output values directly. This kind of *validation invocation* can be characterized as follows for the `Add` precondition: *Given inputs p and pl, what is the value of the Add precondition?*

A more complete validating invocation is this: *Given inputs p and pl, expected output pl', what are the values of the Add precondition and postcondition?*

The concrete syntax for such a validation invocation looks like this:

```
> Add(p, pl) ?-> pl';
```

The output of this validating invocation is a boolean two-tuple, that looks like this:

```
{ true, true }
```

The notational particulars are these:

- the first part of a validation invocation looks like a regular operation call, e.g., `Add(p, pl)`
- the '`?->`' is the validation operator[1]; per the preceding characterization, it means the following in this example: *Given inputs p and pl, is the Add precondition true, and given pl', is its postcondition true?*
- the output value of `{ true, true }` is the standard curly brace notation for a boolean two-tuple

[1]The somewhat curious syntax of the validation operator is derived from the FMSL syntax for operation signatures. I.e., the signature of the Add operation is (Person, PersonList) -> PersonList, where the -> notation has been used in other specification languages in the denotation of input/output signatures.

A validation counter example can be tested, such as

```
> Add(p, pl) ?-> pl;
```

which produces the result `{ true, false }`.

The preceding introduction to Chapter 3 has presented a simple motivating example. The remainder of this chapter will cover the details of specification evaluation, including in particular the evaluation of conditions with quantifiers. The coverage will feature the validation of a long-standing pedagogic example, in which the use of validating evaluations revealed a heretofore undiscovered flaw. This is a particularly good result, and demonstrates well the utility of dynamic specification validation.

3.1 Standard Expression Evaluation

In FMSL, expression evaluation entails invoking an operator or operation and returning the calculated result. This is the same behavior as exhibited by interpreted programming languages, including Lisp [61], ML [51], and Python [49].

FMSL has a strongly-typed, functional semantics, much like that of ML. There is limited type inference, in the form of value declaration and let variables, that can be declared without explicit types. More advanced type inference, such as that available in ML and Haskell [39] is purposely omitted from FMSL. As a modeling and specification language, it is considered appropriate for the specifier to declare object and parameter types explicitly, rather than having types inferred by a language translator.

FMSL supports evaluation of a collection of built-in Boolean, arithmetic, tuple, and list expressions as well as evaluation of user-created operations. For a complete list of built-in operators, see Tables 5.2 through 5.6.

The example in Figure 3.5 demonstrates evaluation of the Boolean relational operators: `not`, `and`, `or`, `xor`, => (implication), and <=> (two-way implication. In the example, the FMSL code first declares two `boolean` values and then performs a series of Boolean expression evaluations.

```
(*
 * Declare short value names for true and false
 *)
val t:boolean = true;
val f:boolean = false;

(*
 * Boolean operator examples
 *)
> not t;                          -- evaluates to false
> t and f;                        -- evaluates to false
> t or f;                         -- evaluates to true
> t xor f;                        -- evaluates to true
> t => f;                         -- evaluates to false
> t <=> f;                        -- evaluates to false
```

Figure 3.5: Evaluating Boolean expressions

A notational matter in Figure 3.5 is the use of the abbreviated keyword `val` in place of `value`. FMSL provides abbreviated versions of all major keywords, as a matter of readability.

The example in Figure 3.6 demonstrates evaluation of the arithmetic division operator. In the example, the FMSL code first declares two `real` values and then performs the division (with result: `1.15573`).

```
(*
 * Declare and assign values to x, y
 *)
val x:real = 3.141592654;
val y:real = 2.718281828;

(*
 * Evaluate x divided by y and output the result
 *)
> x / y;
```

Figure 3.6: FMSL division operator expression evaluation

Further examples of expression evaluation appear in this and following chapters.

3.2 Quantifier Evaluation

Quantifiers are Boolean-valued expressions that evaluate a quantified sub-expression multiple times. FMSL supports both bounded and unbounded universal (`forall`) and existential (`exists`) forms of quantification. A bounded quantifier ranges over a discrete set of values. An unbounded quantifier ranges over all of the values in a type of object. For types grounded in integer, real, or string, the quantifier range is unbounded.

Formally, an object definition defines a data type. As noted earlier, FMSL has a strongly typed semantics, meaning that the types of all declared values, variables, and operation parameters are determined statically, before any expression evaluations takes place.

FMSL employs a structural type equivalence rule, meaning two data types are equivalent if they have the same type structure, whether or not they have the same object name. As described below, a name-based typing scheme is used to define the value universes, for the purposes of evaluating unbounded quantifiers in bounded time. This name-based typing is used as an expedience for quantifier evaluation, and does interfere with the purely structural-equivalence typing performed during the static type checking of a specification.

The following sub-sections describe the evaluation of different forms of quantifier expressions. In the examples, the `Person` object is defined by the FMSL code listing in Figure 3.7, which is the definition that appeared in the introductory example at the beginning of Chapter 3.

```
(*
 * Define the Person object type
 *)
object Person is
   components: firstName:Name and lastName:Name and age:age;
   description: (*
      A Person has a first name, last name, and age.
   *)
end Person;
```

Figure 3.7: FMSL Person object type definition

The form of quantification in FMSL is common to that of typed predicate logic. The general format of universal quantification is the following:

```
forall (x:t) predicate
```

This is read “for all values *x* of type *t*, *predicate* is true” where *x* must appear somewhere in predicate.

There are also two extended forms of `forall`, shown in Table 3.1.

Extended Form	**Reading**	**Equivalent To**

`forall (x:t | p1) p2`	For all *x* of type *t*, such that *p1* is true, *p2* is true.	`forall (x:t)`
		`    if p1 then p2`

`forall (x in l) p`	For all *x* in *l*, *p* is true.	`forall (x:`*basetype*`(l))`
		`    if x in l then p`

Table 3.1: Extended forms of forall

Existential quantification has three comparable forms, seen in Figure 3.8:

```
exists (x:t) predicate
exists (x:t | predicate1) predicate2
exists (x in l) predicate
```

Figure 3.8: Existential quantification forms

3.2.1 Bounded Quantifier

The code in Figure 3.9 creates a list of `integer` values and then evaluates a bounded quantifier to check whether all the `integer` elements are positive. Since all the `integer` elements are positive, the result is `true`.

```
(*
 * Declare an IntList object type and an IntList value
 *)
obj IntList = integer*;
val list:IntList = [ 1, 1, 2, 3, 5 ];

(*
 * Test that all the integer elements within list are positive.
 *)
> "Expected: true";
> forall (i in list) i > 0;            -- evaluates to true
```

Figure 3.9: FMSL bounded quantifier example

3.2.2 Unbounded Universal Quantifier: forall

The code in Figure 3.10 declares two `Person` values and then evaluates an unbounded quantifier to test that all the `Person` objects have non-nil last names. Since the two existing `Person` objects have non-nil last names, the result is `true`.

```
(*
 * Create values p1 and p2, which puts them in the Person value
 * Universe.
 *)
val p1:Person = {"Alan", "Turing", 97};
val p2:Person = {"Arnold", "Schwarzenegger", 61};

> forall (p:Person) p.lastName != nil;      -- evaluates to true
```

Figure 3.10: FMSL unbounded forall quantifier example

Conceptually, the universe of all values of type `Person` is unbounded, since it consists of component types `integer` and `string`. Clearly, however, a means must be established to execute the quantifier in bounded time. Simply put, the value universe for an unbounded quantifier consists of all values of the quantified type that have come into existence during a particular execution session. In this small example, there are only two values populating the universe of the `Person` type. Complete details of quantifier evaluation are covered in Chapters 4 through 6 of the thesis.

3.2.3 Unbounded Existential Quantifier: exists

The code in Figure 3.11 declares two `Person` values and then evaluates an unbounded quantifier to indicate whether there exists a `Person` object with a nil last name. Since all the `Person` objects have defined last names, the `exists` expression evaluates to `false`.

```
(*
 * Create values p1 and p2, which puts them in the Person value
 * Universe.
 *)
val p1:Person = {"Alan", "Turing", 97};
val p2:Person = {"Arnold", "Schwarzenegger", 61};

> exists (p:Person) p.lastName = nil;          -- evaluates to false
```

Figure 3.11: FMSL unbounded exists quantifier example

3.2.4 Unbounded Universal Quantifier: forall with such that

The code in Figure 3.12 declares three `Person` values, but unlike the previous two examples this sequence of value declarations includes a `Person` value that has a `nil` last name. The unbounded quantifier with a such that clause evaluates whether all `Person` objects with non-nil last names have last name lengths of at least six characters long. The result of this expression is `true`.

```
(*
 * Create values p1 and p2, which puts them in the Person value
 * Universe.
 *)
val p1:Person = {"Alan", "Turing", 97};
val p2:Person = {"Arnold", "Schwarzenegger", 61};
val p3:Person = {"Charles", nil, 218};

(*
 * Evaluate: for all Person objects such that p.lastName is not nil,
 * the last name length is at least 6 characters.
 *)
> forall (p:Person | p.lastName != nil) #p.lastName >= 6; --eval true
```

Figure 3.12: FMSL unbounded forall / suchthat quantifier example

3.3 Operation Validation

This section describes how a user can utilize the FMSL validation operator (`?->`) to incrementally validate a specification by performing a sequence of operation validations. Recall from the earlier brief description, an invocation of the validation operator requires an operation name, an input argument list, and an output argument list. The general format is the following:

```
operation_name(input argument list) ?-> (output argument list)
```

FMSL uses input and output arguments as values in the specified operation's precondition and postcondition to execute the precondition and postcondition. The result of the validation operator invocation is a tuple that contains two `boolean` values: the first expresses the result of the precondition evaluation and the second expresses the result of the postcondition evaluation.

The material in the following sub-sections steps through the formalization of selected components of a simple user database specification. The user database specification is part of an extended pedagogical example for a distributed calendaring application [31]. The example is used for undergraduate instruction at Cal Poly University, San Luis Obispo. The specific course is Introduction to Software Engineering, CSC 308, as taught by Cal Poly faculty member Gene Fisher.

The following examples come directly from Fisher's CSC 308 lecture notes, weeks 7 and 8 [30]. Some of the explanatory text in the thesis is excerpted verbatim from the notes. For the following examples, the object type definitions in Figure 3.13 apply. These definitions describe individual components of a user record and a user record database.

```
object UserDB
    components: UserRecord*;
    operations: AddUser, FindUserById, FindUserByName ChangeUser,
                DeleteUser;
    description: (*
        UserDB is the repository of registered user information.
    *);
end UserDB;

object UserRecord
    components: name:Name and id:Id and email:EmailAddress and
        phone:PhoneNumber;
    description: (*
        A UserRecord is the information stored about a registered
        user. The Name component is the user's real-world name.  The
        Id is the unique identifier by which the user is known to
        the Calendar Tool.  The EmailAddress is the electronic mail
        address.  The PhoneNumber is for information purposes.
    *);
end UserRecord;

object Name = string;
object Id = string;
object EmailAddress = string;
object PhoneNumber = area:Area and num:Number;
object Area = integer;
object Number = integer;
```

Figure 3.13: FMSL UserDB and UserRecord definitions

3.3.1 AddUser: English Precondition and Postcondition in Comments

In the lecture notes, the formalization process begins by first stating the precondition and postcondition predicates in English. In Figure 3.14, each of the `AddUser` inputs and outputs appears with a name and corresponding type. By convention, if an operation uses the same type as both an input and output, the name of the output is the same as the input with an apostrophe appended; the apostrophe is read "prime". Note that the precondition and postcondition are described in English and are enclosed in comments.

```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;

    precondition:
        (*
         * The id of the given user record must be unique and less
         * than or equal to 8 characters; the email address must be
         * non-empty; the phone area code and number must be 3 and 7
         * digits, respectively.
         *);

    postcondition:
        (*
         * The given user record is in the output UserDB.
         *);

    description: (* As above *);

end AddUser;
```

Figure 3.14: AddUser with English precondition and postcondition

Although the `AddUser` precondition and postcondition descriptions from Figure 3.14 appear only in plain English, this form of the `AddUser` operation already is executable through the validation operator. To demonstrate this executability, in Figure 3.15 we create a set of sample user record inputs, an initial database, and the expected output result of adding a user record to the initial database. The last line of the example invokes the validation operator with input and output arguments, and we expect the precondition and postcondition execution result tuple to be `{ true, nil }`.

```
(*
 * Create some testing values.
 *)
val ur1 = {"Corwin", "1", nil, nil};     -- sample user record
val ur2 = {"Fisher", "2", nil, nil};     -- sample user record
val ur3 = {"Other", "3", nil, nil};      -- record to be added
val udb = [ur1, ur2];                    -- the initial input db
val udb_added = udb + ur3;               -- the expected result

> print("Expected results of AddUser(udb,ur3)?->(udb_added) are:\n");
> print("{ true, nil }\n");
> AddUser(udb,ur3)?->(udb_added);
```

Figure 3.15: AddUser basic tests

By definition an operation without a precondition has no entry constraint, and so the precondition execution result tuple field is `true`. As there is no postcondition defined, and since the absence of a postcondition is represented in the result tuple by `nil`, we see `nil` as the postcondition execution result tuple field.

In Figure 3.15, plain strings are used as output messages. As is typical in interpreted programming languages, top-level execution is performed with a *read-eval-print* loop. That is, an expression is read from a prompted input line, the expression is evaluated, and the result is printed. There is a built-in print function in FMSL, to provide more in the way of output formatting, but plain strings can be usedful for simple output messaging.

3.3.2 AddUser: Basic Postcondition Logic

The English comment in the postcondition (“`The given user record is in the output UserDB`”) describes the essence of an additive collection operation: the output collection (`udb'`) must contain the user record to add (`ur`). To formally represent this concept, we use the `in` operator shown in Figure 3.16.

```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;

    postcondition:
        (*
         * The given user record is in the output UserDB.
         *)
        ur in udb';

end AddUser;
```

Figure 3.16: AddUser with basic postcondition logic

In Figure 3.17 we create a set of sample user record inputs, an initial database, and the expected output result of adding a user record (`ur3`) to the initial database. The last line of the example invokes the validation operator with input and output arguments. According to the postcondition, since `udb_added` contains `ur3` we expect the precondition and postcondition execution result tuple to be `{ true, true }`.

```
(*
 * Create some testing values.  These are the same as the
 * comment-only version.
 *)
val ur1 = {"Corwin", "1", nil, nil};
val ur2 = {"Fisher", "2", nil, nil};
val ur3 = {"Other", "3", nil, nil};
val udb = [ur1, ur2];
val udb_added = udb + ur3;

> "Expected results of AddUser(udb,ur3)?->(udb_added) are: ";
> "{ true, true }:";
> AddUser(udb,ur3)?->(udb_added);
```

Figure 3.17: Basic tests for formal postcondition

3.3.3 AddUser: Basic Postcondition Logic Challenged

Generally, a fundamental question to ask about preconditions and postconditions is: are they strong enough? Since there is no precondition in the AddUser example, that means it is maximally weak. A later example will focus on strengthening the precondition. In the meantime, we will focus on the postcondition. To check whether the postcondition is strong enough, we can use the validation operator to run some example inputs and outputs against AddUser. The example in Figure 3.18 tests whether the postcondition is strong enough to enforce that there are no spurious additions or deletions from the user database collection.

```
val ur1 = {"Corwin", "1", nil, nil};
val ur2 = {"Fisher", "2", nil, nil};
val ur3 = {"Other", "3", nil, nil};
val ur4 = {"Extra", "4", nil, nil};
val udb = [ur1, ur2];

(*
 * A database value representing a spurious addition having
 * been made.
 *)
val udb_spurious_addition = udb + ur3 + ur4;

(*
 * A database value representing a spurious deletion having
 * been made.
 *)
val udb_spurious_deletion = udb + ur3 - ur2;

> AddUser(udb,ur3)?->(udb_spurious_addition);

> AddUser(udb,ur3)?->(udb_spurious_deletion);
```

Figure 3.18: Test for postcondition strength

The first invocation of the validation operator in Figure 3.18 tests whether the postcondition prevents a spurious addition to the user database, since the output argument contains an extra user record (`ur4`). The second validation operator invocation tests whether the postcondition prevents a spurious deletion from the user database, as that output argument contains a user database that specifically lacks `ur2`. Whereas we would like to see a `{ true, false }` result in both cases, instead the validation tuple that returns is `{ true, true }` since the lack of precondition comes back with a `true` value and the postcondition only tests whether `udb'` contains `ur3`. From that result we can deduce that the `AddUser` postcondition is not strong enough.

3.3.4 AddUser: Strengthened Postcondition Logic

The `AddUser` postcondition in Figure 3.16 checked the fundamental property that we want to hold true: the output collection must contain the user record designated for addition. What it lacked, as evidenced by the results of running the test in Figure 3.18, was a guarantee that the rest of the database would remain intact. To build on the previous postcondition, we can add an additional condition to enforce that all other records in the output database are those – and only those – from the input database. The postcondition in Figure 3.19 reflects this additional constraint on the output database.

```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;

    postcondition:
        (*
         * The given user record is in the output UserDB.
         *)
        (ur in udb')

            and

        (*
         * All the other records in the output db are those from the
         * input db, and only those.
         *)
        forall (ur':UserRecord | ur' != ur)
            if (ur' in udb)
            then (ur' in udb')
            else not (ur' in udb');

end AddUser;
```

Figure 3.19: AddUser with stronger postcondition

When we re-run the test from Figure 3.18 against this updated specification of `AddUser` that contains a stronger postcondition, we find that the validation operator invocation result tuple is `{ true, false }` in both cases. Running sample inputs and outputs through FMSL's validation operator helped uncover that the postcondition initially was too weak, and we used it to verify that the revised postcondition was strong enough to properly handle the "no spurious additions or deletions" requirement.

3.3.5 AddUser: Constructive Postcondition

So far the examples presented have utilized only analytic operations in the postcondition, but when describing preconditions and postconditions we also have at our disposal constructive operations. Constructive operations perform an actual constructive calculation, whereas analytic operations evaluate Boolean expressions about the arguments. In some cases a precondition or postcondition that utilizes constructive operations may be clearer than its corresponding analytic operation-based counterpart. For example, in Figure 3.20 see the `AddUser` specification with a postcondition that contains a constructive operation (the '`+`' or concatenation operator).

```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;

    postcondition:
        (*
         * The given user record is in the output UserDB.
         *)
        udb' = udb + ur;

end AddUser;
```

Figure 3.20: AddUser with constructive postcondition

Analytic specifications, as in Section 3.3.4, have the benefit of introducing minimum implementation bias. Constructive specifications can be useful to simplify specification logic. A complete discussion of the relative merits of analytic versus constructive specification is beyond the scope of this thesis. Validation invocations can be used with either style.

While value construction need not be used in a postcondition, it is definitely required for validations calls. The point of a validation call is to test constructed values against pre- and postcondition logic.

There are different styles to accomplish this. Which style to use is a matter of convenience and clarity of presentation. For example, the set-up in Figure 3.21 creates the same testing values as in the preceding examples, but without using list concatenation or deletion operators. These tests produce the same results, with either the constructive or analytic `AddUser` specification.

```
val ur1:UserRecord = {"Corwin", "1", nil, nil};
val ur2:UserRecord = {"Fisher", "2", nil, nil};
val ur3:UserRecord = {"Other", "3", nil, nil};
val ur4:UserRecord = {"Extra", "4", nil, nil};

> "Expected retults are";
> "{ true, true }";
> AddUser([ur1, ur2], ur3) ?-> [ur1, ur2, ur3];

> "Expected results are";
> "{ true, false }";
> AddUser([ur1, ur2], ur3) ?-> [ur1, ur2, ur3, ur4];

>  "Expected  results  of  AddUser(udb,ur3)?->(udb_spurious_deletion)
are";
> "{ true, false }";
> AddUser([ur1, ur2], ur3) ?-> [ur1, ur3];
```

Figure 3.21: Alternate style of validation invocations

3.3.6 FindUserByName: English Definition in Comments

The following sequence of examples steps through the definition of the `FindUserByName` operation, which is intended to search through the user database and return records with names that match the given `name` input argument. Figure 3.22 has the `FindUserByName` definition, with the precondition and postcondition described in English.

```
operation FindUserByName
    inputs: udb:UserDB, name:Name;
    outputs: ur':UserRecord*;

    precondition: (* None yet. *);

    postcondition:
        (*
         * A record is in the output list if and only if it is in
         * the input UserDB and the record name equals the Name
         * being searched for
         *);

    description: (*
        Find a user or users by real-world name. If more than one is
        found, output list is sorted by id.
    *);
end FindUserByName;
```

Figure 3.22: FindUserByName with English precondition and postcondition

As with the `AddUser` example, at this point `FindUserByName` is sufficiently formally defined so that we can begin running validation operator invocations against it. The FMSL code below creates several `UserRecord` values, a `UserDB`, and collection of possible outputs. The final statements of the example invoke the validation operator on `FindUserByName` to test postcondition strength.

```
(*
 * Create some testing values.
 *)
val ur1:UserRecord = {"Corwin", "1", nil, nil};
val ur2:UserRecord = {"Fisher", "2", nil, nil};
val ur3:UserRecord = {"Other", "3", nil, nil};
val ur4:UserRecord = {"Extra", "4", nil, nil};
val ur5:UserRecord = {"Fisher", "5", nil, nil};

val udb = [ur1, ur2, ur3, ur4, ur5];
val unsorted_result = [ur5, ur2];
val sorted_result = [ur2, ur5];
val too_many_sorted = [ur2, ur2, ur2, ur5];
val too_many_unsorted = [ur2, ur5, ur2, ur2];

(*
 * We want a generously populated universe of integers to be
 * available to FindUser precondition and postcondition
 * constraints, so let's do some populating.
 *)
> [1 .. 100];

> "What happens if there are unique, unsorted records?";
> FindUserByName(udb,"Fisher")?->unsorted_result;

> "What happens if there are unique, sorted records?";
> FindUserByName(udb,"Fisher")?->sorted_result;

> "What happens if there are non-unique, unsorted records?";
> FindUserByName(udb,"Fisher")?->too_many_unsorted;

> "What happens if there are non-unique, sorted records?";
> FindUserByName(udb,"Fisher")?->too_many_sorted;
```

Figure 3.23: FindUserByName operation validation tests

The comment about populating the integer value universe relates to the manner in which unbounded quantifiers are evaluated. This topic is covered fully in Chapter 6 of the thesis.

As in the example from Section 3.3.1, the precondition and postcondition in Figure 3.23 are not yet formally defined, so we expect the result for all four tests to be `{ true, nil }`. Figure 3.24 shows the output where this is the case.

```
"What happens if there are unique, unsorted records?"
{ true, nil }
"What happens if there are unique, sorted records?"
{ true, nil }
"What happens if there are non-unique, unsorted records?"
{ true, nil }
"What happens if there are non-unique, sorted records?"
{ true, nil }
```

Figure 3.24: FindUserByName initial validation results

3.3.7 FindUserByName: Basic Postcondition Logic

A sensible next step in formalizing the postcondition is to make sure that the operation output consists of all records of the given name in the input db. The formal logic in Figure 3.25 contains a postcondition that satisfies this constraint.

```
operation FindUserByName
    inputs: udb:UserDB, n:Name;
    outputs: url:UserRecord*;

    precondition: (* None yet. *);

    postcondition:
        (*
         * The output list consists of all records of the given name
         * in the input db.
         *)
        (forall (ur: UserRecord)
            (ur in url) iff (ur in udb) and (ur.name = n));

    description: (*
        Find a user or users by real-world name.  If more than one
        is found, the output list is sorted by id.
    *);
end FindUserByName;
```

Figure 3.25: FindUserByName with basic postcondition

To test our new definition of `FindUserByName`, we run the same set of tests from Figure 3.23 against it. Since in all these examples the output records all have the given name field, in all cases we expect the result to be `{ true, true }` (see Figure 3.26).

```
"What happens if there are unique, unsorted records?"
{ true, true }
"What happens if there are unique, sorted records?"
{ true, true }
"What happens if there are non-unique, unsorted records?"
{ true, true }
"What happens if there are non-unique, sorted records?"
{ true, true }
```

Figure 3.26: FindUserByName basic validation results

3.3.8 FindUserByName: Formal Postcondition Logic with Sort Constraint

Although the `FindUserByName` definition in 3.3.7 ensures that all the records in the output collection have names that match the given name, the postcondition does not address the constraint that the matching records should be sorted alphabetically. Ultimately we would like the `FindUserByName` postcondition to reject validation operator invocations where the output collection is unsorted, which was not the case in Figure 3.26. To address this requirement, the `FindUserByName` definition in Figure 3.27 adds a sort constraint to the postcondition.

```
operation FindUserByName
    inputs: udb:UserDB, n:Name;
    outputs: url:UserRecord*;

    precondition: (* None yet. *);

    postcondition:
        (*
         * The output list consists of all records of the given name
         * in the input db.
         *)
        (forall (ur: UserRecord)
            (ur in url) iff (ur in udb) and (ur.name = n))

            and

        (*
         * The output list is sorted alphabetically by id
         *)
        (forall (i:integer | (i >= 1) and (i < #url))
           (url[i].id <= url[i+1].id));

    description: (*
        Find a user or users by real-world name.  If more than one
        is found, the output list is sorted by id.
    *);
end FindUserByName;
```

Figure 3.27: FMSL FindUserByName with sort constraint

When running the tests in Figure 3.23 against the updated `FindUserByName`, the unsorted cases' postconditions now fail with `{ true, false }` while the sorted cases' postconditions pass with `{ true, true }` (see output in Figure 3.28).

```
"What happens if there are unique, unsorted records?"
{ true, false }
"What happens if there are unique, sorted records?"
{ true, true }
"What happens if there are non-unique, unsorted records?"
{ true, false }
"What happens if there are non-unique, sorted records?"
{ true, true }
```

Figure 3.28: FindUserByName with sort constraint validation results

3.3.9 FindUserByName: Strengthened Postcondition

As we ask the question "is the postcondition strong enough?" we focus on the results of the last validation operator invocation from the tests in Figure 3.23. According to the output in Figure 3.28, the FindUserByName postcondition defined in 3.3.8 accepts an output collection where the matched record collection contains duplicates of the same record. Since we would like record uniqueness in the output collection, those results indicate that the postcondition is not yet strong enough. By examining the postcondition, we can see that the specification contains an easy-to-miss logic error: the sort constraint uses the '<=' operator to validate sortedness, and replacing it with the '<' operator would validated sortedness and uniqueness. See the listing in Figure 3.29 for an updated `FindUserByName` definition that utilizes the '<' operator in the sort constraint.

```
operation FindUserByName
    inputs: udb:UserDB, n:Name;
    outputs: url:UserRecord*;

    precondition: (* None yet. *);

    postcondition:
        (*
         * The output list consists of all records of the given name
         * in the input db.
         *)
        (forall (ur: UserRecord)
            (ur in url) iff (ur in udb) and (ur.name = n))

            and

        (*
         * The output list is sorted alphabetically by id
         *)
        (forall (i:integer | (i >= 1) and (i < #url))
           (url[i].id < url[i+1].id));

    description: (*
        Find a user or users by real-world name.  If more than one
        is found, the output list is sorted by id.
    *);
end FindUserByName;
```

Figure 3.29: FindUserByName with strengthened postcondition

As we've updated our `FindUserByName` postcondition, we re-run the validation tests against it. As we'd hoped, the output in Figure 3.30 shows that the `FindUserByName` postcondition now accepts only the output collection that contains matching, unique, sorted records; it rejects all the others.

```
"What happens if there are unique, unsorted records?"
{ true, false }
"What happens if there are unique, sorted records?"
{ true, true }
"What happens if there are non-unique, unsorted records?"
{ true, false }
"What happens if there are non-unique, sorted records?"
{ true, false }
```

Figure 3.30: FindUserByName strengthened validation results

3.3.10 FindUserByName: Postcondition with Auxiliary Functions

FMSL allows users to define functions that accept one or more input parameters and return an output value, which is set to the result of last expression evaluation in that function. Functions can be invoked from within preconditions and postconditions, and that abstraction can lead to clearer specifications. For example, the `FindUserByName` definition in Figure 3.31 abstracts out the concepts of `RecordsFound` and `SortedById` into their own respective functions that return a Boolean `true` or `false` result.

```
operation FindUserByName
    inputs: udb:UserDB, n:Name;
    outputs: url:UserRecord*;

    postcondition:
        RecordsFound(udb,n,url)
            and
        SortedById(url);

end FindUserByName;

function RecordsFound(udb:UserDB, n:Name, url:UserRecord*) =
    (*
     * The output list consists of all records of the given name in
     * the input db.
     *)
    (forall (ur' in url)
       (ur' in udb)
          and
       (ur'.name = n));

function SortedById(url:UserRecord*) =
    (*
     * The output list is sorted alphabetically by id.
     *)
        (if (#url > 1) then
            (forall (i in [1..(#url - 1)])
               url[i].id < url[i+1].id)
         else true);
```

Figure 3.31: FindUserByName with auxiliary functions

The `FindUserByName` definition in Figure 3.31 is functionally equivalent to the `FindUserByName` definition in Figure 3.29, although it's arguably more readable. Observe in Figure 3.32 that the validation tests yield the same results, so this postcondition that utilizes auxiliary functions is equally as strong as the postcondition from the example in Figure 3.29.

```
"What happens if there are unique, unsorted records?"
{ true, false }
"What happens if there are unique, sorted records?"
{ true, true }
"What happens if there are non-unique, unsorted records?"
{ true, false }
"What happens if there are non-unique, sorted records?"
{ true, false }
```

Figure 3.32: FindUserByName with aux. functions validation results

3.4 Additional Uses of Validation Invocations and Exploratory Expression Evaluation

An important part of refining a specification is translating user-level requirements, stated in English prose, into Boolean logic. Exploratory expression evaluation, including validation invocations, can be useful in this translation process.

The following are typical user-level requirements for an operation like adding a record to a database, i.e., the `AddUser` operation described in the previous section of the thesis:

- There is no user record in the input database with the same id as the record to be added; this is a *no duplicates requirement*.
- The id of an added user record cannot be empty and must be no more than 8 characters in length; this is an *id syntax constraint*.
- If the area code and phone number are present, they must be 3 digits and 7 digits respectively; these are *phone number format constraints*.

Figure 3.33 contains a sample specification of a flawed `AddUser` precondition. The intent of the precondition logic is to define these requirements. This sample characterizes the kind of logic oversights that have been observed regularly in students' initial efforts to translate user-level requirements from English prose into formal logic.

```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;

    precondition:
        (*
         * There is no user record in the input UserDB with the same
         * id as the record to be added.
         *)
        (not (ur in udb))

            and

        (*
         * The id of the given user record is not empty and 8
         * characters or less.
         *)
        (#(ur.id) <= 8)

            and

        (*
         * If the phone area code and number are present, they must
         * be 3 digits and 7 digits respectively.
         *)
        (#(ur.phone.area) = 3) and
        (#(ur.phone.num) = 7);


    postcondition: (* Same as above *);

end AddUser;
```

Figure 3.33: Flawed attempt at AddUser precondition

Figure 3.34 has corrected logic, for comparison purposes.

```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;

    precondition:
        (*
         * There is no user record in the input UserDB with the same
         * id as the record to be added.
         *)
        (not (exists (ur' in udb) ur'.id = ur.id))

            and

        (*
         * The id of the given user record is not empty and 8
         * characters or less.
         *)
        (ur.id != nil) and (#(ur.id) <= 8)

            and

        (*
         * If the phone area code and number are present, they must
         * be 3 digits and 7 digits respectively.
         *)
        (if (ur.phone.area != nil) then (#(ur.phone.area) = 3)) and
        (if (ur.phone.num != nil) then (#(ur.phone.num) = 7));


    postcondition: (* Same as above *);

end AddUser;
```

Figure 3.34: Improved AddUser precondition

As with any form of debugging, there are a variety of ways to test and correct flaws in logic. Validation invocations provide a useful tool that can help in the process. In the example at hand, each flaw can be revealed with a single, reasonably straightforward validation invocation.

The first flaw is the translation of the English requirement "*There is no user record in the input UserDB with the same id as the record to be added.*" The flawed versus correct versions of the logic are

```
(not (ur in udb))
```

versus

```
(not (exists (ur' in udb) ur'.id = ur.id))
```

This flaw can be detected with a validation condition that attempts to add a user record with the same id, but different name, to the database. E.g.,

```
val phone:PhoneNumber = {805, 5551212};
val email:EmailAddress = "pcorwin@calpoly.edu";
val ur:UserRecord = {"Corwin", "1", email, phone};
val ur_duplicate_id:UserRecord = {"Fisher", "1", email, phone};
val udb:UserDB = [];
val udb_added:UserDB = [ur];

> AddUser(udb_added, ur_duplicate_id) ?-> (udb_added);
```

The correct output of this validation is `{ false, nil }`, since the precondition should fail when trying to add a record with the same id value to a database containing a record with that id, i.e., "1". The flawed logic is not strong enough, since it does not check specifically for the id value of each extant record. This kind of error is typical with students who may be initially averse to using quantifiers, and will do their best to avoid their use. A validation counter-example can succinctly illustrate the problem with the flawed logic.

The second flaw is the translation of "*The id of the given user record is not empty and 8 characters or less.*" The flawed versus correct versions of the logic are:

```
        (#(ur.id) <= 8)
```

versus

```
        (ur.id != nil) and (#(ur.id) <= 8)
```

The problem here is that the length operator returns `0` for a `nil string` value. The following validation condition reveals the problem:

```
val ur_empty_id:UserRecord = {"Corwin", nil, email, phone};
> AddUser(udb, ur_empty_id) ?-> (udb);
```

The result of this evaluation should be `{ false, nil }`, since the precondition should fail if the id is `nil`. Here `nil` is the translation of "empty" in the prose statement of the requirement. The flawed logic precondition evaluates to `{ true, nil }`, since `#(ur.id) = 0` when `ur.id` is `nil`, and hence `0 <= 8` evaluates to `true`.

To some extent, this problem has to do with the specific semantics of FMSL. However, all formal specification languages have specific rules, and users of the languages must understand clearly what the rules are. Using validation invocations and additional exploratory evaluation can help a user develop such understanding.

Some additional exploration of this example could take the following form:

```
val empty_integer:integer = nil;
val empty_string:string = nil;
obj StringList = string*;
val empty_list:StringList = nil;

> #empty_integer;
> #empty_string;
> #empty_list;
```

where all three expressions evaluate to `0`. In the case of the integer value, the length operator is overloaded to evaluate to the number of integer digits. The rules illustrated here could be read in the FMSL users manual. However, the ability to explore interactively can be enlightening, as it is in the environments of interpretive and conversational programming languages.

The third and final flaw in Figure 3.33 is the translation of "*If the phone area code and number are present, they must be 3 digits and 7 digits respectively.*" The flawed and correct versions of the logic are:

```
(#(ur.phone.area) = 3) and
(#(ur.phone.num) = 7));
```

versus

```
(if (ur.phone.area != nil) then (#(ur.phone.area) = 3)) and
(if (ur.phone.num != nil) then (#(ur.phone.num) = 7));
```

The problem is revealed with the following validation invocation:

```
val ur_empty_phone:UserRecord = {"Corwin", "1", email, nil};

> AddUser(udb, ur_empty_phone)?->(udb);
```

The correct validation result is `{ true, nil }`, since the requirement allows the phone number components to be empty. Without the explicit check for this, the sub-expression `ur.phone.area` evaluates to `nil`. As explained in the previous example, the length operator applied to a `nil` value uniformly returns `0`. This means that `#(ur.phone.area)` returns `0`, which leads the precondition to evaluate to `false` instead of `true`.

Chapter 4 Overall System Design

Prior to the work of this thesis, the mechanized checking of an FMSL specification was limited to static syntax and semantic analysis. As with most programming language compilers, the output of the static analysis is empty, unless errors are detected. Figure 4.1 is a visual representation of the FMSL translator initial structure.

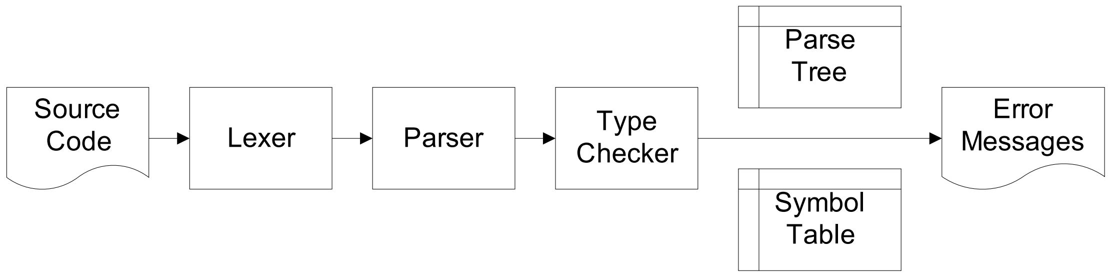

Figure 4.1: FMSL translator initial structure

The work for this thesis has added support for evaluating expressions through a functional interpreter. This functional interpreter implementation does not perturb the existing type-checking capabilities of FMSL. Per conventional compiler design principles, the interpreter implementation relies on the type-checker's results.

With the addition of functional interpretation, the execution output is no longer limited to type errors, but it also includes – where appropriate – results from expression evaluations and any run-time errors. Figure 4.2 is a visual representation of the revised FMSL translator structure and where the functional interpreter fits into the design. Functional interpreter implementation details are discussed in Chapter 5.

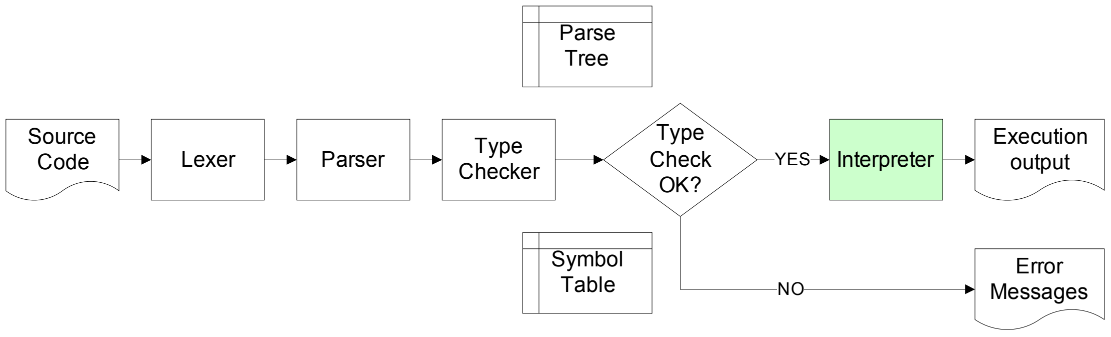

Figure 4.2: FMSL translator structure with interpreter

4.1 Execution of Preconditions and Postconditions

Preconditions and postconditions describe properties of the input and output values for an operation before and after execution of that operation. To meet the goal of allowing the user to execute a specification, a key capability is the ability to execute preconditions and postconditions.

To test the specification, the user creates a set of inputs and outputs for a given operation. By providing an operation name along with the inputs and outputs, connected by the validation operator, the user instructs FMSL to run these inputs and outputs against the operation's formal description. FMSL performs the execution and returns a meaningful response that consists of a pair of Boolean values that indicate results from precondition and postcondition evaluation.

It is important to note that precondition and postcondition evaluation can take place even when the operation is not constructively defined. A constructive function definition is denoted in FMSL in a manner comparable to functional programming languages. For example, the following is the constructive definition of an operation that checks if all the elements of an integer list are positive:

```
operation ConfirmPositiveConstructive(il:integer*) =
    if #il = 0 then true
    else il[1] > 0 and ConfirmPositiveConstructive(il[2:#il])
end;
```

This is a standard tail-recursive definition, with the idiom `[2:#il]` denoting the 2nd through last elements of a list. I.e., this is the FMSL analog of Lisp's cdr function.

For comparison, the following is the purely analytic definition of this function:

```
operation ConfirmPositiveAnalytic(il:integer*)
    pre: ;
    post: forall (i in il) i > 0;
end;
```

Comparative invocations of these two functions are the following:

```
(*
 * evaluates to false
 *)
> ConfirmPositiveConstructive([1,2,-3,4]);
(*
 * evaluates to {true,true}
 *
> ConfirmPositiveAnalytic([1,2,-3,4]) ?-> false;
```

Chapter 5 of the thesis discusses the details of how these two forms of invocation are implemented. The point of this comparative example has been to clarify the two forms of invocation for operations defined constructively versus analytically.

4.2 Quantifiers

In order to facilitate execution and evaluation of sufficiently useful preconditions and postconditions, FMSL includes support for quantifiers. Quantifiers are Boolean-valued expressions that evaluate a quantified sub-expression multiple times. FMSL supports universal and existential quantifiers, both bounded and unbounded. A bounded quantifier is a quantifier that iterates over a discrete set of values. An unbounded quantifier, on the other hand, iterates over values within a universe that is unbounded or, conceptually, infinitely large. Whereas a bounded quantifier might iterate through all the values within a fixed-size list, an unbounded quantifier might iterate over the set of all integers.

To evaluate a bounded quantifier is straightforward, and likewise the FMSL implementation approach was relatively clear-cut. Some mystery surrounded how to approach and implement something useful for unbounded quantifications as, so it turned out, an infinitely large value space can be rather difficult for computers to internalize. Although some tools and languages employ other approaches to handle this evaluation, for this thesis the decision was made to evaluate unbounded quantifications by treating them like a bounded case where the object values are supplied to the predicates from an incrementally built universe of values. Chapter 6 covers quantifier implementation details and provides a more in-depth discussion of approaches to dealing with unbounded quantifiers.

4.3 Value Universe for Unbounded Quantifier Evaluation

The Value Universe is a discrete pool of values, indexed by type, that supply meaningful values to unbounded quantifier predicates. When the FMSL interpreter encounters an unbounded quantifier, the interpreter iterates over all values of the type of interest to evaluate the predicate result. FMSL's Value Universe grows incrementally as values appear during specification execution, whether through purposeful Universe population operations or through normal specification execution. The Value Universe can contain values of any value type, ranging from simple atomic types to complex types defined as lists and tuples.

The decisions regarding when the FMSL interpreter should add values to the Value Universe were influenced by the importance of repeatability, i.e., that tests and executions should be repeatable so that running the same data through the same operations in the same order should consistently result in the same outputs. That in mind, the FMSL implementation adds values to the Value Universe primarily in contexts where the values cannot be mutated: let expressions, parameter binding, and list construction. Although value mutation is still possible, and so the Universe values can be changed in some cases, the FMSL user should understand that performing mutations can cause undesirable side effects that ripple throughout the universe and in normal execution. The bottom line is that non-functional value mutation may lead to unrepeatable testing results. This is consistent with the notion that value mutations are generally considered harmful in a functional environment.

All of the examples presented in Chapter 3 were fully functional, i.e., no value mutating operators were used. The only mutation-producing operator in FMSL is named set. Its semantics are comparable to Lisp's `setf` function, or mutations through references in ML. Chapter 5 discusses the use of set in FMSL. The rule for avoiding potentially harmful mutations in FMSL is very simple -- do not apply the set operator to anything but a plain variable.

By default, FMSL does not allow a value to be added into in the Universe if the Universe already contains that value (of a specific type). Although this decision adds up-front processing time when calculating whether to add a value to the Universe, it saves memory and cuts processing time during evaluation of unbounded quantifiers. To give the user additional control over whether the FMSL implementation should check for duplicates upon adding a value to the Universe, the user can enable Universe duplicates by appending the "`-universe-duplicates`" command-line parameter when invoking the FMSL translator.

4.3.1 Universe Implementation Details

The Value Universe is implemented as a block of memory where each memory slot is a pointer to a homogeneous list of values for a particular type. Figure 4.3 is a visual representation of the Value Universe structure.

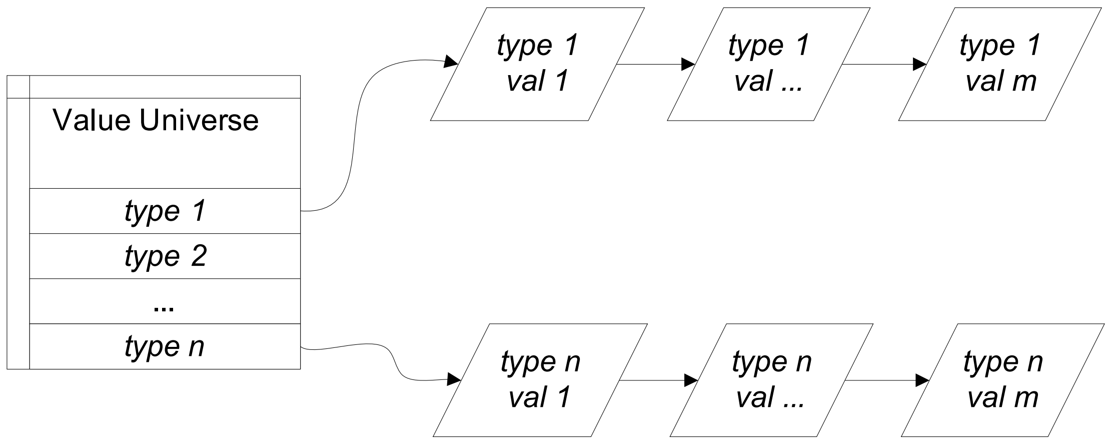

Figure 4.3: Value Universe structure

The FMSL code listing in Figure 4.4 declares a `Person` object type and contains two "`let`" expressions.

```
(*
 * Define the Person object type
 *)
object Person is
   components: firstName:string and
               lastName:string and
               age:integer;
end Person;

(*
 * Let p1 and p2 be specific Person values
 *)
> (let p1:Person = {"Alan", "Turing", 97}; true;);
> (let p2:Person = {"Arnold", "Schwarzenegger", 61}; true;);
```

Figure 4.4: Universe Person FMSL code listing

Upon encountering the "`let p2`" expression in this context, the FMSL implementation first looks up the `Person` memory slot in the Value Universe by hashing the `Person` type name to an index location. If there doesn't already exist such a slot, it assigns one and creates a value list of that type. Since a `Person` slot already exists in the Universe (see Figure 4.5:1) and since we are not allowing duplicates, the FMSL implementation accesses the list of `Person` values and verifies that the value represented by `p2` does not already exist in the Universe. Since it does not already exist in the Universe, the FMSL implementation adds the value represented by `p2` to the end of the `Person` list (see Figure 4.5:2).

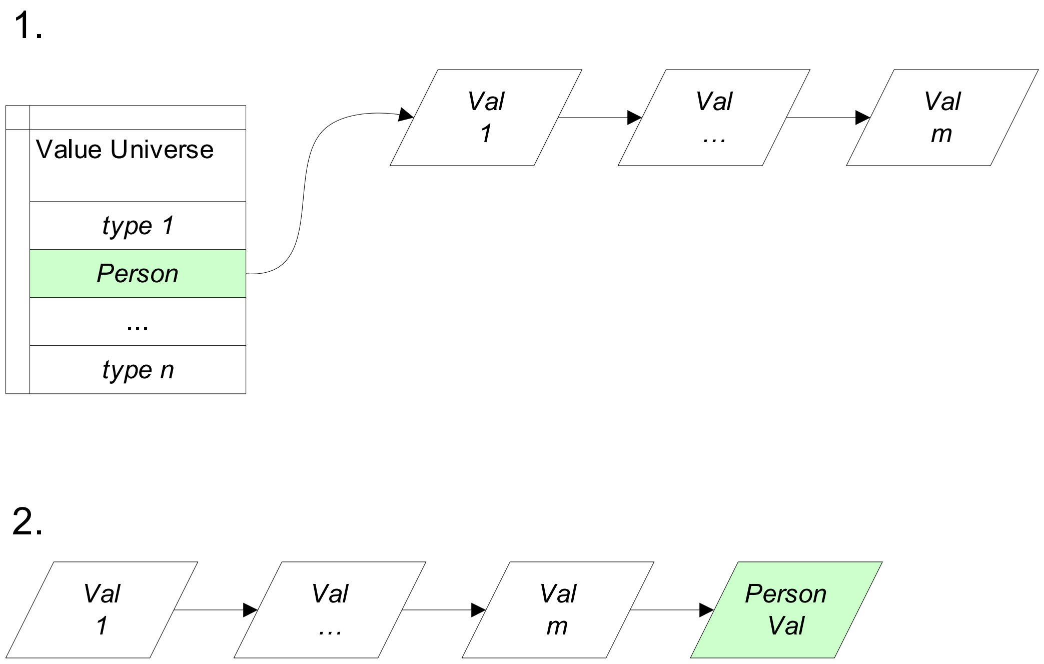

Figure 4.5: Value Universe Add Person Value

By executing the code in Figure 4.4 from the command-line with the `-dump-universe` parameter we can see a listing of what's contained in the Value Universe at the end of specification execution. Figure 4.6 has the FMSL output, which shows that the Value Universe contains both `Person` values, after executing the code in Figure 4.4 with the "`-dump-universe`" command line option.

```
true
true
Value Universe contains: <
Person: [ { "Alan", "Turing", 97 }, { "Arnold", "Schwarzenegger", 61
} ]
>
```

Figure 4.6: FMSL output after lets

Chapter 5 The Functional Interpreter

The functional interpreter goes beyond type checking and allows for actual expression evaluation, maintains internal storage for objects of various types, supports operation invocation, validation operator invocation, and more within a specification.

5.1 Basic Object Types and Operator Interpretation

FMSL supports the following basic atomic types: `boolean`, `integer`, `real`, and `string`. `boolean` objects hold values of `true` or `false`. `integer` objects hold non-fraction numbers. `real` objects hold double-precision decimal numbers. `string` objects hold sequences of characters or the empty string. FMSL has a uniform `nil` value, which symbolizes the concept of "no value" and can be the value of any object. FMSL also provides built-in support for a collection of operators that act on these basic types.

5.1.1 Basic Object Type Implementation

All object values in FMSL are stored internally within a common structure, called a `ValueStruct`, which gives the interpreter access to meta-information about the value. A `ValueStruct` is a C structure that stores all the information, shown in Table 5.1.

Internal Name	Description
`LorR`	whether the underlying value is an L- or R-value
`tag`	the general type of the value
`type`	the full type structure
`size`	the type size, which can be number of elements or number of bytes
`val`	the value's actual byte representation in memory

Table 5.1: Contents of ValueStruct

Internally the C code accesses and manipulates the object's value in memory by referencing the `val` field within the `ValueStruct`. The `val` field is a C `union` that can represent any FMSL value (or a pointer to the FMSL value), as illustrated in Figure 5.1.

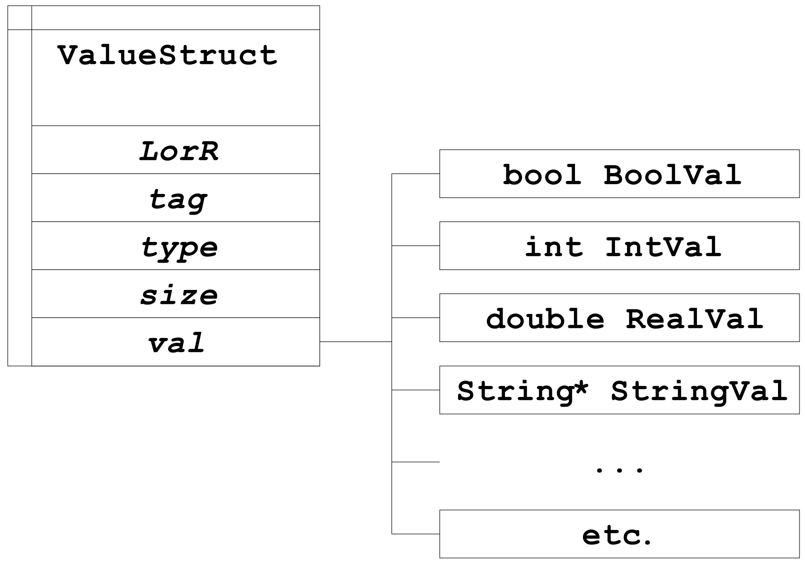

Figure 5.1: ValueStruct structure with val union

5.1.2 Operator Descriptions

FMSL provides built-in support for a collection of operators on these basic types. For descriptions of the built-in operators available for `boolean`, number (`integer` and `real`), and `string` typed objects see Table 5.2, Table 5.3, and Table 5.4, respectively.

Operator	Description	Returns
`not`	negation	`boolean`
`and`	conjunction	`boolean`
`or`	disjunction	`boolean`
`xor`	exclusive disjunction	`boolean`
`=>`	implication	`boolean`
`<=>`	two-way implication; if and only if	`boolean`
`if b1 then b2` where `b1`, `b2` are Boolean expressions	conditional	`boolean`
`if b1 then b2 else b3` where `b1`, `b2`, `b3` are Boolean expressions	conditional with else	`boolean`

Table 5.2: Operators on booleans

Operator	Description	Returns
`+`	Addition	`integer` or `real`
`-`	Subtraction	`integer` or `real`
`*`	multiplication	`integer` or `real`
`/`	Division	`integer` or `real`
`mod`	Modulus	`integer`
`+ (unary)`	returns 1*the number	`integer` or `real`
`- (unary)`	returns -1*the number	`integer` or `real`
`=`	Equality	`boolean`
`!=`	Inequality	`boolean`
`>`	greater than	`boolean`
`<`	less than	`boolean`
`>=`	greater than or equal to	`boolean`
`<=`	less than or equal to	`boolean`

Table 5.3: Operators on numbers

Operator	Description	Returns
`=`	equality	`boolean`
`!=`	inequality	`boolean`
`#`	string length	`integer`
`in`	membership test	`boolean`
`+`	concatenation	`string`
`[n]`	single character selection	`string`
`[m .. n]`	range / substring selection	`string`

Table 5.4: Operators on strings

5.1.3 Operator Implementations

When the interpreter is tasked with evaluating the result of a simple expression that involves an operator, the interpreter runs through a series of steps to determine what it's supposed to do. Those steps involve first determining the structure of the expression (does the expression have one operand? Two operands? Three operands? No operands at all? etc.). The interpreter then determines which specific operator is being called. Once it has established the structure and operator, the interpreter calls the proper C function with the operand(s).

A straightforward example traces the execution path of the binary division operator (`/`). Note that the term *binary operator* here means that there are two operands, not that the operands are represented in binary format. In the listing in Figure 5.2, the last line of FMSL code tells the interpreter to perform division where the operands are of type `real`.

Code listing:

```
(*
 * Declare and assign values to x, y
 *)
val x:real = 3.141592654;
val y:real = 2.718281828;

(*
 * Evaluate x divided by y and output the result
 *)
> x / y;
```

Output:

```
1.15573
```

Figure 5.2: FMSL division example listing and output

The interpreter processes the last expression by following these steps:

1. Determine that the expression involves a binary operator
2. Determine the operator (`/`)
3. Call and return the result of the function that performs the division (`doRealDiv`), and pass as parameters the `ValueStruct`s corresponding to the `x` and `y` operands

The C code for evaluating the division appears in Figure 5.3.

```
ValueStruct doRealDiv(ValueStruct v1, ValueStruct v2, nodep t) {

    /*
     * Propagate null value if either is operand is null.
     */
    if ((v1 == null) or (v2 == null))
        return null;

    /*
     * Handled the overload for real or integer operands.
     */
    switch (v1->tag) {
        case RealTag:
            if (v2->tag == IntTag) {
                if (v2->val.IntVal == 0) {
                    free(v2);
                    lerror(t, "Divide by zero.\n");
                    return null;
                }
                v1->val.RealVal = v1->val.RealVal / v2->val.IntVal;
            }
            else {
                if (v2->val.RealVal == 0) {
                    free(v2);
                    lerror(t, "Divide by zero.\n");
                }
                v1->val.RealVal = v1->val.RealVal / v2->val.RealVal;
            }
            free(v2);
            return v1;
        case IntTag:
            if (v2->tag == RealTag) {
                if (v2->val.RealVal == 0) {
                    free(v2);
                    lerror(t, "Divide by zero.\n");
                    return null;
                }
                v1->val.RealVal = v1->val.IntVal / v2->val.RealVal;
                v1->tag = RealTag;
            }
            else {
                if (v2->val.IntVal == 0) {
                    free(v2);
                    lerror(t, "Divide by zero.\n");
                    return null;
                }
                v1->val.IntVal = v1->val.IntVal / v2->val.IntVal;
            }
            free(v2);
            return v1;
    }
}
```

Figure 5.3: doRealDiv implementation

Tracing through the code, `doRealDiv` inspects the `ValueStruct`'s `tag` field and establishes that we're dealing with parameters of type `real`. It's important to make this determination since, as indicated in Table 5.3, the `/` operator also can be used on `integer` operands or mixed `real` and `integer` operands.

It is worth noting that the only runtime type checking that is necessary is for overloaded operators, such as arithmetic. The static type checker ensures that arithmetic operators are never applied to non-numeric operands. Doing so results in a type checking error, which precludes any subsequent expression evaluation. From a type-theoretic standpoint, FMSL is a 100% statically typed language. The use of types at runtime is an overloading implementation technique. Conceptually, there are separate versions of each overloaded operator, for each combination of operand types.

There is a third parameter in `doRealDiv`: `nodep t`. Within `doRealDiv`, `t` is referenced to help describe the location of a runtime error if one occurs, which in this function could happen since we might see an attempt to divide by zero. Since we're not dividing by zero in this example, the C code performs the division and assigns the result. Finally, `doRealDiv` returns `v1`, the `ValueStruct` that contains the result.

The FMSL interpreter evaluates all the expressions that contain FMSL operators in a fashion similar to the example described above.

5.2 Complex Structures

In addition to the basic object types (`boolean`, `integer`, `real`, and `string`), FMSL supports structured types with lists and tuples. FMSL lists are homogeneous data structures that hold zero or more object values, analogous to an array with no predetermined, fixed size. FMSL tuples are heterogeneous data structures that hold a fixed number of components of specific object types, similar to a C `struct`. See Table 5.5 and Table 5.6 for details on list and tuple operators, respectively.

Operator	Description	Returns
`=`	equality	`boolean`
`!=`	inequality	`boolean`
`in`	membership	`boolean`
`#`	element count	`integer`
`+`	concatenation	`list type`
`-`	deletion from list	`list type`
`[n]`	element selection	`list type`
`[m .. n]`	range selection	`list type`

Table 5.5: Operators on lists

Operator	Description	Returns
`=`	Equality	`boolean`
`!=`	inequality	`boolean`
`.`	field access	`any field type`

Table 5.6: Operators on tuples

The following FMSL code declares an object type called `IntegerList`, which is a list of integers.

```
object IntegerList = integer*;
```

The FMSL code in Figure 5.4 declares an object type called `Person`, which contains several fields that together help describe a person.

```
object Person
   components: firstName:string and
               lastName:string and
               age:integer;
end Person;
```

Figure 5.4: Person object type definition

5.2.1 List and List Operator Implementation

Internally, an FMSL list is implemented as a `ValueStruct` where the `val` union data item is a pointer to a C list structure called `ListVal`. `ListVal` is a `ListStruct` (see Figure 5.5), which is a C `struct` that contains a linked list of generic list elements and other list metadata such as list size.

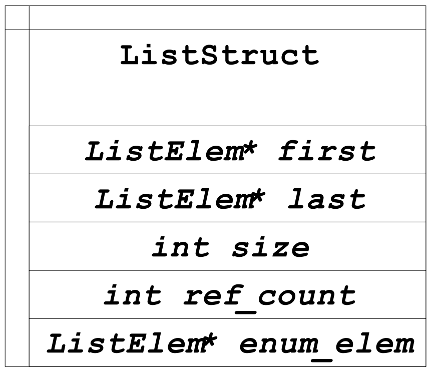

Figure 5.5: ListStruct definition

The FMSL code snippet in Figure 5.6 below defines an `IntegerList` object type and creates an `IntegerList` instantiation called `intlist`.

Code listing:

```
(*
 * Declare the IntegerList type.
 *)
object IntegerList = integer*;

(*
 * Declare an intlist value and assign a collection of integers.
 *)
val intlist:IntegerList = [1,1,2,3,5,3+5];

> intlist;
```

Output:

```
[ 1, 1, 2, 3, 5, 8 ]
```

Figure 5.6: FMSL IntegerList initialization

To construct an FMSL list, the FMSL implementation first builds a `ValueStruct` to hold a list of values of the specified element type. It then iterates through and evaluates each item in the expression list of elements, which was assembled by the parser and the type checker. The result of each expression evaluation is placed at the end of the list, and finally the list constructor function returns the newly assembled list `ValueStruct`.

Note the importance of evaluating expressions when creating the internal representations of the list elements: in the code listing in Figure 5.6, the last element of the list of integers is `3+5`. During list construction, the C implementation evaluates that expression – i.e., in this case it performs the addition – and stores the result (`8`) at the end of the list. See Figure 5.7 for the `doListConstructor` C code that performs list construction.

```
ValueStruct doListConstructor(t)
    nodep t;
{
    TypeStruct type             /* Type of the array */
              = t->header.attachment.type;
    ValueStruct rtn,            /* Return val temp */
           rval;                /* Value of each elem expr */
    nodep e;                    /* Working expr pointer */

    /* if we arrive here and type is undefined, return nil now */
    if (!type)
    {
        rtn = MakeVal(RVAL, NilType);
        return rtn;
    }

    rtn = MakeVal(RVAL, type);
    rtn->val.ListVal = NewList();

    for (e = t->components.expr.left_operand; e;
                e = e->components.exprlist.next) {

        /*
         * Evaluate the value expressions along the way and
         * assign to a memory slot.
         */
        rval = interpExpr(e->components.exprlist.expr);
        PutList(rtn->val.ListVal, (ListElemData*)rval);

        /*
         * Add constructed list elements to the universe of the
         * list's base type.
         */
        if (isIdentType(basetype)) {
            UniverseAddValue1(
                basetype->components.type.kind.ident.type->
                components.atom.val.text, rval);
        }
    }

    return rtn;
}
```

Figure 5.7: doListConstructor implementation

An example list operator implementation that is notable is the range selection operator. The range selection or list-slice operator returns a list of subcomponents.

For example, the last line of the code listing in Figure 5.8 returns a list that consists of components at indexes 3, 4, and 5 within the list.

Code listing:

```
(*
 * Declare the IntegerList type
 *)
object IntegerList = integer*;

(*
 * Declare an intlist value
 *)
val intlist:IntegerList = [1,1,2,3,5,3+5];

(*
 * Select the subcomponents at indexes 3, 4, and 5.
 *)
> intlist[3..5];
```

Output:

```
 [ 2, 3, 5 ]
```

Figure 5.8: FMSL list selection example

When the interpreter is tasked with evaluating a list selection expression, the interpreter first establishes that the expression of interest has three operands: the list, the lower bound of the range selection, and the upper bound of the range selection. The interpreter then evaluates each of the three operands and passes them as parameters to the `doArraySliceRef` function, seen in Figure 5.9. Next, the code determines that the `v1` parameter is an FMSL list[2] and so the C code initializes `result` as an empty list. By looping from the lower bound value `v2` to the upper bound value `v3`, one at a time the code accesses the selected subcomponents of `v1`

[2] Recall that according to Table 5.4, the selection operator also applies to `string` objects and so the code here must determine whether `v1` is a `string` or a list.

and copies (or puts) them into `result`. Finally `doArraySliceRef` returns `result`, which is the `ValueStruct` that contains the sub-list.

```
ValueStruct doArraySliceRef(v1, v2, v3)
    ValueStruct v1;
    ValueStruct v2;
    ValueStruct v3;
{
    ValueStruct result;
    int i;

    /* start building the new list */
    result = MakeVal(RVAL, v1->type);
    if (v1->tag == ListTag) {
        result->val.ListVal = NewList();

        /*
         * loop through from lower .. upper and add the elements
         * to result.
         */
        for (i = v2->val.IntVal; i <= v3->val.IntVal; i++) {
            PutList(result->val.ListVal,
                    GetListNth(v1->val.ListVal, i));
        }
    }
    else if (v1->tag == StringTag) {
        result->val.StringVal =
            (String *)SubString(v1->val.StringVal,
                                v2->val.IntVal,
                                v3->val.IntVal);
    }
    return result;
} /* end function doArraySliceRef */
```

Figure 5.9: doArraySliceRef implementation

5.2.2 Tuple and Tuple Operator Implementation

Internally, an FMSL tuple is implemented as a `ValueStruct` where the `val` union data item is a pointer to a C list structure called `StructVal`. Like `ListVal`, `StructVal` also is implemented as a `ListStruct` (Figure 5.5);

however, unlike `ListVal`, each item in the `StructVal` list corresponds to a field within the FMSL tuple. Figure 5.10 has an FMSL code listing that declares a variable of type Person, initializes that variable through tuple construction and then accesses a field within the tuple.

Code listing:

```
(*
 * Declare p, a person variable
 *)
val p:Person = {"Arnold", "Schwarzenegger", 61};

(*
 * Access p's last name field
 *)
> p.lastName;
```

Output:

```
"Schwarzenegger"
```

Figure 5.10: Person tuple FMSL code listing

The strategy for constructing an FMSL tuple in C is similar to the strategy for constructing lists, although there are some differences. To construct a tuple, `doTupleConstructor` (see Figure 5.11) first checks to make sure it has field values to instantiate and add to the tuple. It then creates the `rtn` tuple `ValueStruct` and initializes it with the correct type. Internally, the field order within a tuple is relevant and so in order `doTupleConstructor` loops through evaluating field expression values and placing each result in `rtn`'s `StructVal` field. Finally, `doTupleConstructor` returns the `rtn` tuple `ValueStruct`.

```
ValueStruct doTupleConstructor(t)
    nodep t;
{
    ValueStruct rtn,
        rval;
    nodep e;
    TypeStruct tupleType;

    /* if this isn't going to work, return nil now */
    if (!t->components.unop.operand)
    {
        rtn = MakeVal(RVAL, NilType);
        return rtn;
    }

    /* get the tuple type and initialize it */
    tupleType =
        t->components.unop.operand->components.exprlist.type;
    rtn = MakeTupleVal(RVAL, tupleType);
    rtn->val.StructVal = NewList();

    /*
     * loop through the tuple fields and add each one
     * as a list element.
     */
    for (e = t->components.unop.operand;
                e;
                e = e->components.exprlist.next) {
        rval = interpExpr(e->components.exprlist.expr);
        PutList(rtn->val.StructVal, (ListElemData*)rval);
    }

    return rtn;
}
```

Figure 5.11: doTupleConstructor implementation

An example operator on tuple objects is the field access operator (“`.`”), which is used as follows: `<tuple object>.<field name>`. The field access operator returns the value contained in the tuple within the stated field, much like the way `struct` access works in C. The last line of the code listing example in Figure 5.10 demonstrates field access.

To evaluate tuple field access, the FMSL interpreter first determines that it is processing a binary operator with two operands: the tuple and the field within the tuple. The interpreter then calculates the memory location of the tuple and calls `RecordRef` (see Figure 5.12), passing in the memory location of the tuple and information about the the field to be accessed. `RecordRef` first determines the position of the field within the list of fields for this tuple. In our field access example from Figure 5.10 we're accessing a field via a field name ("`lastName`"), and so `RecordRef` accesses the tuple's symbol table to look up the field's ordinal position from the textual field name. `RecordRef` then gets the `ValueStruct` stored at that field position within the tuple `ValueStruct`'s `StructVal`. Next, `RecordRef` allocates memory for `newDesig`, a new `ValueStruct` pointer. Finally, `RecordRef` makes `newDesig` point to the field value of interest and returns it.

```
ValueStruct RecordRef(desig, field)
    ValueStruct desig;  /* L-value for the left operand. */
    nodep field;        /* Ident for the right operand. */
{
    ValueStruct valueField,
        tuple,
        newDesig;
    SymtabEntry *f;
    int n;
    TypeStruct type = ResolveIdentType(desig->type, null, false),
        fieldType;

    /*
     * If the field is represented by a field name, look up
     * the field name in the symbol table to get the position
     * within the list.
     *
     * Otherwise we have an anonymous access into a tuple, so
     * we already have the numbered position.
     *
     * In either case we need to get the field type.
     */
    if (field->header.name == Yident) {
        f = LookupIn(field->components.atom.val.text,
                     type->components.type.kind.record.fieldstab);
        fieldType = ResolveIdentType(f->Type, null, false);
    }
    else {
        f = null;
        n = field->components.atom.val.integer;
        fieldType = ResolveIdentType(
          GetNthField(type->components.type.kind.record.fields, n)->
            components.decl.kind.field.type,
          null, false);
    }

    /*
     * coming in, desig->LVal should point to the ValueStruct
     * of the struct.
     */
    tuple = (ValueStruct)*(desig->val.LVal);

    /* Note: Our lists are 1-indexed */
    valueField = (ValueStruct)GetListNth(tuple->val.StructVal,
        f ? f->Info.Var.Offset + 1 : n);
    /*
     * if we have valueField filled in, use its type.
     * Otherwise, use the fieldType.
     */
    if (!valueField) {
        newDesig = MakeVal(LVAL, fieldType);
    }
    else {
        newDesig = MakeVal(LVAL, valueField->type);
    }

    /*
     * Allocate some storage for the field ValueStruct pointer
     * and put field value there.
     */
    newDesig->val.LVal = (ValueStruct *) malloc(sizeof(Value **));
    *(newDesig->val.LVal) = valueField;

    return newDesig;
}
```

Figure 5.12: RecordRef implementation

5.3 Operation Invocation

As outlined in Chapter 4, FMSL supports the definition of computation operations. These have the standard semantics of procedural abstractions definable in almost all programming languages. Parameter passing is strictly call-by-value. When operations have no mutating set expressions, they are side-effect free. This is the case for all of the examples presented in the thesis. Figure 5.13 has an example of a simple FMSL operation called `Cube`, which returns the result of cubing the integer input parameter.

```
Code listing:

operation Cube (x:integer) = x * x * x;

> Cube(2);
> Cube(5);

Output:

8
125
```

Figure 5.13: Cube operation FMSL listing

The FMSL implementation performs operation invocations by first pushing an activation record onto the stack. The implementation then evaluates each of the input parameters and binds the corresponding values to the proper memory locations according to the formal parameter names. After performing the parameter binding, the implementation pushes the local symbol table to the top of the symbol table stack and executes the operation body. The operation result is equal to the result of the last expression in the operation, which gets saved off before popping the activation record and returning the symbol table to its original state. Finally, the implementation returns the `ValueStruct` operation result.

5.4 Operation Validation through the Validation Operator

FMSL's validation operator is designed to support incremental testing of a specification. Whereas a more standard operation invocation involves passing only input parameters to an operation, the validation operator accepts an operation name, input parameters, and output parameters. Generically, the validation operator usage is:

```
operation_name(input argument list) ?-> (output argument list)
```

The in arguments are values that map to the operation's input parameters and the out arguments map to the operation's output parameters. The result of a validation operator invocation is a tuple that contains two `boolean` values: the first expresses the result of the precondition evaluation and the second expresses the result of the postcondition evaluation. See Table 5.7 for a list of potential value combinations within the returned tuple.

Tuple Returned	**Indication**
`{ nil, nil }`	execution failure in the precondition; postcondition evaluation not attempted
`{ false, nil }`	precondition evaluation failed; postcondition evaluation not attempted
`{ true, nil }`	precondition evaluation passed; no postcondition specified or there was an execution failure in the postcondition
`{ true, false }`	Precondition evaluation passed; postcondition evaluation failed
`{ true, true }`	Both precondition and postcondition evaluation passed

Table 5.7: Validation result values

The "execution failure" referred to in Table 5.7 results from an expression returning a nil value. Genuine failures include fatal arithmetic errors, such as division by zero; list index out-of-bounds; or access to uninitialized tuple fields. A complete discussion of such errors is in the FMSL reference manual [29].

An expression can also produce a `nil` result on purpose, for example an operation that returns a `nil` value to indicate that no meaningful value was computed. Conceptually, an evaluation result of `nil` means "undefined". Whether such is the result of a specific error or purposeful computation is based on the context of the evaluation. In this sense, an evaluation result of `nil` represents an abstract representation of undefinedness. This is comparable to the evaluation of null pointer values in programming languages, where null may be the result of a computational error, or used to represent a purposeful result.

By executing a sequence of validation operator invocations with varying, thoughtfully selected values for the input and output arguments, the user can gain additional confidence in both the test data and the specification or discover errors in the data or the specification. Examples of such value selections were presented in Chapter 3. In the event that the validation operator invocation returns a tuple with both values of true, the test inputs and outputs agreed with both the operation's precondition and postcondition. In the event where there were failures along the way, the user might see other meaningful combinations of `boolean` values in the result tuple.

As outlined in Table 5.8, if the first tuple field is false then the test values for the inputs were invalid or the precondition was specified incorrectly. If the first tuple field is true and the second tuple field is false, the test input values were valid and the output values were invalid or the postcondition was specified incorrectly. The first occurrence of a nil value in the returned tuple could signify that there is a problem with the specification of the precondition or postcondition.

Tuple Returned	Significance
`{ nil, nil }`	The precondition may be specified incorrectly since a run-time / execution error was detected during precondition execution
`{ false, nil }`	Test values for inputs were invalid or the precondition was specified incorrectly
`{ true, nil }`	Test values for inputs were valid, but the postcondition either wasn't specified or it may be specified incorrectly since a run-time / execution error was detected during postcondition execution
`{ true, false }`	Test values for inputs were valid, but the output values were invalid or the postcondition was specified incorrectly
`{ true, true }`	Test values for both inputs and outputs agreed with both the precondition and postcondition

Table 5.8: Validation result significance

When choosing test data for inputs and outputs in a validation operator invocation, the user may want to create and run some test data inputs and outputs against an operation such that the result is known to *not* be `{ true, true }`. While some symbolic model checking tools initialize input fields only to values that adhere to the precondition [44], with FMSL's validation operator the user also can get additional, helpful assurance that there is an absence of unintended behavior instead of just "verify[ing] the existence of a particular feature" [45]. Through comprehensive test data selection and by observing the feedback FMSL provides after performing a validation operator invocation, the user can utilize FMSL to help detect specification and test data errors.

Chapter 6 Quantifier Execution

FMSL supports both bounded and unbounded universal (`forall`) and existential (`exists`) forms of quantification. Table 6.1 has a summary of the FMSL bounded and unbounded quantifier syntax.

Syntax	**Quantifier Type**	**Reads Like …**
forall (*x* in *S*) *p*	bounded	for all values *x* in list *S*, *p* is true
forall (*x*:*t*) *p*	unbounded	for all values *x* of type *t*, *p* is true
forall (*x*:*t* | *p1*) *p2*	unbounded	for all values *x* of type *t* such that *p1* is true, *p2* is true
exists (*x* in *S*) *p*	bounded	there exists an *x* in list *S* such that *p* is true
exists (*x*:*t*) *p*	unbounded	there exists an *x* of type *t* such that *p* is true
exists (x:*t* | *p1*) *p2*	unbounded	there exists an *x* of type *t* such that *p1* is true and *p2* is true

Table 6.1: FMSL quantifier syntax

A bounded quantifier evaluates over a discrete universe of values, as seen in Figure 6.1. In the example, the `forall` ranges over all five elements [1, 1, 2, 3, 5] that make up `IntList list`. Since each `integer` element within `list` is greater than zero in the example in Figure 6.1, the bounded universal quantifier evaluates to `true`.

```
(*
 * Declare an IntList object type and an IntList value
 *)
obj IntList = integer*;
val list:IntList = [ 1, 1, 2, 3, 5 ];

(*
 * Evaluate: all the integer elements within list are positive.
 *)
> forall (i in list) i > 0;          -- evaluates to true
```

Figure 6.1: Example of a bounded quantifier in FMSL

In the example in Figure 6.2, unlike in Figure 6.1, it is not immediately clear how the interpreter should evaluate the unbounded universal quantifier since the `Person` space is a potentially infinitely-large universe.

```
obj Person = name:Name and age:Age;
obj Name = string;
obj Age = integer;
> forall (p:Person) p.age >= 21;
```

Figure 6.2: Example of an unbounded quantifier in FMSL

For this thesis, FMSL evaluates unbounded quantifiers by iterating through an incrementally built universe of values and evaluating the predicate for each value. Other methods were considered, and Section 6.1 discusses quantifier execution approaches found in other formal methods tools. Section 6.2 lays out several quantifier examples and describes their implementations.

6.1 Methods of Quantifier Execution

Formal methods tools and methods that support specification execution take different approaches to handling unbounded quantifiers. For example, Aslantest [24], Jahob [64], and executable Z [35] all handle unbounded quantifiers in different ways.

The symbolic execution tool for Aslan, Aslantest [24], attempts to automatically evaluate all Boolean expressions contained within a specification. When Aslantest encounters a Boolean expression – like an unbounded quantifier – that it cannot automatically reduce to a simple true or false, it suspends specification execution and calls upon the user to play the role of the simplifier. The user then must enter the Boolean value result of the expression that could not be reduced. The Aslantest tool takes record of the user response and then execution continues.

The Jahob verification system [64] proves correctness properties by generating condition formulas – that together show that a program respects preconditions, postconditions, and invariants – and then proving them using theorem proving techniques. When Jahob encounters an unbounded quantifier, the Jahob user is encouraged to utilize Jahob's *pickAny* construct that makes the variable involved in the unbounded quantifier predicate appear to be a specification variable with an arbitrary value. The Jahob user also can state lemmas that involve the variable of interest, which together with the *pickAny* construct effectively remove the unbounded quantifier evaluation and thus simplify the theorem proving task.

Z is a "formal notation which aims to support, besides others, the specification of early requirements" [35]. In [35] Grieskamp et al. detail their experiments with use cases described in an executable form of Z. When they describe constraints that involve unbounded universal quantifiers then their execution or computation diverges, or in other words unbounded universal quantification is a "source of non-executability" in their setting. To avoid the problem of non-executability, their solution involves generally treating these constraints as compiler assumptions.

6.2 Unbounded Quantifier Execution in FMSL

What follows is a description of the implementation approaches taken to evaluate unbounded universal, existential, and universal with suchthat ("`|`") quantifiers.

6.2.1 Example: forall

The code listing in Figure 6.3 demonstrates a `forall` example.

```
(*
 * Perform lets with p1, p2 to put them in the Universe
 *)
> (let p1:Person = {"Alan", "Turing", 97};);
> (let p2:Person = {"Arnold", "Schwarzenegger", 61};);

> "Expected: true";
> forall (p:Person) p.lastName != nil;

(*
 * Since p3, with a nil last name, has been introduced
 * then we expect false below.
 *)
> (let p3:Person = {"Charles", nil, 218};);
> "Expected: false";
> forall (p:Person) p.lastName != nil;
```

Figure 6.3: FMSL forall example code listing

To populate the Universe with `Person` values, the code uses some `let` expressions that assign `Person` values to identifiers (`p1` and `p2`). We expect the first `forall` example to evaluate to `true` since at this point all `Person` values in the Universe have defined `lastName` fields.

To evaluate the `forall` expression, the FMSL interpreter first identifies that `p` is of object type `Person`. It then hashes the `Person` type name to locate the slot in the Value Universe where `Person` values should be found (see Figure 6.4:1). After discovering that there exist `Person` values in the Universe, the FMSL interpreter accesses that list of `Person` values (see Figure 6.4:2). The FMSL interpreter then iterates through each `Person` value in the list, temporarily assigning the current `Person` value to `p` in the local symbol table. At each step along the way, the FMSL interpreter evaluates the predicate (`p.lastName != nil`) and essentially ANDs the results together (see Figure 6.4:3) to arrive at the final evaluation result.

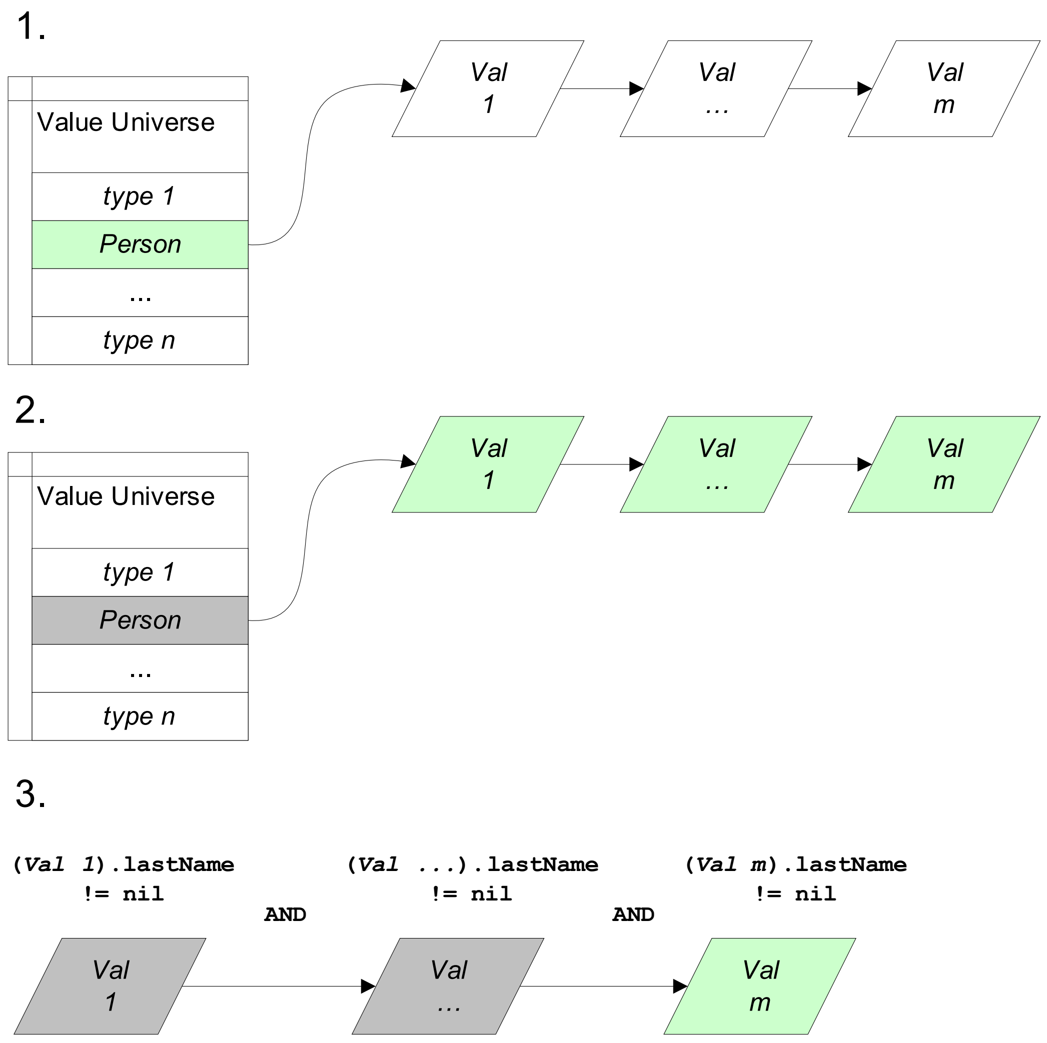

Figure 6.4: Forall example universe access

Note that in the example in Figure 6.3, we expect the first `forall` expression to evaluate to `true` and we expect the second `forall` expression to evaluate to `false`. Just prior to executing the second `forall` in the example, the FMSL interpreter processes the `let p3` expression where `p3` is assigned a `Person` value with the `lastName` field set to `nil`. Since the FMSL interpreter picks up that `p3` `Person` value and places it in the `Person` pool of values in the Value Universe, the second `forall` expression should evaluate to `false`. This expectation turns out to be correct, as evidenced by the output in Figure 6.5 below.

```
{ "Alan", "Turing", 97 }
{ "Arnold", "Schwarzenegger", 61 }
"Expected: true"
true
{ "Charles", nil, 218 }
"Expected: false"
false
```

Figure 6.5: FMSL forall example output

6.2.2 Example: exists

The code listing in Figure 6.6 demonstrates an `exists` example.

```
(*
 * Perform lets with p1, p2 to put them in the Universe
 *)
> (let p1:Person = {"Alan", "Turing", 97};);
> (let p2:Person = {"Arnold", "Schwarzenegger", 61};);

> "Expected: false";
> exists (p:Person) p.lastName = nil;

(*
 * Since p3, with a nil last name, has been introduced
 * then we expect true below.
 *)
> (let p3:Person = {"Charles", nil, 218};);
> "Expected: true";
> exists (p:Person) p.lastName = nil;
```

Figure 6.6: FMSL exists example code listing

The example in Figure 6.6 starts out the same as in Figure 6.3 where the Universe gets populated with some `Person` values. Where it is different are the `exists` quantifiers instead of `forall` quantifiers.

As when evaluating a `forall` quantifier, to evaluate the `exists` expression the FMSL interpreter first identifies that `p` is of object type `Person`. It then hashes the `Person` type name to locate the slot in the Value Universe where `Person` values should be found (see Figure 6.7:1). After discovering that there exist `Person` values in the Universe, the FMSL interpreter accesses that list of `Person` values (see Figure 6.7:2). The FMSL interpreter then iterates through each `Person` value in the list, temporarily assigning the current `Person` value to `p` in the local symbol table. At each stop along the way, the FMSL interpreter evaluates the predicate (`p.lastName = nil`) and ORs the results together (see Figure 6.7:3) to arrive at the final evaluation result.

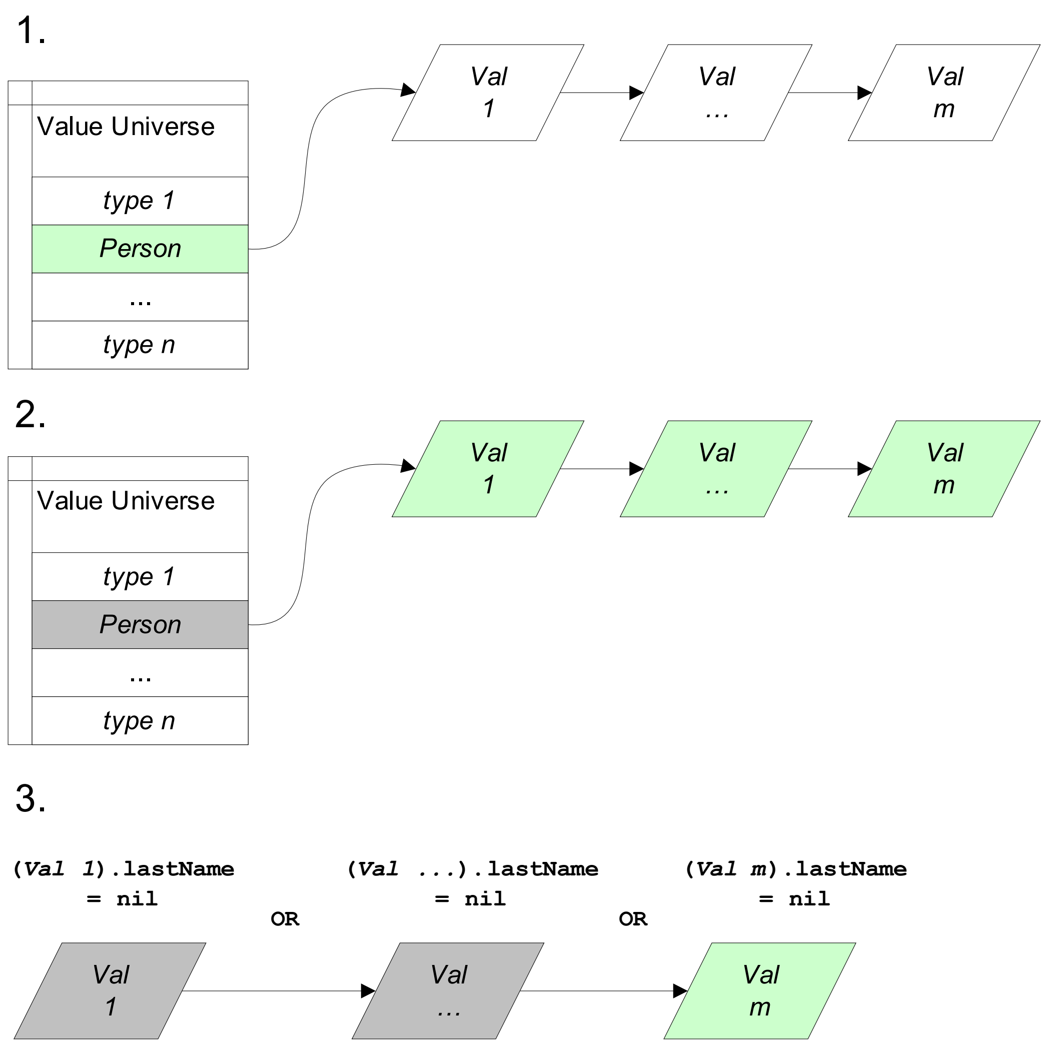

Figure 6.7: Exists example universe access

By the point where the first exists expression gets executed, none of the Person values picked up by the Universe have a nil lastName field. As a result, we expect the first exists expression to evaluate to false. Just prior to executing the second exists example, though, the FMSL interpreter processes the `let p3` expression where `p3` is assigned a `Person` value with the `lastName` field set to `nil`. Since the FMSL interpreter picks up that `p3 Person` value and places it in the Person pool of values in the Value Universe, the second forall expression should evaluate to false. This expectation turns out to be correct, as evidenced by the output in Figure 6.8.

```
{ "Alan", "Turing", 97 }
{ "Arnold", "Schwarzenegger", 61 }
"Expected: false"
false
{ "Charles", nil, 218 }
"Expected: true"
true
```

Figure 6.8: FMSL exists example output

6.2.3 Example: var:type such that

The code listing in Figure 6.9 demonstrates a forall with suchthat example.

```
(*
 * Perform lets with p1, p2, p3 to put them in the Universe
 *)
> (let p1:Person = {"Alan", "Turing", 97};);
> (let p2:Person = {"Arnold", "Schwarzenegger", 61};);
> (let p3:Person = {"Charles", nil, 218};);

(*
 * Evaluate: for all Person objects such that p.lastName is
 * not nil, the last name length is at least 6 characters
 *)
> "Expected: true";
> forall (p:Person | p.lastName != nil) #p.lastName >= 6;
```

Figure 6.9: FMSL forall with suchthat example code listing

In Figure 6.9 the FMSL code populates the Value Universe with three unique `Person` values, and one of those `Person` values (`p3`) has a `nil lastName` field. In this example the FMSL interpreter accesses the Value Universe in the same fashion as in the `forall` and `exists` examples. When evaluating the `forall` suchthat expression, though, the FMSL interpreter first evaluates the suchthat predicate (`p.lastName != nil`) and if it evaluates to true then it evaluates the second predicate (`#p.lastName >= 6`). Although there exists in the Universe a Person value with a `nil lastName` field, the `lastName` has at least six characters in all those `Person` values with a `lastName` that is not `nil`. See evidence below in Figure 6.10 for evidence.

```
{ "Alan", "Turing", 97 }
{ "Arnold", "Schwarzenegger", 61 }
{ "Charles", nil, 218 }
"Expected: true"
true
```

Figure 6.10: FMSL forall with suchthat example output

Chapter 7 Conclusions

The focus of this thesis has been a technique and tool to facilitate the incremental validation of formal software specifications. Demonstrations of the tool's functionality were presented, as were a detailed review of the tool's design and implementation.

7.1 Summary of Contributions

The specific contributions of the thesis are these:

1. The design and implementation of a functional interpreter for a formal specification language, rendering the language executable for the first time.
2. The design and implementation of a novel technique to execute purely predicative specifications, using validation invocations.
3. Demonstration of how the execution capabilities can be used to validate formal specifications.
4. A thorough discussion of how the specification execution capabilities fit into the realm of lightweight and heavyweight formal methods.

7.2 Future Work

The following section describes potential future work, which could involve creating a GUI front end to facilitate testing, creating a UML to FMSL conversion tool, adding a test case generator, improving FMSL's execution speed, making FMSL use memory more efficiently, and performing end-user studies.

7.2.1 UML to FMSL Tool

As UML is the standard for modeling software applications [56], a UML front-end for creating FMSL models could speed up the FMSL formal description creation process. Similarly, some people might find it helpful to view some parts of an FMSL specification in UML.

The general approach of UML-to-FMSL mapping is similar to the approach taken with other formal specification languages, such as UML-B [59]. Since UML does not have its own fully formal semantics, constructs of UML are mapped to the specification language, and those constructs assume the semantics of the language. This idea is consistent with the overall philosophy of UML, whereby the semantics of a particular UML diagram can "absorb" the semantics of an underlying language to which the diagram maps. For example, a UML inheritance diagram for a C++ program can assume the semantics of inheritance in C++. The same diagram used to depict a Java program has a different semantics of inheritance.

The FMSL reference manual describes a UML-to-FMSL mapping. Given this mapping, extant UML tools can be employed to render FMSL with UML diagrams.

For example, the Dia diagram editor [22] provides a plug-in capability, with which a textual representation of a diagram can be rendered as an editable drawing. An experimental version of a UML-to-FMSL graphical editor was implemented as a senior project at Cal Poly University, San Luis Obispo [53]. Since Dia is a Linux-based tool, its distribution is limited to Linux platforms. Wider distribution could be supported by using some other open-source UML editing framework, such as that currently under development for the Eclipse IDE [27].

7.2.2 Test Case Generator

As broad test coverage tends to build confidence about an implementation's correctness, so would broad test coverage build confidence about a model's correctness. Currently, FMSL validation operator test cases must be generated by hand. There are some benefits to generating test cases by hand, such as that the person generating the test cases may gain a better understanding of the model and data. Also, the person generating the test cases can carefully pick meaningful test cases. This process could be time-consuming and there exist tools, such as Korat [3, 17], that automatically generate test cases. Combining automated test case generation with FMSL's specification execution capabilities could make FMSL an even more useful tool for validating specifications.

7.2.3 GUI Front End

The validation operator allows the FMSL user to specify operation inputs and outputs and then see the result of their evaluation against the preconditions and postconditions. As described above, currently the user must choose the inputs and outputs, and enter them for execution in a text-based interpreter environment. A GUI front end to the specification validation and testing process could speed up and streamline the test case generation and evaluation process. It could help the user to manage a specification's test suite, which would consist of a set of test plans. Each operation could have its own test plan that consists of a set of inputs, outputs, and expected results of validation operator invocations.

Figure 7.1 is a sketch of the user interface for a GUI front-end to the specification validation functionality of the FMSL interpreter.

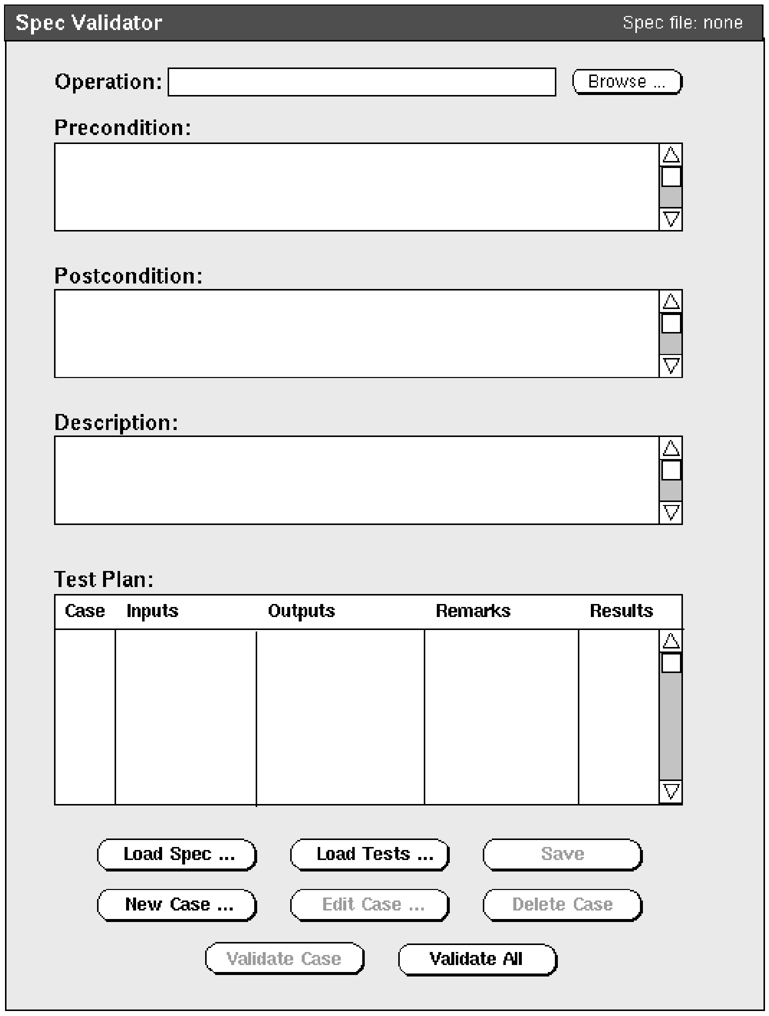

Figure 7.1: GUI overview sketch

The interface allows the user to load a specification, and focus on a particular operation. Each line in the Test Plan table corresponds to a validation invocation of the operation in view.

7.2.4 Improve Value Universe Performance

Although FMSL evaluation of quantifiers is fast on even a relatively slow personal computer, some improvements can be made to the Value Universe to improve execution time when many values of a particular type have been ingested by the Value Universe. As shown in Figure 4.3, the values for a given type are maintained in a simple linked list structure. Since by default FMSL checks for value existence before adding a new value to the Value Universe, this existence search process can slow down execution. The search process execution time could be reduced by implementing a companion structure that allows for translation of a hashed value pointer into an existence determination.

7.2.5 Garbage Collector

The current FMSL implementation does not manage memory very carefully, so we expect that FMSL executions probably lead to memory leaks. This behavior is considered acceptable for the proof-of-concept implementation developed in the thesis. An improvement to FMSL memory management would be to utilize a third party C-based garbage collector, so FMSL would perform all memory allocation and de-allocation through the garbage collector's interfaces.

7.2.6 End-User Studies

To assess the efficacy of the incremental validation tool, groups of tool users should be studied. A particular focus is use of the tool in undergraduate software engineering courses. This will involve the development of a suitable experimental framework, such as that presented by Sobel and Clarkson [60].

For example, student groups could be taken from two sections of the same class, with each section working on the same projects. One section uses the validation tool, the other does not. The student specifications can be assessed quantitatively and qualitatively to determine their accuracy, completeness, consistency, and soundness. The instructor can ensure that certain aspects of the specification are covered in both versions of the projects, so that a specific definition of soundness can be made.

Within this definitional framework, specific types of specification errors can be defined, and the existence of such errors can be determined in both the control and tool-use groups. These data can then be used to determine if the tool-use group performs better than the control group.

Bibliography

[1] S. Agerholm and P.G. Larsen. A lightweight approach to formal methods. In *Proceedings of the International Workshop on Current Trends in Applied Formal Methods*, pages 168–183, 1998.

[2] R. Alur, L. de Alfaro, R. Grosu, T.A. Henzinger, M. Kang, C.M. Kirsch, R. Majumdar, F. Mang, and B.Y. Wang. jMocha: a model checking tool that exploits design structure. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 835–836, 2001.

[3] A. Andoni, D. Daniliuc, S. Khurshid, and D. Marinov. Evaluating the "small scope hypothesis." Technical Report MIT-LCS-TR-921, MIT CSAIL, 2003.

[4] Association for Computing Machinery. *Proceedings of the fourth ACM workshop on Formal methods in security*, Fairfax , VA, Nov 3 2006.

[5] Association for Computing Machinery. *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, Atlanta, Georgia, USA. Nov 05–09, 2007.

[6] Association for Computing Machinery. *Proceedings of the twenty-third IEEE/ ACM international conference on Automated software engineering*, L'Aquila, Italy. Sept 15–19, 2008.

[7] D. Astels. *Test Driven Development: A Practical Guide*. Prentice Hall PTR, Upper Saddle River, New Jersey, 2003.

[8] B. Auernheimer and R.A. Kemmerer. ASLAN user's manual. Department of Computer Science, University of California, Santa Barbara, California, TRCS84–10, Apr 1992.

[9] K.S. Barber, T. Graser, and J. Holt. Providing early feedback in the development cycle through automated application of model checking to software architectures. In *Proceedings 16th Annual International Conference on Automated Software Engineering*, pages 341–345, 2001.

[10] K. Beck. *Test-Driven Development: by Example*. Addison-Wesley, 2003.

[11] B. Beizer. *Software Testing Techniques, 2nd edition*. International Thomson Computer Press, Boston, MA, 1990.

[12] D. Bjorner and C.B. Jones. *Formal Specification and Software Development*. Prentice Hall, Englewood Cliffs, NJ, 1982.

[13] B.W. Boehm. *Software Engineering Economics*. Prentice Hall, Englewood Cliffs, New Jersey, 1981.

[14] G. Booch, J. Rumbaugh, and I. Jacobsen. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[15] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. A subset of precise UML for model-based testing. In *Proceedings of the Third International Workshop on Advances in Model-based Testing*, pages 95–104, 2007.

[16] J.P. Bowen and M.G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):4–14, Jul 1995.

[17] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *Proceedings from the International Symposium on Software Testing and Analysis*, pages 123–133, 2002.

[18] W. Chan, R.J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J.D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, Jul 1998.

[19] S. Chandra, P. Godefroid, and C. Palm. Software model checking in practice: an industrial case study. In *Proceedings of the 24th International Conference on Software Engineering*, pages 431–441, 2002.

[20] M. Chechik and J. Gannon. Automatic Analysis of Consistency between Requirements and Designs. *IEEE Transactions on Software Engineering*, 27(7): 651–672, Jul 2001.

[21] E. Ciapessoni, A. Coen-Porisini, E. Crivelli, D. Mandrioli, P. Mirandola, and A. Morzenti. From formal models to formally-based methods: an industrial experience. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 8(1):79–113, Jan 1999.

[22] Dia User Community. The Dia Diagram Creation Program. http://projects.gnome.org/dia.

[23] A. Diller. *Z: An Introduction to Formal Methods*. John Wiley & Sons, New York, 1990.

[24] J. Douglas and R.A. Kemmerer. Aslantest: a symbolic execution tool for testing Aslan formal specifications. In *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 15–27, 2004.

[25] M.B. Dwyer and J. Hatcliff. Bogor: a flexible framework for creating software model checkers. In *Proceedings Testing: Academic and Industrial Conference - Practice And Research Techniques*, pages 3–22, 2006.

[26] S. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton. Experiences using lightweight formal methods for requirements modeling. *IEEE Transactions on Software Engineering*, 24(1):4–14, Jan 1998.

[27] Eclipse Foundation. Eclipse Model Development Tools. http://www.eclipse.org/modeling/.

[28] H. Erdogmus. On the effectiveness of test-first approach to programming. *IEEE Transactions on Software Engineering*, 31(1):1–12, Jan 2005.

[29] G. Fisher. *A Formal Modeling and Specification Language*. Department of Computer Science, California Polytechnic State University, San Luis Obispo, Technical Report No. CPSLO-CSC-09-01.

[30] G. Fisher. CSC 308 Course Lecture Notes Weeks 7 and 8: Introduction to Fully Formal Specification. http://users.csc.calpoly.edu/~gfisher/classes/308/lectures/7-8.html.

[31] G. Fisher. The Calendar Tool Project, http://www.csc.calpoly.edu/~gfisher/projects/calendar.

[32] D.C. Gause and G. M. Weinberg. *Exploring Requirements: Quality Before Design*. Dorset House Publishing, New York, New York, 1989.

[33] R.L. Glass. The mystery of formal methods disuse. *Communications of the ACM*, 47(8):15–17, Aug 2004.

[34] P. Godefroid. VeriSoft: a tool for the automatic analysis of concurrent reactive software. In *Proceedings of the 9th Conference on Computer Aided Verification*, Springer-Verlag, pages 476–479, 1997.

[35] W. Grieskamp and M. Lepper. Using use cases in executable Z. In *Third IEEE International Conference on Formal Engineering Methods*, pages 111–120, 2000.

[36] M.P.E. Heimdahl and N.G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, Jun 1996.

[37] C. Heitmeyer. On the need for *practical* formal methods. In *Proceedings of the Symposium on Formal Techniques in Real-Time and Real-Time Fault-Tolerant Systems*, pages 18–26, Sep 1998.

[38] P. Höfner and G. Struth. Automated Reasoning in Kleene Algebra. In *Proceedings of the 21st International Conference on Automated Deduction*, vol. 4603, pages 279–294, Springer-Verlag, Berlin, Heidelberg, 2007.

[39] P. Hudak and P. Wadler. *Report on the Functional Programming Language Haskell*. Department of Computer Science, Yale University, New Haven, Technical Report No. YALEU/DCS/RR656, 1988.

[40] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, Apr 2002.

[41] D. Jackson. Dependable software by design. *Scientific American Magazine*, pages 68–75, Jun 2006.

[42] D. Jackson and M. Rinard. Software analysis: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, pages 133–145, 2000.

[43] D. Janzen and H. Saiedian. Does test-driven development really improve software design quality? *IEEE Software*, 25(2):77–84, Mar 2008.

[44] S. Khurshid, C.S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568, 2003.

[45] D.R. Kuhn, R. Chandramouli, and R.W. Butler. Cost effective uses of formal methods in verification and validation. *Foundations '02 Workshop*, US Dept of Defense, Laurel MD, Oct 22-23, 2002.

[46] R.P. Kurshan. Formal verification in a commercial setting. In *Proceedings of the Design Automation Conference*, pages 258–262, 1997.

[47] P. Larsen, J. Fitzgerald, and T. Brookes. Lessons learned from applying formal specification in industry. *IEEE Software*, May 1996.

[48] G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, Mar 2006.

[49] M. Lutz. *Programming Python*. O'Reilly & Associates, Sebastopol, CA, 1996.

[50] K.L. McMillan. The SMV system for SMV version 2.5.4. http://www.cs.cmu.edu/~modelcheck (last updated: Nov 6, 2000).

[51] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[52] K. Myers, K. Dionne, J. Cruz, V. Vijay, S. Dunlap, and D.P. Gluch. The practical use of model checking in software development. In *Proceedings IEEE SoutheastCon 2002*, pages 21–27, 2002.

[53] T. Ober. *Graphical Editing of a Formal Specification Language*, Department of Computer Science Senior Project, California Polytechnic State University, San Luis Obispo, Jun 2006.

[54] Parasoft, Inc. JTest. http://www.parasoft.com/.

[55] M.N. Paryavi and W.J. Hankley. OOSPEC: an executable object-oriented specification language. In *Proceedings of the 1995 ACM 23rd Annual Conference on Computer Science*, pages 169–177, 1995.

[56] D. Pilone and N. Pitman. *UML 2.0 in a Nutshell*. O'Reilly Media, Sebastopol, CA, 2005.

[57] D. Richardson, O. O'Malley, and C. Tittle. Approaches to specification-based testing. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification*, ACM Press, pages 86–96, Dec 1989.

[58] B. Rumpe. Executable modeling with UML – A Vision or a Nightmare? In *Issues & Trends of Information Technology Management in Contemporary Associations*. Idea Group Publishing, Hershey, London, pages 697–701, 2002.

[59] C. Snook and M. Butler. UML-B: Formal modeling and design aided by UML. *ACM Transactions on Software Engineering and Methodology*, 15(1):92–122, Jan 2006.

[60] A.E.K. Sobel and M.R. Clarkson. Formal methods application: an empirical tale of software development. *IEEE Transactions on Software Engineering*, 28(3):308–320, Mar 2002.

[61] G.L. Steele. *Common Lisp the Language, 2nd Edition*. Digital Press, 1990.

[62] M. Vaziri and D. Jackson. Some shortcomings of OCL, the object constraint language of UML. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00)*, IEEE Computer Society, Washington, D.C., pages 555–562, 2000.

[63] J. Warmer and A. Kleppe. *The Object Constraint Language*. Addison-Wesley, 1998.

[64] K. Zee, V. Kuncak, and M.C. Rinard. Full functional verification of linked data structures. *ACM SIGPLAN Notices*, 43(6):349–361, Jun 2008.