PROVIDING A MEANS TO INCREMENTALLY VALIDATE FORMAL SPECIFICATIONS IN A STRAIGHTFORWARD MANNER

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

April 2009

AUTHORIZATION FOR REPRODUCTION OF MASTER'S THESIS

I reserve the reproduction rights of this thesis for a period of seven years from the date of submission. I waive reproduction rights after the time span has expired.

Signature

Date

APPROVAL PAGE

TITLE:

AUTHOR: Paul Corwin

DATE SUBMITTED: April 2009

Dr. Gene Fisher	
Advisor or Committee Chair	Signature
Dr. David Janzen	
Committee Member	Signature
Dr. Clark Turner	
Committee Member	Signature

Abstract

Providing a Means to Incrementally Validate Formal Specifications in a Straightforward Manner

by

Paul Corwin

Creating a software solution can be a difficult and complex process, and there are many ways to improve the this process: refine the requirements gathering process, improve specifications, create more rigorous test disciplines, select suitable and effective implementation methodologies, etc. This thesis work aims to improve the software development process by upgrading a formal methods tool to support specification execution, so that it helps users more effectively detect errors in the early stages of software development~~.~~

This lightweight formal methods tool and language, the Formal Modeling and Specification Language (FMSL), now provides a means to incrementally validate formal specifications in a straightforward manner that naturally fits into the software development process. FMSL also is used as a vehicle to teach software engineering students about formal methods, and so the updated FMSL can be used to facilitate instruction of more advanced formal methods topics.

Acknowledgements

afsdasdf asdfasdfas asdf asd fasdf asd fasdf quick brown fox acknowledgements bklajsjkldjklasdfk laklsjdklfkl kljsdjklf jklasjkldf klskldfkl lazy squirrel.

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~~ ~~this~~ thesis focuses on enabling early error detection ~~by providing a means~~ to validate formal specifications in a straightforward manner that naturally fits into ~~the~~ software development process.

~~To provide the capability to incrementally validate a specification this thesis work modifies the~~ Formal Modeling and Specification Language (FMSL)~~, which is a language that's intended to be used for specifying system behavior or system requirements definitions.~~ FMSL ~~also~~ is used as a vehicle to teach formal methods to software engineering students at California Polytechnic State University, San Luis Obispo (Cal Poly). ~~This thesis work extends FMSL to add a functional interpreter that executes a specification and evaluates reasonably complex preconditions and postconditions.~~

1.1 Description of the Problem

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. Many researchers have investigated methods to identify and correct errors during the earlier phases of the software development process, which is a time when error correction can have an attractive payoff since bugs tend to be cheaper to fix the earlier they are discovered [18].

Formal methods and models ~~are tools available that can help~~ describe and analyze a system ~~before software engineers implement the software solution~~. Key here is their pre-implementation use~~;~~ ~~because they~~ can help expose errors, misunderstood properties, and improperly stated behaviors that otherwise might have been overlooked. ~~After all, like~~ Gause and Weinberg warn, humans are~~n't~~ especially good at seeing what we've overlooked [19] 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" [32]. ~~So, it would seem~~ there is a need for tools and methods that help detect errors and increase confidence in formal models.

Add new paragraph

Critics claim that there is little utility to the use of formal methods: they doubt both their scalability and cost-effectiveness and they perceive that "formal notations and formal analysis techniques are difficult to understand and apply" [10]. These critics may have good reasons to doubt formal methods, as Heitmeyer points out that "the use of formal methods in practical software development is rare" [10]. Also, Bowen and Hinchey explain that "few people understand exactly what formal methods are or how they are applied" [17]. Despite ~~all the research and effort that's~~

~~gone into educating students and industry about the benefits of formal methods, more can be done in this area.~~

1.2 Overview of the Solution

This thesis' aims are twofold: to provide a means to validate formal specifications in a straightforward manner that naturally fits into the software development process and ~~to add credibility to the use of formal methods in the software engineering process.~~

Prior to beginning work on this ~~project~~/thesis, FMSL existed as a predicative specification language with a formal semantics that supported describing a system comprised of objects and operations. As "verification at early stages is more likely to be tractable" [4], FMSL is designed to be especially useful early on in the software engineering process. 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" [33].

~~To enable an engineer or student to increase confidence in a model and to detect model errors early on, this thesis provides a means to incrementally validate formal specifications in a straightforward manner by supplementing FMSL's ability to type-check specifications with the addition of a functional interpreter and a means to execute preconditions and postconditions. This execution of preconditions and postconditions, known as *operation validation*, can help answer questions like "is this postcondition strong enough?" or "do these preconditions and postconditions cover~~

~~the case where…?” FMSL’s precondition and postcondition execution is sophisticated to a point where it supports evaluation of both bounded and unbounded quantifiers. Test cases generated for operation validation purposes also can be re-used later on to test the implementation~~.

The FMSL language itself provides all these capabilities in a practical and balanced fashion. For example, while FMSL has a formal semantics – a must for modeling languages [32] – it exhibits a natural language expressiveness that should be “familiar to the user” [10] and so it could be appealing to non-software professionals [14] and engineers alike [7]. Also, unlike some other languages FMSL does not fall into the category of being so implementation-oriented that it’s not “well-suited for conceptual modeling” [33]. Quite the contrary, since FMSL is naturally expressive and could be understood by non-specialists, an FMSL specification might be appropriate to serve as the basis for a contract [17]. In short, using FMSL is designed to be convenient and easy, which Heitmeyer suggests is necessary in order for a formal methods tool “to be useful in practice” [10].

We believe this combination of a functional interpreter and a means to execute preconditions and postconditions will be make FMSL particularly useful and appealing to software engineering students, who expect an executable specification language [22]. Although 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 reap some benefit from using it [11]. On top of the academic benefits, while Jackson cautions that “as in a building, when the software’s foundation is unsound, the resulting structure is unstable” [31], using

FMSL to describe and validate a model makes it more likely that the model will serve as better foundation for the software that implements the model.

1.3 Outline of the Thesis

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

Chapter 2 Background and Related Work

~~FMSL is intended to be used in the context of light-weight formal methods. The following sections discuss formal methods, formal methods success stories, model checking, and light-weight model checking and light-weight formal methods.~~ The related work section provides a ~~brief~~ survey of relevant specification languages and model checkers.

2.1 Formal Methods

For decades, formal methods have been ~~hailed~~ by researchers as ~~invaluable tools that deserve a place in the~~ 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" [20]. ~~Historically, industry and researchers have used formal methods in a variety of scenarios: more broadly at the system~~ level to evaluate whether a system specification satisfies properties~~, but also in smaller granularities~~ to "formalize, debug, and prove the correctness of algorithms and protocols" [10].

Despite researchers' best efforts, critics ~~write~~ that formal methods have played ~~only~~ a small and insignificant roll ~~– if any –~~ in the software engineering process over the last 30 years [20]. ~~Formal methods~~ critics claim that formal methods ~~use~~ has a high barrier of entry, especially since many formal methods techniques ~~seem to~~ employ ~~a mathematical~~ notation ~~best used only by experts~~ [7]. Some argue that formal methods approaches are impractical at best, ~~so~~ there is no compelling reason to incorporate ~~formal methods~~ into their software engineering processes [20]. While that may be true in some cases, ~~ironically,~~ formal methods proponents argue that "formal methods are usually the only practical means of demonstrating absence of undesired behavior" [36]. Whether practical or impractical, difficult ~~to understand~~ or easy to understand, if a method helps expose errors then people likely will consider that method useful~~:~~ "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" [8].

2.1.1 Beneficial Uses of Formal Methods

~~Although a formal methods battle wages on,~~ there ~~seem to be many~~ beneficial uses for formal methods throughout the software engineering process. Formal methods can ~~have a place early on~~ during requirements development, specification, ~~and even later down the road during~~ implementation phases. Formal methods ~~usually involve formal~~ notations ~~and~~ a ~~forced~~ structure, which ~~together~~ can be used to present requirements. ~~In fact,~~ presenting requirements in formal notations can make "reviewing and inspection easier and therefore useful in locating errors" [12]. Some have found it useful to involve formal methods during the requirements engineering stages, where the formality prompt~~ed~~ the engineers to raise questions and "improve the overall quality of the existing specifications" [16]. Although there is more cost associated with formally defining and maintaining a system in multiple notations, ~~it seems especially in the case of unstable requirements~~ that early modeling ~~often~~ prove~~s~~ beneficial [16]. 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 [31] – 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, ~~effectively~~ separate concerns and express system properties [32]. 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 [12].

2.1.2 Formal Methods for System Parts

~~When people consider using formal methods, they should not assume that f~~ormal methods ~~are appropriate~~ across an entire system. ~~Rather, they should thoughtfully consider where and when to use them.~~ 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" [17]. Agerholm et al. conclude that sometimes "only parts of the systems would benefit from a formal model" [12]. 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 [16]. ~~Although it requires some~~ consideration to determine where formal methods use might be most advantageous, ~~that flavor of development process enhancement is precisely what Larsen et al. suggest~~ [7].

The benefits of code reuse are ~~obvious: it's sensible to reuse code modules that have been tested thoroughly~~. A~~nother~~ benefit ~~to~~ using formal methods is ~~that~~ ~~o~~nce system parts have been formalized into a model then those model parts can be reused [36], for example in later projects. That formal methods are reusable is a major benefit, but formal methods also promote code reuse: ~~especially~~ when the code has an accompanying, succinct description of guarantees and assumptions then it's easier to effectively re-use that code [32]. 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 [7]. 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 [7].

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. [4], model checking is a "formal verification technique based on state exploration" ~~while 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 [36], although failure to discover a counterexample does not necessarily prove correctness [35].

Confidence in a formal model is important because "an incorrect model can be worse than no model at all" [32]. Jackson et al. recommend developing a formal model of a system so long as it can be shown that the model describes the system [32]. 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" [36]. Although theorem proving technology has been around for decades, it has~~n't~~ 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" [8].

Model checking also can bring to the surface hard-to-find design errors [2], and it does so in a fashion that Kurshan [8] 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 [14]. That makes sense not just for convenience reasons, but also for cost benefits as Beizer [34] 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~~ come along with it~~. Model checking tends to require ~~a~~ specialized expertise, and when it's performed by hand then it can be very time consuming or even error-prone [1]. Experts are often needed because model languages can be rather difficult to learn [10]. These specialized experts may be called upon to translate a system into the model checking tool or language and then to interpret the results [1]. Given that experts may be involved and that this process can be time consuming, model checking can be costly [4] 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 [4, 6]. 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 [3]. In fact, 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 [6]. 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" [4].

All this considered, scalability with model checking remains a challenge [5]. Despite the difficulties that may come with performing model checking, model checking activities can help people better understand a system and specification [4]. 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 [6]. That in mind, it's better to discover these sorts of errors earlier rather than later.

2.3 Light-weight Formal Methods

~~Whereas heavy-duty formal methods often set out to completely prove or verify system behavior, there is another class of formal methods considered to be more "light-weight." Unlike the heavy-duty formal methods, light-weight formal methods involve using formal techniques as a reasoning aid, which can be beneficial since sometimes total proof of correctness may not be possible [36]. These light-weight methods – that Easterbrook et al. [16] describe~~ as involving "partial analysis on partial specifications, without a commitment to developing … complete, consistent formal specifications" ~~–~~ do not require that the user be trained in advanced mathematics or be ~~particularly savvy at coming up with novel, tricky~~ proof strategies [10].

Simulation is an example of a light-weight formal methods technique that animates or "electrifies" a model by examining a small subspace of possible states and transitions [32]. Especially when building a model incrementally, simulation may immediately expose easy-to-make mistakes [32]. Having this model available for early simulation also provides the users the convenient ability to test functional requirements of interest [21]. 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" [32].

While heavy-duty formal methods do have a use – ~~perhaps~~ in especially interesting or critical software components – Jackson explains that light-weight formal methods can be more practical [31]. Dwyer et al. [3] concur that in some cases it~~’s~~ 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 just is~~n’t~~ necessary [4] and “not everything should be formalized” [37].

2.3.1 Light-weight Formal Methods and Test~~-First~~ 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, ~~e.g.,~~ 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 [27, 28, 29]. Erdogmus [29] found that following this “test-first” philosophy seems to improve productivity. Janzen et al. [30] 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" [6]. This improved understanding can help developers to more easily spot errors or problems, and it can improve customer-developer communication [36]. 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 [34] then this synergy should be friendly on the budget.

2.3.~~2~~ Cost Effectiveness

On the topic of costs, the choice to utilize light-weight methods may be both practical and cost-effective [10]. 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" [31]. This relatively low-cost investment ~~that~~ provides ~~some~~ reasonable coverage of test cases against a model yields an increased confidence in the model's correctness [4]. If more assurance is needed ~~even~~ after utilizing light-weight formal methods, ~~it might make sense to apply heavy-duty formal methods – like model checking – after less expensive methods have been attempted~~ [3].

For all the above reasons, light-weight formal methods may be attractive to industry. Since light-weight formal methods provide something of an incremental change to existing software processes – rather than a revolutionary change – they ~~are much~~ more likely to be seriously considered ~~especially~~ in large organizations where it~~'s~~ difficult to push against process inertia [7, 37]. The FMSL modifications for this thesis transform FMSL into a more effective and useful tool that fits well with light-weight formal methods techniques, and so its use could be introduced incrementally.

2.4 Model Checking Tools and Formal Specification Languages

~~While~~ ~~m~~any automated model checking tools and formal specification languages exist~~,~~ ~~e~~ach has a set of characteristics that make ~~them best used in certain scenarios.~~ In kind, FMSL has its own characteristics and potential uses~~, but~~ ~~w~~hat follows is a brief survey of some ~~already~~ 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'" [2] 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 [39]. Rather than analyzing a ~~hand-written model of a system,~~ 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 [6, 40]. 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) algorithm 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 [40] 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. [6] 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) [41] is a behavioral interface specification language that~~'s~~ intended to be used for specifying Java modules by describing preconditions, postconditions, and intermixed assertions. Leavens et al. [41] 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" [13]. 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 [9, 13].

2.4.4 UML and OCL

The Unified Modeling Language (UML) [38] 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" [42]. A UML model generally consists of one or more diagrams and "provides a more compact code description than an ordinary programming language does" [24].

Although UML typically is not thought of as an executable language, there is ~~at least~~ some subset of UML that ~~could be executed,~~ ~~e.g.,~~ class diagrams, StateChart diagrams, activity diagrams, sequence diagrams, and the Object Constraint Language

(OCL) [24]. Bouquet et al. [26] 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 [42]. 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 [33, 35] yet they ~~admit~~ it'~~s~~ more easily used by non-mathematicians compared to some other modeling languages [35]. Also, OCL is not a standalone language since it always must be accompanied by a UML diagram [33, 35]. 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 [36].

2.4.5 OOSPEC

OOSPEC [22] 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. [22] 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 [43] 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 [21] 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. 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 [7]. 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, ~~etc~~.

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 [11]. 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 13 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 (~~namely;~~ they interchanged existential and universal quantifiers), their system description ~~still~~ 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 ~~remarkably~~ better test success rate compared to the control group teams' submissions: 100% correctness vs. 45.5% correctness. Of the 13 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 [16]. 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 [16] 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.

Chapter 3 ~~Examples of Use~~

~~FMSL specifications can include object type and operation declarations as well as expressions, quantifier evaluations, operation validations, and comments. Within an FMSL specification, object type, variable, value and operation declarations appear inline, while expressions and operation invocations are preceded by the '>' character. Comments are enclosed in `(* *)` or single line comments begin with "--". See the example below in Figure 3.1, where two values are declared and a summation expression is evaluated.~~

```
(*
 * file: sample_spec.fmsl
 *)

(*
 * Declare and assign values to i, j
 *)
val i:integer = 42;
val j:integer = 73;

(*
 * Evaluate and output their sum
 *)
> i + j;
```

~~Figure 3.1: Sample FMSL specification~~

~~The FMSL translator tool that executes specifications is run from the command line. The FMSL binary executable accepts as command line parameters~~

~~the file names of the FMSL specification source files. The specification execution output includes a listing of any errors encountered, or if no errors are encountered then it outputs the results of expression evaluations. To execute a specification contained within a file called~~ `sample_spec.fmsl`, ~~which we expect to output the sum of~~ `42` ~~and~~ `73`, ~~the user would type:~~

```
os_prompt:$ fmsl sample_spec.fmsl
115
```

~~The following sections 3.1, 3.2, and 3.3 discuss more examples of expression evaluation, quantifier evaluation, and operation validation, respectively.~~

3.1 Standard Expression Evaluation

In FMSL, expression evaluation ~~amounts to~~ invoking an operator or operation and returning the calculated result. FMSL supports evaluation of a collection of built-in Boolean, arithmetic, tuple, and expressions as well as evaluation of user-created operations. For a complete list of built-in operators, see ~~Table 5.2, Table 5.3, Table 5.4, Table 5.5, and Table 5.6.~~

The example in Figure 3.2 demonstrates evaluation of the Boolean relational operators: `not`, `and`, `or`, `xor`, => (implication), and <=> (two-way implication); the output appears in Figure 3.3. In the example, the FMSL code first declares and assigns values to two `boolean` variables and then performs a series of Boolean expression evaluations.

```
(*
 * Declare and assign values to t, f
 *)
val t:boolean = true;
val f:boolean = false;

(*
 * Boolean operator examples
 *)
> "Expected: false";
> not t;

> "Expected: false";
> t and f;

> "Expected: true";
> t or f;

> "Expected: true";
> t xor f;

> "Expected: false";
> t => f;

> "Expected: false";
> t <=> f;
```

~~**Figure 3.2: FMSL Boolean expression evaluation examples**~~

```
"Expected: false"

false

"Expected: false"

false

"Expected: true"

true

"Expected: true"

true

"Expected: false"

false

"Expected: false"

false
```

Figure 3.3: ~~Output from~~ evaluating boolean expressions

The example in Figure 3.4 demonstrates evaluation of the arithmetic division operator. In the example, the FMSL code first declares and assigns values to two `real` variables 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 3.4: FMSL division operator expression evaluation

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 ~~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 infinitely large~~.

The following sub-sections ~~contain introductions to these~~ different ~~types~~ of quantifiers. In the examples, the `Person` object is defined by the FMSL code listing in Figure 3.5~~.~~

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

Figure 3.5: FMSL Person object type definition

3.2.1 Bounded Quantifier

The ~~following FMSL~~ code ~~listing~~ in Figure 3.6 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 ];

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

Figure 3.6: FMSL bounded quantifier example

3.2.2 Unbounded Quantifier: forall

The following ~~FMSL~~ code ~~listing~~ in Figure 3.7 performs some simple `let` expressions involving `Person` objects and then evaluates an unbounded quantifier to ~~make sure~~ that all the `Person` objects have non-nil last names. Since ~~all~~ the `Person` objects have non-nil last names, the result is `true`.

```
(*
 * 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.last~~Name~~ != nil;
```

Figure 3.7: FMSL unbounded forall quantifier example

3.2.3 Unbounded Quantifier: exists

The ~~following FMSL~~ code ~~listing~~ in Figure 3.8 performs some simple `let` expressions involving `Person` objects 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`.

```
(*
 * 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;
```

Figure 3.8: FMSL unbounded exists quantifier example

3.2.4 Unbounded Quantifier: forall with such that

The ~~following FMSL~~ code ~~listing~~ in Figure 3.9 performs some simple `let` expressions involving `Person` objects, but unlike the previous two examples this sequence of `let` expressions includes a `Person` value that has a `nil` last name. ~~Next,~~ ~~t~~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`.

```
(*
 * 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 3.9: FMSL unbounded forall / suchthat quantifier example

3.3 Operation Validation

This section describes how a user might utilize the FMSL validation operator (`?->`) to incrementally validate a specification by performing a sequence of operation validations. An invocation of the validation operator requires an operation name, an input argument list, and an output argument list and generally is used as follows:

```
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 user database system. Specifically, the sub-sections below follow the weak-to-strong evolution of preconditions and postconditions of the `AddUser` and `FindUser` operations. The `AddUser` operation adds a user record to the database and the `FindUser` operations take in a `Name` or `Id` key to locate user records within the user database.

The following examples come directly from Dr. Gene Fisher's CSC 308 (Undergraduate Software Engineering I) lecture notes, weeks 7 and 8 [44], and some of the explanation text comes straight out of these lecture notes as well. For the following examples, the object type definitions from Figure 3.10 apply. These object type definitions describe individual components of a user record, a user record, and a user 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 3.10: 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 3.11 below, 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 3.11: AddUser with English precondition and postcondition

Although the `AddUser` precondition and postcondition descriptions from Figure 3.11 appear in plain English, that form of the `AddUser` operation already is executable through the validation operator. To demonstrate this executability, in Figure 3.12 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 }`.

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.

```
(*
 * 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

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

Figure 3.12: AddUser basic tests

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 demonstrated in Figure 3.13.

```
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 3.13: AddUser with basic postcondition logic

In Figure 3.14 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 3.14: 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, it's safe to classify that as a weak precondition. To check whether the postcondition is strong enough, we might use the validation operator to run some example inputs and outputs against AddUser. The example in Figure 3.15 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 3.15: Test for postcondition strength

The first invocation of the validation operator in Figure 3.15 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` field 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 3.13 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 3.15, 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 3.16 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 3.16: AddUser with stronger postcondition

When we re-run the test from Figure 3.15 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 3.17 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 3.17: AddUser with constructive postcondition

In the above example, the postcondition succinctly describes that the output user database must be equal to the input database with the input user record appended to it.

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. See Figure 3.18 for 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 3.18: FindUserByName with English precondition and postcondition

Just like 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 3.19: FindUserByName operation validation tests

Just like the in example from 3.3.1, since the precondition and postcondition are not formally defined then we expect the result for all four tests to be `{ true, nil }`. See Figure 3.20 for 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 3.20: FindUserByName initial validation results

3.3.7 FindUserByName: Basic Postcondition Logic

A sensible next step in formalizing the postcondition might be to make sure that the operation output consists of all records of the given name in the input db. The formal logic in Figure 3.21 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 3.21: FindUserByName with basic postcondition

To test our new definition of `FindUserByName`, we run the same set of tests from Figure 3.19 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 3.22).

```
"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 3.22: 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 must 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 3.22. To address this requirement, the `FindUserByName` definition in Figure 3.23 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 3.23: FMSL FindUserByName with sort constraint

As we expected, when running the tests in Figure 3.19 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 3.24).

```
"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 3.24: 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 3.19. According to the output in Figure 3.24, 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 that easily can be fixed: 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 3.25 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 3.25: 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 3.26 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 3.26: 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 3.27 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 3.27: FindUserByName with auxiliary functions

The `FindUserByName` definition in Figure 3.27 is functionally equivalent to the `FindUserByName` definition in Figure 3.25, although it's arguably more readable. Observe in Figure 3.28 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 3.25.

```
"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 3.28: FindUserByName with aux. functions validation results

Chapter 4 Overall System Design

Initially the FMSL translator ingested FMSL source code and provided execution output that included information about any type errors detected. If no type errors were detected, the FMSL translator did not provide any output. See Figure 4.29 for a visual representation of the FMSL translator initial structure.

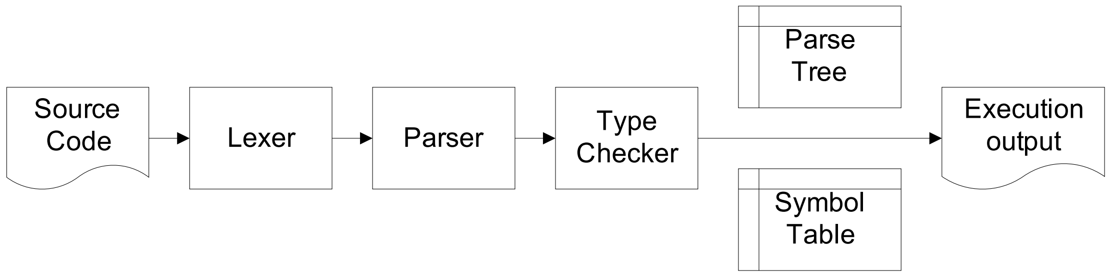

Figure 4.29: FMSL translator initial structure

To make FMSL at least incrementally more useful, the work for this thesis called for adding support for evaluating expressions through a functional interpreter. This functional interpreter implementation does not perturb any type-checking capabilities of FMSL. Per conventional compiler design principles, the interpreter implementation relies on the type-checker's results. In the latest FMSL version, the execution output is not just limited to type errors but it also includes – where appropriate – results from expression evaluations or any run-time errors. See Figure 4.30 for 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 5.

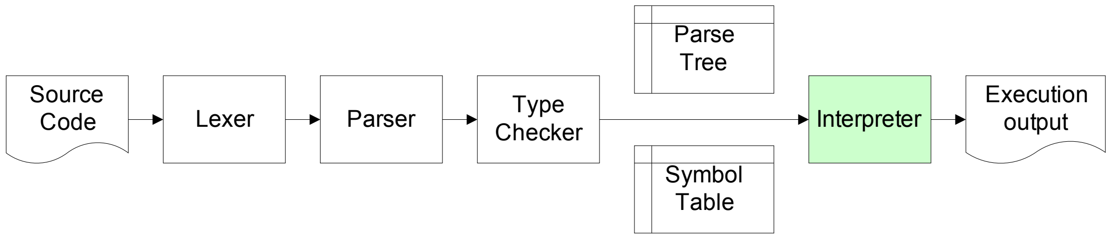

Figure 4.30: FMSL translator structure with interpreter

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 help meet the goal of allowing the user to execute a specification, this thesis next called for adding 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 nominally Boolean values that indicate results from precondition and postcondition evaluation. It is important to note that precondition and postcondition evaluation takes place even though the operation implementation need not be defined in FMSL. This evaluation is possible because preconditions and postconditions don't describe implementation details but instead they describe properties about the input and output values. For more details on the validation operator implementation, see section 5.4

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 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 conceptually 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. See 6 for quantifier implementation details and for 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 such as lists and tuples.

The decisions regarding when the FMSL interpreter should add values to the Value Universe kept in mind 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.

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 control 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 homogenous list of values for a particular type. See Figure 4.31 for a visual representation of the Value Universe structure.

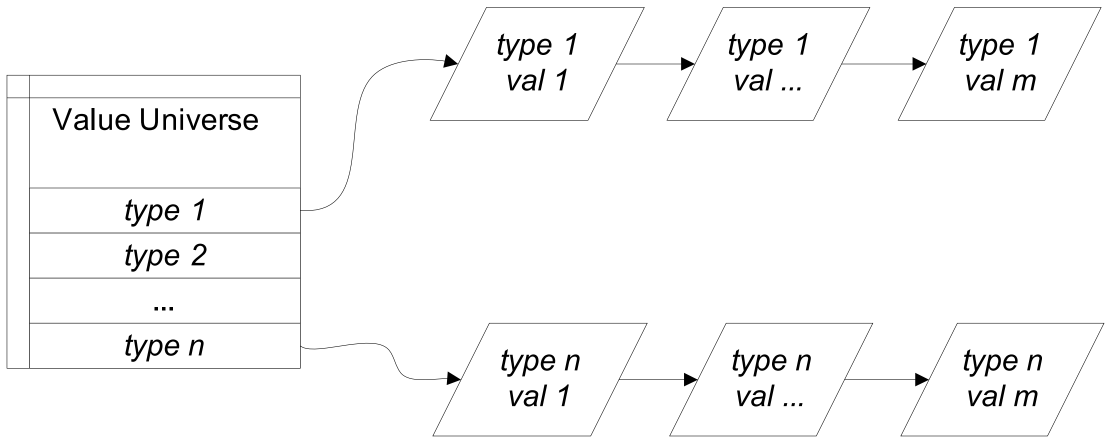

Figure 4.31: Value Universe structure

The FMSL code listing below in Figure 4.32 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 4.32: 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 4.33:1) 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 doesn't already exist in the Universe, the FMSL implementation adds the value represented by `p2` to the end of the `Person` list (see Figure 4.33:2).

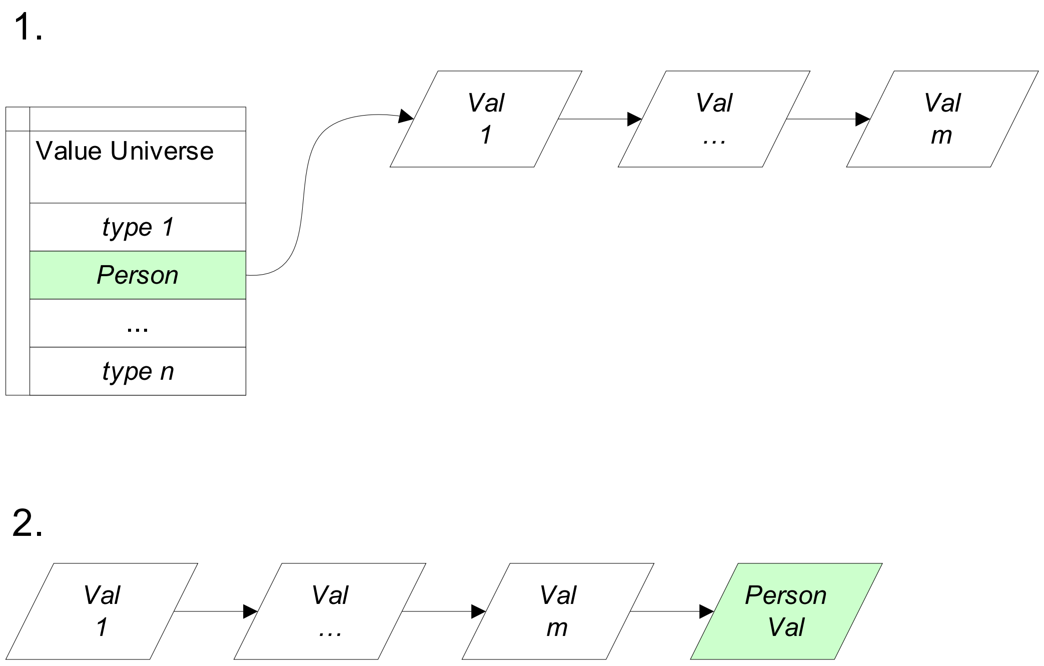

Figure 4.33: Value Universe Add Person Value

By executing the code in Figure 4.32 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. See Figure 4.34 for the FMSL output, which shows that the Value Universe contains both `Person` values, after executing the code in Figure 4.32 with the “`-dump-universe`” command line option.

```
true

true

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

Figure 4.34: 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. Note that FMSL also supports the concept of a `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 this information, which is listed 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 5.35.

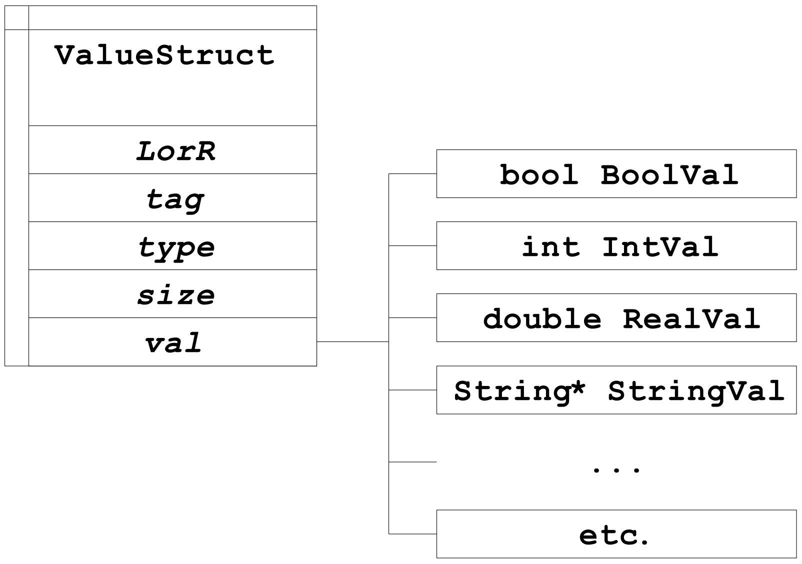

Figure 5.35: ValueStruct structure with val union

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's 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 following listing in Figure 5.36, 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 5.36: 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 `ValueStructs` corresponding to the `x` and `y` operands

The C code for evaluating the division appears below in Figure 5.37.

```
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;

    /*
     * The possibilities here are only real and longreal.
     */
    switch (v1->tag) {
        case RealTag:
            if (v2->tag == IntTag) {
                if (v2->val.IntVal == 0) {
                    free(v2);
                    lerror(t, "Divide by zero.\n");
                    longjmp(RuntimeError, DivideByZeroStatus);
                }
                v1->val.RealVal = v1->val.RealVal / v2->val.IntVal;
            }
            else {
                if (v2->val.RealVal == 0) {
                    free(v2);
                    lerror(t, "Divide by zero.\n");
                    longjmp(RuntimeError, DivideByZeroStatus);
                }
                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");
                    longjmp(RuntimeError, DivideByZeroStatus);
                }
                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");
                    longjmp(RuntimeError, DivideByZeroStatus);
                }
                v1->val.IntVal = v1->val.IntVal / v2->val.IntVal;
            }
            free(v2);
            return v1;
    }
}
```

Figure 5.37: 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. Note that there's 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 also supports more complex structures: lists and tuples. FMSL lists are homogenous data structures that hold zero or more object values, analogous to an array with no predetermined, fixed size. FMSL tuples are and-composed data structures that hold a fixed number of components of specific object types, similar to a record or 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 following FMSL code in Figure 5.38 declares an object type called `Person`, which contains several fields that together help describe a person.

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

Figure 5.38: 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 5.39), which is a C `struct` that contains a linked list of generic list elements and other list metadata such as list size.

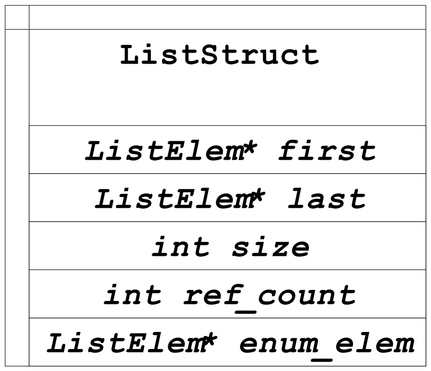

Figure 5.39: ListStruct definition

The FMSL code snippet in Figure 5.40 below defines an `IntegerList` object type and creates an `IntegerList` instantiation called `intlist`.

Code listing:

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

(*
 * Declare the intlist variable
 *)
var intlist:IntegerList;

(*
 * Assign a collection of integers to intlist
 *)
> set intlist = [1,1,2,3,5,3+5];
```

Output:

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

Figure 5.40: 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 5.40, 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 5.41 for the `doListConstructor` C code that performs list construction.

```
ValueStruct doListConstructor(t)
    nodep t;
{
    TypeStruct type              /* Type of the array */
              = t->header.attachment.n2;
    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);
    } /* end foreach expression */

    return rtn;
}
```

Figure 5.41: doListConstructor implementation

An example list operator implementation that's 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 5.42 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 the intlist variable
 *)
var intlist:IntegerList;

(*
 * Assign a collection of integers to intlist
 *)
> set intlist = [1,1,2,3,5,3+5];

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

Output:

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

[ 2, 3, 5 ]
```

Figure 5.42: 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 5.43. Next, the code determines that the `v1` parameter is an FMSL list[1] and so the C code initializes `result` as an empty list. By looping from the lower bound value `v2` to the upper

[1] 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.

bound value `v3`, one at a time the code accesses the selected subcomponents of `v1` 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 5.43: 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 5.39); however, unlike ListVal, each item in the StructVal list corresponds to a field within the FMSL tuple. See Figure 5.44 for 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
 *)
var p:Person;

(*
 * Initialize p
 *)
> set p = {"Arnold", "Schwarzenegger", 61};

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

Output:

```
{ "Arnold", "Schwarzenegger", 61 }

"Schwarzenegger"
```

Figure 5.44: 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 5.45) 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 5.45: 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 5.44 demonstrates field access.

To evaluate tuple field access, the FMSL interpreter first determines that it's 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 5.46), 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 5.44 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 5.46: RecordRef implementation

5.3 Operation Invocation

In order to support easier-to-read specifications and to give FMSL more utility as a specification language, the FMSL interpreter supports operation invocation. User-defined, named operations can range from simple to complex. See Figure 5.47 for 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 5.47: 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 classic 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 }`	run-time / execution error 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 a run-time / execution error 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

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. 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's 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

Although it might be unintuitive, 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 [15], 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" [36]. 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. See Table 6.9 for 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 6.9: FMSL quantifier syntax

A bounded quantifier iterates over a discrete universe of values, as seen in in Figure 6.48. In the example, the `forall` implementation loops over all five elements [1, 1, 2, 3, 5] that make up `IntList list`. Note that since each `integer` element within `list` is greater than zero in the example in Figure 6.48, 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.
 *)
> "Expected: true";
> forall (i in list) i > 0;
```

Figure 6.48: Example of a bounded quantifier in FMSL

In the example in Figure 6.49, unlike in Figure 6.48, it's not clear how the interpreter should evaluate the unbounded universal quantifier since the `Person` space is an undefined and potentially infinitely large universe.

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

Figure 6.49: 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 handle unbounded quantifiers. For example, Aslantest [21], Jahob [23], and executable Z [25] all handle unbounded quantifiers in different ways.

The symbolic execution tool for Aslan, Aslantest [21], 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 [23] 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" [25]. In [25] 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 following code listing in Figure 6.50 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 6.50: FMSL forall example code listing

In the above listing, to populate the Universe with `Person` values, the code lists 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 that the Universe could pick up 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 6.51:1). After discovering that there exist `Person` values in the Universe, the FMSL interpreter accesses that list of `Person` values (see Figure 6.51:2). 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 essentially ANDs the results together (see Figure 6.51:3) to arrive at the final evaluation result.

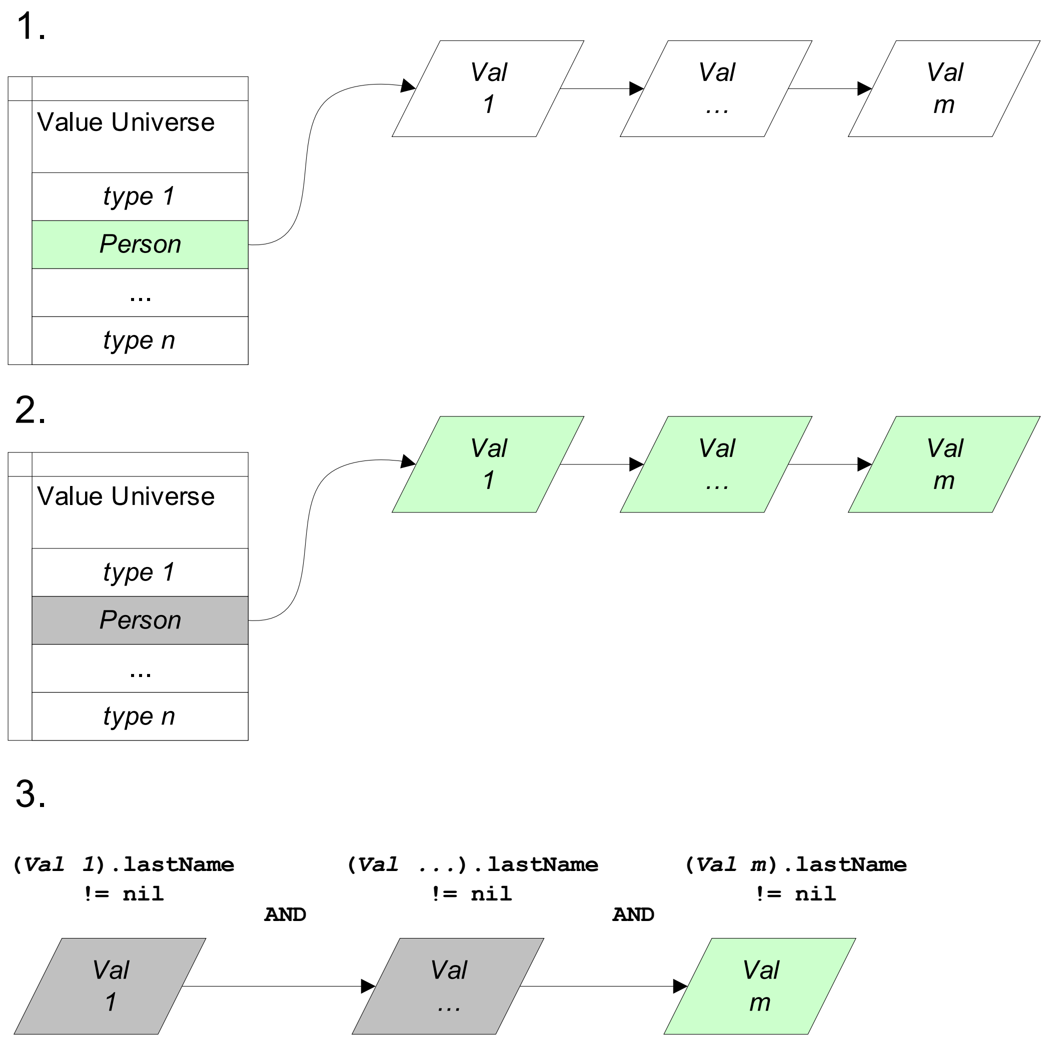

Figure 6.51: Forall example universe access

Note that in the example in Figure 6.50, 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 6.52 below.

```
{ "Alan", "Turing", 97 }

{ "Arnold", "Schwarzenegger", 61 }

"Expected: true"

true

{ "Charles", nil, 218 }

"Expected: false"

false
```

Figure 6.52: FMSL forall example output

6.2.2 Example: exists

The following code listing in Figure 6.53 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 6.53: FMSL exists example code listing

The example in Figure 6.53 starts out the same as in Figure 6.50 where the Universe gets populated with some `Person` values. Where it's different, though, is that we see `exists` quantifiers instead of `forall` quantifiers.

Like 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 6.54:1). After discovering that there exist `Person` values in the Universe, the FMSL interpreter accesses that list of `Person` values (see Figure 6.54:2). 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 6.54:3) to arrive at the final evaluation result.

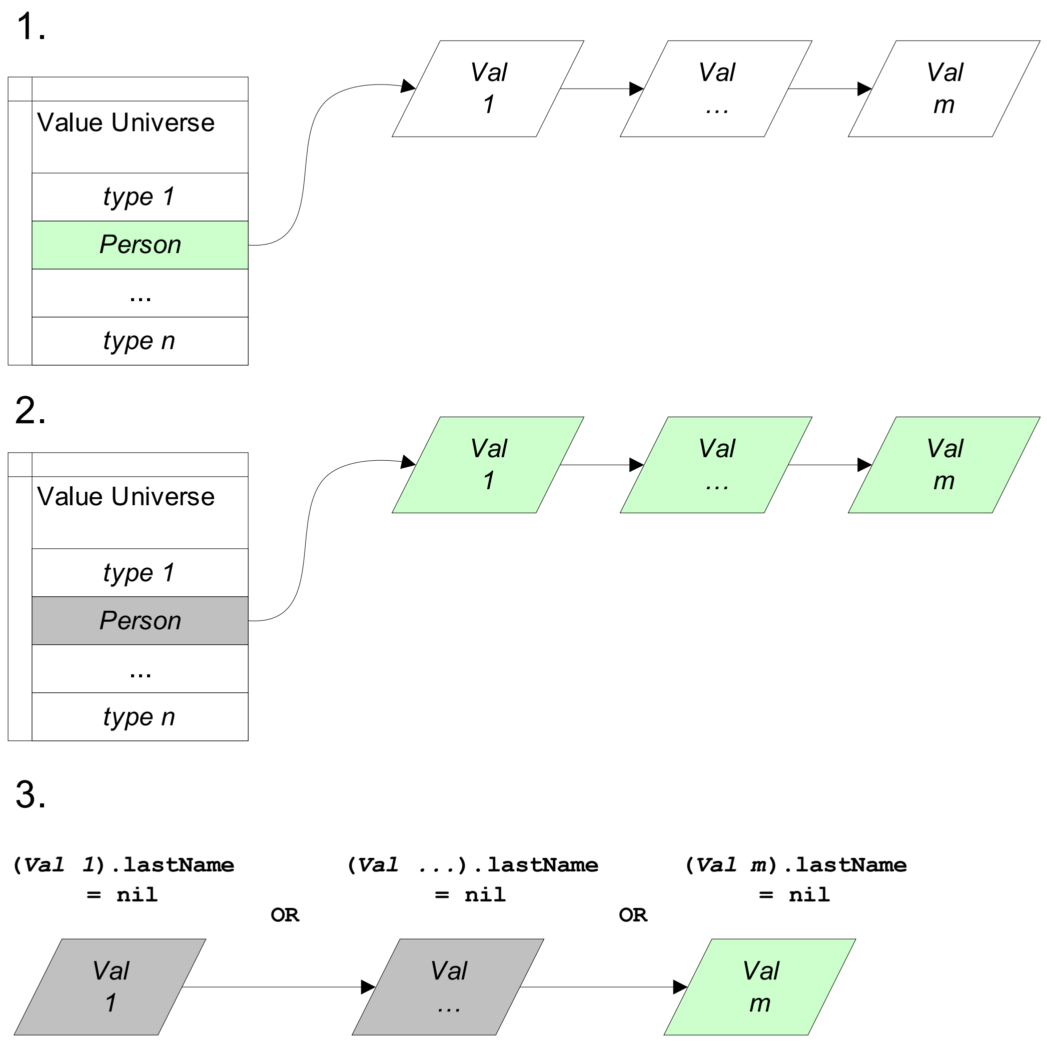

Figure 6.54: Exists example universe access

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 6.55.

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

Figure 6.55: FMSL exists example output

6.2.3 Example: var:type such that

The following code listing in Figure 6.56 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 6.56: FMSL forall with suchthat example code listing

In Figure 6.56 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 6.57 for evidence.

```
{ "Alan", "Turing", 97 }

{ "Arnold", "Schwarzenegger", 61 }

{ "Charles", nil, 218 }

"Expected: true"

true
```

Figure 6.57: FMSL forall with suchthat example output

Chapter 7 Conclusions

Creating a software solution can be a difficult and complex process, and there are many ways to improve the this process: refine the requirements gathering process, improve specifications, create more rigorous test disciplines, select suitable and effective implementation methodologies, etc. This thesis work aimed to improve the software development process by upgrading a formal methods tool to support specification execution, so that it would help users more effectively detect errors in the early stages of software development. FMSL, as a lightweight formal methods tool and language, now provides a means to incrementally validate formal specifications in a straightforward manner that naturally fits into the software development process.

7.1 Summary of Contributions

To facilitate specification execution, the work for this thesis first involved creating a functional interpreter for FMSL. Next we added support for operation invocation and execution of preconditions and postconditions. Then we added the capability to evaluate bounded and unbounded quantifiers in a meaningful way. Finally, we implemented a validation operator that enables the user to test inputs and outputs against formally specified operations. The creation of test cases for use with

FMSL's validation operator also has a strong synergy with the test-first methodology [27, 28, 29, 30], and these test cases can be reused at any later point in the software development process. All these contributions together support the goal of providing a means to incrementally validate formal specifications in a straightforward manner, and the results also can facilitate instruction of effective formal methods use to software engineering students.

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, and making FMSL use memory more efficiently.

7.2.1 UML to FMSL Tool

As UML is the standard for modeling software applications [42], 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. Although there isn't a one-to-one correspondence between UML classes and FMSL object types, there is some overlap in meaning and so such a tool set seems feasible and useful.

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 likely would get a better understanding of the model and data and the person generating the test cases could carefully pick meaningful test cases. This process could be time-consuming and there exist some tools, such as Korat [9, 13], 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 mentioned above, currently the user must choose the inputs and outputs (and type them out by hand into ASCII format). 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.

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 4.31, 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 doesn't manage memory very carefully, so we expect that FMSL executions probably lead to memory leaks. An improvement to FMSL memory management would be to utilize a third party C-based garbage collector, and so FMSL would perform all memory allocation and de-allocation through the garbage collector's interfaces.

Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

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

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

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

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

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

[21] 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.

[22] 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.

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

[24] 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.

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

[26] 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.

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

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

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

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

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

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

[33] 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.

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

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

[36] 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.

[37] 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.

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

[39] 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.

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

[41] 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.

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

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

[44] G. Fisher. Introduction to Fully Formal Specification. http://users.csc.calpoly.edu/~gfisher/classes/308/lectures/7-8.html.

[45] G. Fisher. ~~FMSL User Manual. TBD~~