2002-08-01

A Formal Method to Analyze Framework-Based Software

Trent N. Larson
Brigham Young University - Provo

A FORMAL METHOD TO ANALYZE FRAMEWORK-BASED SOFTWARE SYSTEMS

by

Trent Larson

A dissertation submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

Brigham Young University

March 2002

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a dissertation submitted by

Trent Larson

This dissertation has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Phillip J. Windley, Chair
Date	J. Kelly Flanagan
Date	Mike D. Jones
Date	Scott N. Woodfield
Date	Dan R. Olsen

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the dissertation of Trent Larson in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

Phillip J. Windley
Chair, Graduate Committee

Accepted for the Department

David Embley
Graduate Coordinator

Accepted for the College

G. Rex Bryce, Associate Dean
College of College of Physical and Mathematical Sciences

ABSTRACT

A FORMAL METHOD TO ANALYZE FRAMEWORK-BASED SOFTWARE SYSTEMS

Trent Larson

Department of Computer Science

Doctor of Philosophy

Software systems are frequently designed using abstractions that make software verification tractable. Specifically, by choosing meaningful, formal abstractions for interfaces and then designing according to those interfaces, one can verify entire systems according to behavioral predicates. While impractical for systems in general, framework-based software architectures are a type of system for which formal analysis can be beneficial and practical over the life of the system. We present a method to formally analyze behavioral properties of framework-based software with higher-order logic and then demonstrate its utility for a significant, modern system.

ACKNOWLEDGMENTS

For Lynnette, because a process such as this is so tightly intertwined with my life as a whole and you've been incredibly supportive and understanding as I pursued my passion. Thank you!

For Phil, because you supplied your ample technical vision, expertise, and teaching skills so many times in my behalf. If that weren't enough, you gave much needed moral support delivered to me personally and to others on my behalf.

For Mike, Annette, Robert, Paul, Scott, Kelly, and Leanne from the Laboratory for Applied Logic, because you helped with my education both in the technical arts and in my personal growth. My life has been forever touched and enriched by your examples and assistance.

Finally, for the people of BYU, other academic and commercial institutions, and friends and relatives who have touched my life, because your ideals and committment to them have inspired me in my continuing growth. I will pay forward the favor.

Contents

List of Tables

List of Figures

Chapter 1

Introduction and Background

The dream of incorporating mathematical theory into the software design process is still alive, albeit with a different perspective from a decade ago. Cutting edge and commercial software development continues to deploy useful methodologies and abstractions before rigorous mathematical models can be developed and we do not see an end to this trend anytime soon. However, languages and design tools are giving more attention to theoretical foundations and more research and conferences are devoted to applying mathematical theories to large, practical problems. Just as hardware verification has evolved from a scattered, "conquer-the-world" approach to an effort focused on certain behaviors and systems, software verification is working its way into niches where it can be the most effective. For example, there exist automatic analysis tools for low-level memory security or type-safety.

This work identifies a type of object-oriented software system, a formal approach, and specific logical translations that make it practical to formally analyze the system. In particular, we show how the properties and structure of a software framework can be modelled using abstract theories and verified using a simple system of operational semantics.

Section 1.1 discusses lessons learned from hardware verificiation that apply in this work. Section 1.2 defines the framework systems this thesis aims to handle. Section 1.3 discusses related ideas and tools in the field of system verification.

Our thesis statement is found at the beginning of Chapter 3. In short, this work aims to make large-scale software verification more practical. It contributes to other formal methodologies by addressing the following problems:

1. Since verification of generic systems has proven impractical, we identify a class of systems which lend themselves to formal methods, showing why and how to verify them beneficially.

2. Since formalisms are expensive to create and difficult to share, we explain a method to specify and verify systems of components at multiple levels of abstraction such that the formal work at different levels can be shared and reused.

1.1 The Hardware Verification Process

We introduce three approaches followed by three specific tasks which have proven useful for formal hardware verification. We apply these principles later when we describe our method for framework software.

Before explaining these approaches and tasks, let us review the use of logic in verification. Every logical term states a relationship. For example, the following term states that, for every possible substitution of the variables `a`, `b`, `c`, `s`, and `cout`, the sum of the first three is equal to the sum of `2*s` and `cout`:

```
∀ a b c s cout . (a+b+c = 2*s + cout)
```

Almost any relationship can be expressed in logic, including ones that could never be proven such as the example above. After changing the statement above to mean that there exists *some* combination of values for those variables that satisfies the

relationship, any useful reasoning tool should allow it to be proven as a true statement. Proven statements are called theorems and marked with a turnstile:

```
⊢ ∃ a b c s cout . (a+b+c = 2*s + cout)
```

Sometimes a term can be a definition, ie. it gives a name to a relationship such that the name and the relationship are interchangeable in subsequent terms. A definition is created by setting the name equal to the relationship term[1], such as in the following definition of `full_adder`:

```
∀ a b c s cout . full_adder (a,b,c) (s,cout) = (a+b+c = 2*s + cout)
```

See Section 1.4.2 for the details of the logic notation used in this thesis.

1.1.1 Approach: organize architectures hierarchically

The first lesson is that large systems must be designed in a way which groups similar or closely-interacting components together, and which also shows only a single unified interface to external components. If possible, the entire system should be split into distinct parts which can be specified alone, and each of those parts be split into distinct parts, and so on until each part is fine-grained enough to be implemented with a few components in a simple manner. One good example of this is the formally verified Uinta microprocessor, which is organized into 4 different levels of abstractions to deal with 4 different issues.

In a previous work [GC94], the verification was complicated because one signal sent information from the basic gates to the logic which organized the subcomponents. This made the overall proof more complex, so we reworked the architecture and a few subcomponent specifications in order to clarify and distinguish the functional behavior of each part, resulting in a simpler proof [BJLW97].

[1] In HOL, definitions are created by the function `bossLib.Define`.

A fundmental assumption of this work is that better architectural planning will reduce the time and effort for humans to understand and develop each subsystem, and thus the system as a whole.

1.1.2 Approach: choose meaningful, distinct abstractions

This approach refers to the organization of a system by classes or types. Common wisdom dictates that objects and components are designed with "high-cohesion" (meaning that constituent parts of a component are closely related) and "low-coupling" (meaning that components are not dependant on one another). In other words, each object should be as self-contained as possible to deal with a specific concept, and there should be as few interdependencies between objects as possible.

This is critical for formal methods, where each component must be specified exactly in a form that often spans pages. The specification of an object with low cohesion (ie. with too broad a domain or too many functions) will be cumbersome and difficult to conceptualize as a whole. The specification of an object with high coupling (ie. with many dependencies on other elements) will be difficult to conceptualize and the elements will be hard to track.

There are various classifications of abstractions in the literature. Melham describes four types: structural, behavioral, data, and temporal [Mel89]. Liskov et al describe two types, parameterized and specification, and how they enable procedural, data, and iterative abstractions [LG86]. Alagar discusses the concept of abstraction detail and shows how various specification languages support it, especially in representation and operational abstractions [Ala98].

1.1.3 Approach: use model-checkers for the details, theorem provers for the abstractions

For large systems, the low-level details are often relationships between binary bits or simple signals between processes. Components at that level are usually simple

enough to verify automatically. This is done by model-checkers or, more recently, symbolic trajectory evaluation tools.

In contrast, when dealing with higher levels of abstraction, the interactions and meanings can be much more complex. Components at that level should be specified and verified with behavioral predicates that are more natural for people to understand. Theorem provers deal well with this type of formalism.

While not employed in this thesis, an appropriate mixture of tools could be beneficial in future work as explained in Chapter 6.

1.1.4 Task: specify devices with predicates

The first verification task is to specify devices with predicates. Hardware devices (and software functions) are usually written by showing outputs as functions of inputs. However, it is logically difficult to reason about functional expressions. Therefore, in addition to the input parameters, each device (or function) definition also takes the outputs as parameters and gives a boolean result, telling whether the inputs and outputs match the desired functionality.

To illustrate, consider a full-adder. It could be written as a function over three bitwise inputs, where the result is two bits that form a binary result. The `s` output represents a value of 1, the `cout` output represents a value of 2, and together they form the binary representation of the three inputs `a`, `b`, and `c`:

```
(s, cout) = full_adder (a,b,c)
WHERE full_adder (a,b,c) =
          (s := if (a XOR b XOR c) then 1 else 0,
           cout := a+b+c > 1)
```

For formal verification, we modify the definition of `full_adder` to be a predicate over the outputs as well as the inputs. This predicate is true for all combinations of inputs and outputs that satisfy the relation; it is false for those combinations that should not happen:

```
full_adder (a,b,c) (s,cout) = (a+b+c = 2*s + cout)
```

This is one of the most powerful tools in hardware verification because it is a simple way to achieve composability. Predicates facilitate composition since they can be combined with the conjunction operator, whereas functions can have various incompatible result types that are more difficult to compose together.

1.1.5 Task: implement devices to hide internal details

A device will expose inputs and outputs but hide all other internal details from outside view. Again, this is written as a predicate, but this time the definition more closely resembles the wiring logic:

```
adder_implementation (a,b,c) (s,cout) =
∃ s2 .
   s2 = (a AND NOT b) OR (NOT a AND b)
∧ s = (s2 AND NOT c) OR (NOT s2 AND c)
∧ cout = (a AND b) OR (a AND c) OR (b AND c)
```

1.1.6 Task: prove that "implementation implies specification"

After writing a predicate for the specification of the device and translating the implementation into logic terms, one needs to show that any situation satisfied by the implementation is part of the specification. The `implies` operation captures this relationship [2].

```
∀ a b c s cout . adder_implementation ==> full_adder
```

[2]The naive approach would use strict equality, but that is more difficult and is more work than necessary. Equality shows that the implementation covers every input-output combination of the specification, but it is sufficient to show that the implementation is a subset of the specification, meaning that all the input-output combinations are consistent with the specification.

1.1.7 Task: organize hierarchies of interfaces with abstract theories

As we define our predicates and verify some theorems, we begin to build the hierarchy of coarser-grained components and recurse the process until the final, system-wide predicate is verified. Abstract theories help organize this structure by clarifying the requirements at each level, and verifying facts based only on those requirements. They encapsulate much like interfaces or signatures do; they guarantee that, given inputs that meet certain criteria, they will provide service that satisfies the requirements.

One example of an abstract theory is a `group`, which is a collection of three associated entities: an associative function over two values, an inverse function, and an identity value. The three elements of a group are well-defined but abstract so that different sets and operations can be made into a group (such as `multiply`, `1 divided by x`, and `0`; or `add`, `zero minus x`, and `1`). However, no matter what entities are substituted for each value, if they satisfy that behavior then they all share some abstractly proven properties, such as the following:

- The identity element is unique for each associative operation.
- The identity element is determined by the associative operation.

In the next chapter, we demonstrate how framework software follows this pattern; namely, that the framework can be treated as an abstract theory which can be proven to have certain properties if built from components which are described abstractly.

1.2 Frameworks

The term "framework" may has different meanings in different contexts. For this thesis, we define a framework as a complete software system which contains components with well-defined interfaces that must be at least partially implemented by the deployers.

Frameworks fully implement most desired system functions, but leave less critical parts for developers to further customize. An example is an online message system which manages posts related to specific topics, but which leaves the input method unprogrammed. To complete the full system, deployers must supply the unwritten parts, such as the scripts to translate HTTP posts, email, voice messages, etc. into the API provided by the message system. Alone, a framework is not an entire system, but rather a complete implementation of a particular behavior with some interesting pieces missing.

We found approximately 50 examples of frameworks in our research. Half of those were software offerings, some for free and some costing a great deal of money. The rest were documented in some published form, many in books dedicated to the topic [FSJ99] [Rog97] [Lea95]. Some frameworks of general interest have been produced in recent years for common tasks:

- JUnit (www.junit.org) is a framework for testing. Deployers write their tests in classes that satisfy JUnit interfaces and the framework automatically runs batch test suites and reports the results.

- Open For Business is an e-commerce framework with a built-in order entry system. Deployers supply the products but can also insert their own functionality "hooks" which execute as the order process proceeds.

- The wftk framework implements a core of workflow functionality. Deployers write code to set up "processes" and the framework follows those rules to handle user "tasks" and their associated events.

- The EJB framework [MH99] handles transactions, persistence, and distributed processing for simple data objects. Based on the deployer's classes (called

"beans"), it generates code that interacts with the beans and a central server (called a "container") to handle all those issues and more.

To develop a framework, one must choose an application which could be applied in many contexts but which tackles a few closely-related, well-defined tasks. If the framework is too general, it is hard to apply to specific problems and it may be of questionable utility. If too specific and restricted, it will not have the broad appeal needed to grow into a variety of uses. If it tries to handle too many issues, it will wind up doing many things in a mediocre or complicated fashion rather than doing one thing well.

For those reasons, useful frameworks almost always develop over time in an evolutionary way [RJ96]. To some extent, the general and useful functionality may be achieved through good planning and design. However, understanding and implementing that type of functionality requires a few iterations of examples before pinpointing exactly how best to interface with the user-defined components.

1.2.1 Framework architectures

There are many ways that frameworks make use of components:

- The components are written to a well-known interface. The framework is built with methods that deal only with that interface, and the final system is instantiated with classes that satisfy that interface.

- The components are expected to have a very generic structure, such as publicly available `get` and `set` methods, which do the work. The framework discovers those methods and supplies a more powerful interface to them.

- The components have no predetermined structure. They are generated from another language or else examined by a pre-processor, and framework code is generated to deal with the components intelligently.

This thesis is focused on the first type of framework. The other two types are quite common and very powerful but lie beyond the reach of current verification methods.

1.2.2 Framework Properties

Most systems that become popular are designed skillfully (though in an ad hoc manner), deployed quickly, and updated frequently. The industry has shown that rapid time-to-market is more important than correctness or even ease of use for marketplace acceptance. It will never be practical to verify software which is frequently modified and thus still undergoing change. Even small changes to an interface can mean large changes in the formal elements, especially the specifications which are a human-intensive endeavor [3].

In order to make the results of formal analysis worthwhile, we must find systems which are fairly stable but which have formal properties for which formal analysis would be beneficial, especially to reuse for their own purposes [4]. Following are the three properties a system or subsystem needs for formal analysis be worthwhile [5].

1. **System must be stable enough that formal results stay relevant.**

[3] Swartout et al [SB79] claim that it is not possible to perfectly separate specifications from implementations due to imperfect foresight and physical limitations. We agree that specifications evolve as a result of practical concerns but we also see examples of mature systems fitting our criteria and that have progressed to the point that they can be described with well-defined specifications entirely separate from their implementations.

[4] Some systems meeting these criteria might be verified exactly once upon completion, but not reused for any other purpose. This may be desirable for political purposes, where someone must analyze the system in various ways to engender confidence in the system's correctness. These would be costly systems tailored to one large or many homogeneous organizations. Many of the ideas in this thesis may apply to such systems, but that is not our focus.

[5] Although the given criteria are important for any formal methods work, most research focuses on technologies and tools for software in general; we believe it is important to identify a specific problem domain along with the tools and approach that are best suited for its formal analysis.

Since the process is expensive, and specifications are difficult to change, we see no advantage in analyzing a short-lived artifact.

2. **System must have formalizable behavior.** Even though a system is stable, it may not have behavior which we know how to express formally. An example might be a ray-tracing engine, where quality is judged by the visual clarity or raw speed.

3. **System's formal results must be useful, preferably in a different systems.** Our formal methods must serve a useful purpose. Often the benefit is that problems were found and eliminated in the process of formal analysis. But we must extend this benefit to future users or disparate organizations.

Software frameworks satisfy our criteria:

1. **They are stable, so we keep our specifications and verifications.** Whereas most applications evolve over time to add more features, a good framework's interface stays constant and the formal analysis work will not become obsolete. Of course, a framework's implementation may improve over time, but as long as the interface does not change, most of the formal work will stay relevant.

2. **They have well-defined behavior, so they can be specified.** Other large programs are built to serve a variety of needs and often include features of convenience. This makes their behavior hard to formalize, both for the system as a whole and for each specific feature which may be interdependant with other features. Frameworks satisfy this criterion.

   Frameworks usually implement functions that are difficult or time-consuming to implement correctly from scratch, and these are the types of functions that

can benefit the most from formal methods.

3. **They are reused, which increases the payoff.** While not everyone who uses a framework will pay attention to formal specifications, some will use it to clarify meanings and others may create fully verified systems. Also, just like many reviewers help solidify theorems in mathematics, many users help clarify and validate formal analyses over time.

1.2.3 Limited, well-defined viewpoint results in easier proofs

There are two perspectives of a framework, one for each role involved in its use: the designers of the framework code and the deployers who write components that fit in the framework. Typically, the deployers also install and run the framework, which becomes their customized version of the product.

From the designers point of view, there are a few very specific system-wide behaviors to implement, behaviors which can be tweaked depending on the deployers' components. So, in addition to the overall behavior, the designers must specify the behavior they expect from the components or possibly what they expect the components *not* to do. The designers verify the overall system correctness using the component behavior as assumptions.

On the other hand, the deployers see an unfinished system where some framing is left exposed for them to build on. This framing is explicit, with documentation for all the available hooks (functions and objects) such that the deployers can customize as they wish. Also, for our purposes, the designers have given several formal statements describing the exposed framing, the components that the deployers are expected to build, and one overall formal statement of correctness for the finished product. Lastly, the designers have provided a proof of correctness of the system including the deployers' components based on the correctness of those components.

Therefore, the deployers merely need to prove that their components satisfy the required behavior and the result is a finished, verified system.

These roles are explained in depth along with an example and illustrations in Section 3.2.

1.3 Software System Formalisms

Floyd [Flo67] wrote the seminal work on software verification in 1967, where he used a flowchart to model programs and gave a meaning to each statement such that an overall behavior could be derived. Hoare [Hoa69] added to this by defining an axiomatic semantics for program execution, and others have built on this foundation to include most programming constructs in modern languages. This line of research focuses on the meaning of programs as built from primitive program statements and their data. Just as programming languages have evolved to more powerful abstractions, programming semantics now work on the level of abstract data types; see [LG86] for specifics[6].

There are now various semantic models for program verification [GPZ94] [Ala98]. Each comes with its own rules for the basic programming constructs (eg. loops, function calls, etc), and each has strengths and weaknesses for particular types of programs. Since our work analyzes programming statements, we reason with an "*operational*" semantics: this is a straightforward mapping of each statement into a mathematical form which behaves similarly but is based on logic. We do this to make the form as familiar as possible, realizing that it may make some proofs more complex.

[6]Even with highly abstract models and advanced tools, program verification remains a difficult and time-consuming task. We do not claim to significantly simplify this task; rather, we hope to describe and demonstrate a combination of methods with a problem domain where program verification can be practical.

In this section, we describe specific approaches to software reasoning and verification. We then describe the salient features of the main formal languages and tools. See [GPZ94] and [Ala98] for a more detailed discussion.

1.3.1 Verification Approaches

Software verification approaches are divided into "model-checking" and "theorem-proving".

1.3.1.1 Model-checking

Model-checkers automatically decide whether an implementation satisfies certain properties and otherwise gives counter-examples, usually through efficient modelling and most often in temporal logic. However, they only work with decidable problems which restricts their expressiveness.

Within the model-checking arena, the most common tasks are determining conformance to safety and liveness properties. Can the machine ever access an unauthorized area in memory or get in an inconsistent state? Can it ever get caught in a state such that it refuses to respond or do anything useful? Many questions can be phrased one of these two ways, so this approach will remain viable for many problems.

1.3.1.2 Theorem-proving

Theorem-provers deal with very abstract entities and rules, and they can represent virtually any behavior and relationship. However, they require much more human insight and intervention. There are two major approaches for using theorem-provers to verify machines.

The first is "*refinement*". Designers begin with the overall (or most abstract) system specification. From there, the design is transformed by well-understood rules into a more concrete (or less abstract) form. There must always be a mapping between the forms, where the mapping follows rules defined by the logic.

The other theorem-proving approach allows us to independantly develop specifications at different levels of abstraction. In other words, there is no explicit user-defined mapping between specifications. When the specifications are formally verified, the proof may be considered an implicit mapping.

However, even that distinction is somewhat contrived, since most complex logics support both approaches. In fact, the only real classifications within theorem-proving are along the lines of the tools themselves: each combination of logic and tool has its own features and trade-offs, and so each system can be considered its own approach.

We take a theorem-proving based approach because the behavior we wish to explore is expressed in abstract data types, and model-checkers use less expressive logics or temporal logics which are meant for different types of specifications.

1.3.2 How this thesis is unique

As we show in Chapter 2, there are many other groups doing research related to this work. This thesis is unique because it combines work in the following three areas:

- formal modelling of object-oriented constructs
- formal analysis of software at the system level
- identifying a domain in which formal results can be shared

1.4 Thesis Organization

1.4.1 Outline

Chapter 2 delves further into other methods and tools that are closely related to this work. Chapter 3 explains our thesis in detail and introduces the key concepts used to support it. Chapter 4 develops other techniques needed to make the thesis possible. Chapter 5 shows how to apply the concepts and techniques to three significant examples. Chapter 6 summarizes these ideas and suggests further research to extend these ideas and tools to make them even more useful.

1.4.2 Terminology and Notation

We assume the terms "`object`" and "`class`" are standard. We always use the term "`interface`" to mean the definition of an abstract data type, ie. the syntax and semantics that classes must implement at very least; some languages call this concept a "`signature`" or "`module`". We use the terms "`function`" and "`method`" interchangeably, and we assume there are no functions without an associated class; some languages call this a "`message`". A "`field`" is a datum belonging to an object; some languages call this an "`attribute`" or "`member`".

Programming code is given in a slightly different display mode from logic terms. As shown previously in this chapter, logical statements are presented as follows:

```
electric_eel_class =
<| zap :=
   (λ  . (state, obj) state' .
     energy_potential obj state < energy_potential obj state')
|>
```

Programming code is presented as follows:

```
public interface Shocking {
  public void zap(Physical object);
}
```

The logic notation in this thesis is standard, and is the same as the notation used in the Higher-Order Logic (HOL) theorem-prover. For example, all logic operators have their standard meanings: $\wedge$ is conjunction, $\vee$ is disjunction, $\Rightarrow$ is implication, $\neg$ is negation, $*$ is multiplication, $<=$ is less-than-or-equal-to, etc.

The notation for logical records (ie. groups of elements where each position in the group is given a name) needs some explanation. Records are shown by the delimeters

< | and | > along with the assignment operator := to name the slots. For example, the following term declares a record with two positions, named `list` and `sum`, having the values of an empty list (`[]`) and 0 respectively:

```
<| list := []; sum := 0 |>
```

The notation used for logical types also requires explanation. Type declarations are prefaced with the keyword "`type`"[7]. A type can be any of the following:

- a primitive type of number (*num*) or boolean (*bool*)
- an anonymous type, ie. a type which is free to take any value (written with a prefix of ' (apostrophe), such as $'x$ or $'any\_class$)
- a list of another type (written with a suffix of "list", such as *bool list*)
- a pair of types (combined using #, such as *num#bool*)
- an alternative of types (combined using |, such as *bool|num*)
- a function from one type to another (combined using →, such as $'a\ list \rightarrow num$)
- a new type with a constructor (written with a name followed by the word "of" and another type, such as $TwoNums\ of\ num\#num$)
- a record type (written with a slot name and type for each element, such as $< |element :\ 'a;\ measure :\ 'a \rightarrow num| >$)

[7]In HOL, types are declared with the function `bossLib.Hol_datatype`.

Chapter 2

Related Work

There are several other techniques to design and analyze software systems. This chapter describes those that are most relevant and well-developed.

Section 2.1 lists a number of other approaches aimed at developing formal methods for the architectural level of a software system. Many of them address the same issues of composing components to build large, object-oriented systems. In contrast, this thesis does more to share component specifications and proofs.

Sections 2.2 through 2.10 introduce other formal analysis or object-oriented design techniques that are related to this thesis. However, none of them addresses all three of the unique features of this thesis (see Section 1.3.2).

2.1 Other Formal Architectures and Design Patterns

The following approaches each use formal methods for software at the system level. This thesis is unique because it also identifies a type of software system, along with the appropriate formal tools, for which it is practial to share specifications and reuse proof results.

- The Composable Software Systems Group [Gro01a] at CMU has a variety of projects ranging from new architectural languages to component-based teach-

ing tools. One of the pioneering papers on software architecture [GS93] is from this group. One of the fundamental ideas in their approaches is the use of "*connectors*" as first-order elements, along with the components that they attach together. Their formal work is done in Wright, an new language based on temporal logic. Following are some of the more advanced projects in this group:

- Wright is a formal architectural language mentioned earlier (see Section 2.2) which is noteworthy because one result of this project is very comprehensive temporal model for the communication protocol between EJB server and bean methods [SG99]. In particular, it shows how exceptional cases are handled and verifies that the server will always recover after an error state in a bean. Temporal models are used to analyze requirements for protocols such as detecting deadlock or ensuring liveness; since this work (and the logic in this thesis) is focused on more behavioral requirements, Wright is a complimentary approach to ours.
- The Able project works with representations of architectural "*styles*" which classify architectural approaches. Tools based on these styles can then help create environments to support design and analysis of a particular system.
- The Acme architecture description language not only models architectures but also works as an intermediate language for other architectural descriptions.
- The Venari project aims to create the database infrastructure for objects based on their formal semantics.
- The group maintains a list of "*model problems*" for software architecture. These can be used as illustrative examples for education or as reference implementations for comparing approaches.

- The Software Composition Group [Gro01b] at the University of Berne also studies composable software, such as in [NTea95], with a focus on object-oriented approaches. Like the Composable Software Systems Group, they use "connectors" to combine components and then add another concept of "*glue*" to overcome interface mismatches. Their work is also based on a variant of temporal logic called pi-calculus.

- The B Method [Abr96] is based on a formalism called Abstract Machine Notation which can represent system specifications at a high-level as well as lower levels [SS98]; this is similar to refinement (see Section 2.5). Most B tools include executable code libraries to implement many low-level specifications, allowing refinement all the way to an executable system.

- Rapide [AG02] is another set of architectural tools based on its own formal description language. Like the B method, it can specify a system at multiple levels of granularity. It also offers a variety of prototyping and testing tools.

- Amnon Eden et al. [EHY99] have developed a formal language named LePUS for modelling design patterns and have used it on some of the the canonical examples [GHJV95] as well as patterns of their own. LePUS has a visual representation resembling UML descriptions (see Section 2.6). Unfortunately, there is no tool support for LePUS.

2.2 Formal Specification Languages and Associated Tools

There are a number of tools backed by formal specification languages which could support this type of work. Below are those most closely related to this approach. There are other tools logics and specification languages besides those below, but they had unacceptable features (eg. expensive, closed-source).

For this work, we only considered languages with theorem-proving tools. The specifications we wrote required a language with a high level of abstraction, and higher-order logics are the formal languages with the greatest degree of abstraction.

2.2.1 Higher-Order Logic (HOL)

HOL [Ge93] is a theorem-proving environment whose specification language is a strongly-typed, higher-order calculus. One distinguishing feature is that all reasoning is based on eight simple, fundamental axioms and as a result it is very unlikely that there is a bug in the system that would allow incorrect proof results. It is written in ML and has been extended with a large number of libraries and proof tools. Some automatic tools define non-primitive and nested functions, and some tools even include basic model-checking functionality. Since interaction with HOL is done through the programming language ML, it allows a great deal of flexibility when doing exploratory work.

HOL's strength is also its weakness: the language is so expressive that there is very little automated reasoning, so the proof effort is almost entirely the burden of the researcher.

We chose HOL for examples in this thesis for two reasons. First, because we are familiar with it; formal notations and tools have a steep learning curve. Second, because it is open-source; some of our work stretched the type-definition capabilities of HOL and uncovered some bugs, so it was critical that we were able to fix them ourselves.

2.2.2 Other higher-order logics: Isabelle/HOL, PVS, COQ

These are other versions of higher-order logics which include theorem-proving tools.

Isabelle [Pau94] is a generic theorem-prover; given a specification language and related axioms, it provides the infrastructure for reasoning and proof. There is a

version similar to HOL called Isabelle/HOL, and the Bali project 2.3 is written for this platform.

Although the existing Bali project was a useful reference for our work, we found it difficult to initiate new research in Isabelle because knowledgeable experts were not accessible and documentation was not helpful.

The Prototype Verification System (PVS) [ORR+96] is another strongly-typed, classical higher-order logic system. It includes many features such as dependant and uninterpreted types and parameterized theories. The tools provide a gread deal of help with reasoning such that most obligations can be proven automatically.

PVS is a proprietary system and would be impractical for exploratory research such as this.

The Calculus of Inductive Constructions (Coq) [DFH+93] is based on a "*constructive*" logic, as opposed to the "*classical*" logics of the other higher-order logic systems. Constructive logics only allow reasoning based on terms constructed from more primitive terms; they lack the notion of the "*excluded middle*" where a statement is true if not false and vice versa, so every assertion must be built from other known assertions. Proof-checking in Coq is actually type-checking. Coq allows set and propositional sorts in addition to types. Its specification language is said to be very natural for the formalization of mathematical concepts [Zam97].

Coq's meta-language is not as powerful as in other systems, and large-scale problems tend to become unweildy because all terms include their reasoning.

2.2.3 Other specification languages (with tools): VDM, Z, Larch, OBJ, Wright

The Vienna Development Method (VDM) [Jon90] is a combination of a specification language (VDM/SL) and a semantics of program refinement to specify systems in very abstract or more concrete terms. It is therefore well-suited for working with large

systems at increasingly lower levels of abstraction, which may prove useful for a disciplined engineering process. There is also a new variant that supports object-oriented extensions.

Unfortunately, the VDM tools are not as accessible as other tools. Commercial development environments exist, as do theorem-provers and other tools for selected platforms, but each of these options is too expensive for the purpose of this research.

The Z (pronounced "*zed*") notation language is a strongly-typed specification language. It is based on both set theory and first-order logic, making it natural for many types of specification. Z includes refinement much like VDM, though refinement in Z is done to other schemas and not to implementations. It is best suited for representing models or "*schemas*" and includes notation for schema composition. Lately, there have been many variants to support object-oriented constructs [Stu93].

Z is for model-based specifications and does not have good tool support for proof. Most tasks consist of relating schemas to check for consistency among different models. Z seems well-suited for our purposes; we later advocate further research in this area for Z. (See also the explanation of OBJ below.)

Larch [GHG+93] is a specification language and proof tool with a fundamentally different approach. It is specifically for software, and formalisms consist of an abstract specification in the Larch Shared Language (LSL) and also a concrete specification in a Larch Interface Language (LIL). There is a separate LIL to support each programming language and its features, but each is written in a form that can be related to the language-independant LSL. There is a theorem-proving tool, the Larch Prover (LP), which can be used to check properties of Larch specifications, such as consistency and completeness.

Larch is well-suited for specific programs in many languages. However, the LSL and LP (based on first-order logic) without mechanisms like OBJs "parameterized

programming" do not support the higher-order nature of framework systems.

OBJ [GM00] is an algebraic specification language, where all functions are expressed as algebraic expressions. OBJ includes an environment for executing specifications with rewrite rules, which is the method for verifying systems.

OBJ is based on equational logic, which would usually prohibit its use for "higher-order" programs such as frameworks. However, OBJ has some unique features such as parameterized programming which may support this type of work.

Wright [All97] is a formal language based on temporal logic, desiged to reason about the component-connector style of architectures. Temporal logic is especially well-suited for reasoning about aspects of concurrent events and find problems such as deadlock. The reasoning can also be done automatically.

In contrast to Wright, our aim is to reason about different kinds of system behavior at different levels of abstraction. For example, temporal logic will not help determine whether components conform to arbitrary abstract data type specifications.

2.3 Java and other Programming Language Formalisms

This thesis offers concrete programming examples along with a formal representation used for proofs. There have been many other programming language formalisms but none that were readily adaptable to the HOL system in a way that would support our method.

Following are the key references for programming languages in general, especially object-oriented languages. At the start of our research, only the last three had tool support for sharing mechanically-checked logic results and none had tool support for abstract theories as presented in our method. For those interested in theorem-proving tools that deal well with different levels of abstractions (like HOL in this thesis), the last three are most similar to this work.

1. Liskov and Guttag's work includes reasoning about software abstractions and

concrete representations [LG86].

2. Abadi and Leino developed a taxonomy for object-oriented languages organized by their features; it includes a logic to represent constructs in each type of language [AL98],.

3. Wahab presented a formalism for object code including an abstraction language to relate it to higher-level constructs [Wah98].

4. Leino developed an axiomatic semantics for a specific object-oriented programming language (called Esctatic) [Lei97].

5. Oheimb designed a Hoare logic for a subset of Java which is provably complete; they claim it is the first provably complete logic for an object-oriented language [vO01].

6. Norrish developed an axiomatic set of rules for C program verification which is build on a basic operational semantics for the language [Nor96].

There are many analysis approaches dedicated to the Java language [PHM99] [LSS99] [FF00] or its variants (eg. JavaCard [BDJ+01] [PvJ00]), and formal semantics have even been presented for the language [AF99] [vO01] and its bytecode instructions [Qia97]. However, at the time we began our research only the Bali project had published any formalisms or tools, and its purpose was to reason about the Java language itself; this was too complex for our needs since we only reason about programs written in Java. Along with Bali, the following formal tools are worth investigating at the time of publication:

1. The Bali project has formalisms for Java at both the source [vO01] and bytecode [Pus98] levels. They have used these formalisms to show that a subset

of Java is type-safe [ON99] and that a bytecode verifying algorithm is correct [Nip01].

2. The Java PathFinder [HP00] converts Java programs into the PROMELA temporal logic which can be analyzed by the SPIN model-checker for various protocol properties. It can also take annotated Java programs to check more abstract user-defined predicates with the Standford Validity Checker.

3. Park et al. [PSSD00] show how to compile multi-threaded Java programs into executable model-checking programs. The resulting programs are executed to verify the absense of deadlock and assertion violations.

4. Jakarta [BDHS01] is a combination of tools for specification and proof of the JavaCard platform. For verification it uses Coq, another higher-order logic related to HOL.

2.4 Viewpoint or Consistency Checking

The Z community approaches the verification problem from a slightly different angle: rather than verifying properties which are written independant of the problem domain, they write formal descriptions of the problem (or the ideal solution) from more than one vantage-point. Then they verify that the descriptions are all consistent with each other. This approach is sometimes called consistency checking for multiple views, viewpoints, or perspectives [Jac95] [BDBS99] [FKN$^+$92]. Although the formal specifications may not be generally useful, they may prove to be much easier to write and reason about.

2.5 Refinement

Refinement [Bac80] is a very general methodology: one begins with a high-level statement of system behavior, transforms it with trustworthy mappings into more a

concrete program, and repeats until an implementation is generated. This approach is discussed in connection with the B method (Section 2.1 and VDM and Z (Section 2.2).

2.6 Unified Modelling Language

Another method which has gained some popularity is the Unified Modelling Language (UML) [BRJ99] . The basic tenet of UML is that there are multiple ways to communicate about processes and protocols, each with its own language and models and each applicable to a specific type of problem or at a specific time in the software lifecycle. There is a core language and there has been some research into formal semantics for it and there are competing "system modelling" approaches such as OSA [EKW92] with even more formality. But these approaches do not yet have the reasoning infrastructure necessary for program verification.

2.7 Extended Static Checking

Extended static checking [DLNS98] aims to check more complex errors in a program. Example applications include null dereferences, array index bounds errors, and deadlock situations. This is one of the more advanced statement-level methods but does not address issues of defining abstract datatypes or sharing specifications and proofs.

2.8 Abstract Interpretation

Abstract interpretation [CC77] is a statement-based model using abstract operations. Rather than calculating with concrete primitives, such as an integer or a string, abstract interpretation primitives are abstractions of the values. For example, an abstract view of the integers might be only whether the value is positive or negative; an abstract view of threads may simply be the active, waiting, or finished state of the thread. Given an interface between the two methods, this approach may prove useful in conjunction with our work.

2.9 Object Logics

Researchers have recently been investigating "object" logics [AL98] which combine data and methods together as a logical unit. This is in contrast with logics for theorem-provers which are mainly functional, and the difference is akin to the difference between object-oriented and functional programming languages. However, rather than use a new, unconventional object logic (for which we must develop new tools), we translate the object-oriented concepts into a traditional higher-order logic to take advantage of the well-developed tool base. In this work, almost all translations could be automated so that the functional approach is not a major drawback.

2.10 Design-By-Contract

Programming language "*asserts*" are runtime boolean checks for correctness within an executable program. Meyer first added asserts in a disciplined way to his object-oriented language Eiffel [Mey97]. He calls this "design by contract", where designers write boolean language expressions for method pre- and postconditions and class invariants. The expressions are meant to be executed at run-time, so this only allows statements from the source language. This method does not support more powerful specifications to determine more abstract meanings or undecidable properties; however, the approach is simple and encourages programmers to write precise specifications for debugging and documentation.

Chapter 3

A Formal Theory for Frameworks

This thesis shows that, while impractical for systems in general, framework-based software architectures are a type of system for which formal analysis can be beneficial and practical over the life of the system. In this chapter we preview the process of framework specification and verification (3.1) and introduce the specific technical tools used to accomplish our goal (3.2).

3.1 Stages of Framework Verification

This section shows how higher-order logic constructs can naturally and faithfully represent the behavior and structure of software in a manner similar to how they are used to represent hardware. Following are the steps of framework analysis broken down into two sets of responsibilities, the first set being the job of the framework implementor and the second set being the job of the component implementor. They specialize the three hardware verification tasks from Sections 1.1.4 through 1.1.6, applying them to framework software verification.

1. The framework implementor creates the following (possibly with feedback from the component implementors):

(a) **The framework specification** gives the behavior of the final system, usually defined in terms of interfaces (Section 3.2.1.2).

(b) **Component specifications** give behavioral definitions of user-implemented components that interface with the framework (Section 3.2.1.1).

(c) **The framework class implementation** forms an abstract theory of the framework (Section 3.2.1.3); it is built from the component specifications and the logic terms translated from the actual program code (as shown in Chapter 4).

(d) **The framework verification** breaks down the goal into the component abstract theories, then uses domain-specific tactics to verify that the framework code and conforming components together yield the desired overall framework behavior (Section 3.2.1.4).

2. Next, each component implementor creates the following when they deploy that framework:

(a) **Component implementations** written in programming code are translated into logic, just like the framework implementation (Section 3.2.2.1).

(b) **The component verification** shows that the component implementation meets the specification required by the framework. (Section 3.2.2.2).

(c) **Verification of the final framework properties** automatically follows from the framework and component verifications (Section 3.2.2.3).

(d) **Verification of the final client properties** follows directly (though not automatically) from the framework properties (Section 3.2.2.4).

3.1.1 High-level illustration

Take a framework system that handles stock purchases. The system might provide a useful interface, allow deployment on the internet, guarantee a certain response time, or even guarantee that certain legal requirements are met. Deployers of the system (such as stock sellers) might be required to write a module that communicates with external systems that sell stocks; that module would have to ensure that the transaction succeeds or fails gracefully in a given amount of time.

Figure 3.1 is used throughout this thesis to better explain the interactions of the key elements of this system and illustrate our verification process. Each part is explained throughout the example that follows.

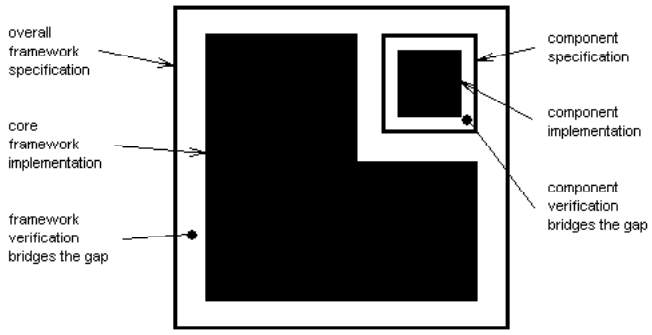


Figure 3.1: The "framework-component" diagram for visualizing framework parts

Say a company named Frieda's Framework Foundry will build the framework described above. This framework's feature list includes qualitative elements, such as ease-of-use and speed-of-deployment, along with quantitative elements which are verified by means other than formal methods, such as platform independance and benchmark measurements. But we are most interested in the formally verifiable features, such as security and transaction consistency. Cory's Component Construction will deploy and run the formally verified stock purchasing system; he will purchase Frieda's framework and write his own component that will interface with his com-

pany's network of stock sellers.

3.1.1.1 Framework Implementor

Following are the four steps Frieda takes to build her framework and achieve verification results that will be useful to deployers such as Cory.

(a) The process begins when Frieda specifies exactly what services the framework will provide, which are security and transaction consistency. The result of this step is a written formal specification to which end users could refer to see if this system satisfies their requirements.

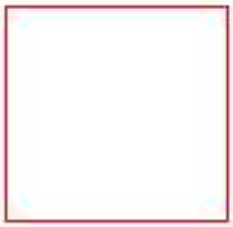


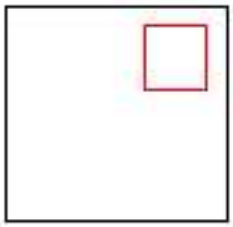
Figure 3.2: The framework's formal specification is the encompassing statement of correctness.

(b) Next, since Cory will deploy the system for his own particular needs, he must receive a formal specification for the component he will implement. So Frieda has to write another formal specification for each component that deployers must implement. As described above, Frieda's example framework requires a communication module for purchasing from external systems; the result of this step is another written formal specification to which Cory will refer when he implements this communication component.

Note that these first two steps of component and framework specification are often developed together, so they are sometimes introduced in a different order or even at the same time in this thesis. In general, we found that all the stages of framework verification are interdependant and choices at later stages may affect the work of earlier ones, but the order we present in this thesis is the order in

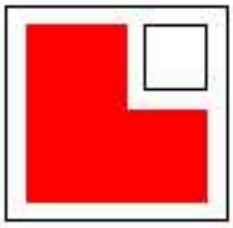
which stages begin; later stages cannot start development until previous stages have begun.

Figure 3.3: The component specification is a critical part of the overall framework specification.



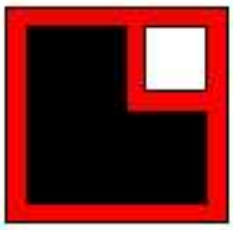
(c) Next, Frieda implements the framework. The result is program code that implements the entire framework functionality, excepting only the component(s) Cory will deploy later.

Figure 3.4: The implementation "fills in" the details of the framework except for the component functionality.



(d) Now Frieda formally verifies her implementation, proving that her program code satisfies the overall framework specification when combined with her customers' components. The result is a proof script that anyone else can check with a theorem-prover to see that the framework is correct.

Figure 3.5: Verification shows that the implementation with the component meet the overall framework specification.



That is the end of Frieda's work. She now has framework and component specifications, an implementation, and a proof which can be delivered to any of her customers. Each customer can implement the components for the framework differently, but as long as they satisfy Frieda's component specifications their entire system is guaranteed to be correct for this property. That is the point of this thesis and many of our examples end here; however, a framework is not actually deployable until the following tasks are completed.

3.1.1.2 Component Implementor

This is where Cory's work begins. He will proceed through the following four steps, building on Frieda's results, to deploy a system with the properties that her framework supplies.

(a) He must develop the component that plugs into Frieda's framework. Again, this is an implementation written in programming code.

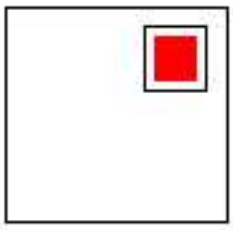


Figure 3.6: Deployers implement the component according to its specification.

(b) Now Cory proves that his code satisfies the component requirements. The deliverable is a proof script for the component implementation.

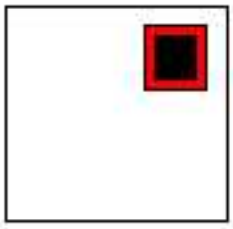


Figure 3.7: Component writers then verify that their implementation fits into the framework properly.

(c) Now Cory's component proof can be combined with Frieda's framework proof to verify the system they have cooperatively implemented. This task is automatic due to the structure of framework and component theorems.

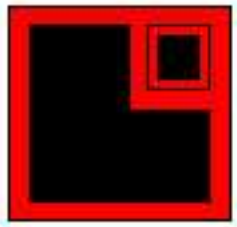


Figure 3.8: The correctness of the entire framework follows automatically from earlier proofs.

(d) Cory's ultimate goal is to verify that his system works correctly for his customers and that each of their stock purchases will succeed correctly or else completely fail. So he uses the framework properties to fulfill properties that are stricter than those in the framework.

In particular, he will guarantee to his customers that he will not charge their credit card without transferring the stock, and he will guarantee to the seller that he will not transfer stocks to the customer without obtaining the customer's money. Since his components have been guaranteed to satisfy the framework transaction properties, these more specific properties are now easier to prove.

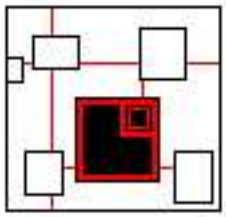


Figure 3.9: The client's critical properties can be proven more easily with the framework properties.

This final step is where framework users see the benefit of the earlier framework verification. Because the framework was previously proven to preserve transactional properties, this final proof is much simpler for the deployer because its properties are

a direct consequence of the earlier results. In other words, he is able to leverage the framework designer's proof to get his own verified system with very little effort on his part. This is the most significant benefit of framework verification.

3.2 The Stages in More Detail

The remainder of this chapter elaborates on each of these eight steps, shows in more technical detail how each step is accomplished, and develops a small illustrative example. This section explains each step and includes logical details but does not discuss how the details were generated. The details are the focus of Chapter 4; readers without a background in logical models of software may wish to read that chapter first.

First, let us point out a key enabler of this work: all object data is accessed through methods. No internal fields are directly available to the users of any class (though constants are allowed). Internal data can always be made available to clients when necessary with `get` and `set` methods, though of course such access should be allowed only with care.

We have found this to be a general rule of good design; it is simply encapsulation in action. However, it is especially true for formal methods work: method-based interfaces are the focus of all specifications since they encapsulate the behavior into abstract data types. For those to whom this seems unnecessarily restrictive we offer the following in its defense:

- Modern OO languages such as Java and Smalltalk only allow methods and constants in interfaces.

- Data should be hidden by default to encourage well-defined abstractions. It should cost extra effort to make data available externally so that programmers do not unwittingly allow accesses to data in a way contrary to the programmer's

intentions.

- An interface's specification might be written in terms of internal data members, but only such that the data is part of the specification. If a specification depends on internal representation details then either those details are critical to the abstract data type and should be external and understood by users or they are not critical and the specification should be rewritten without them. Put another way, if the data is an important part of the specification then the only additional work is in changing the data accesses to "getter" and "setter" methods since they are already an integral part of the definition.

The interface obligations are used in different ways by the two parties in component-sharing:

- Framework writers assume that the component code satisfies the predicates and use them as assumptions when building and verifying framework systems.
- Component writers must offer proof that their code satisfies the predicates.

In mathematical terms, the framework writers implement an "*abstract theory*" of their system: they take advantage of a few key facts about the building blocks to create a larger system, realizing that the system's correctness depends explicitly on the correctness of each of the blocks. An abstract theory is a collection of related items which can be proven to satisfy certain theorems if its elements satisfy certain more basic properties. Frameworks are structured precisely the same way: an abstract system is created but is essentially useless until some primitive components with certain features are supplied.

Throughout this thesis, the suffix "`_sig`" marks type signatures and and the suffix "`_obligs`" marks the behavioral predicates. The suffix "`_obj`" is used for types that

contain object data and "`_ptr`" marks object variables that are references to that data found somewhere in memory.

As for implementations (Sections 3.2.1.3 and 3.2.2.1), Java is the language used for all examples in this thesis but these verification principles are designed to apply to any object-oriented language. Chapter 4 shows how to translate object-oriented constructs into logic terms; although Java is used to illustrate concepts, the techniques are generally applicable.

As a running example, consider a framework that `Calculates`; it coordinates the actions of simple functions and provides services such as repeated operations, macro definition, history tracking, and so on. The framework will use `Operates` components which implement a single method (named `operate`) that takes two numbers and outputs a third. One implementation of this framework is a class named `Calculator` below, and a sample component used to create a working system is a class named `Checksum`. They fit together in the framework diagram as shown in Figure 3.10.

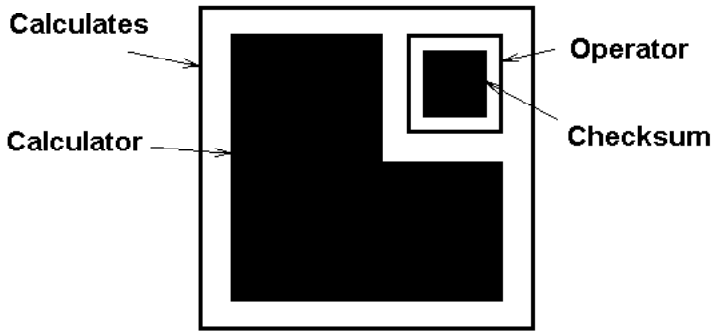
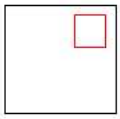


Figure 3.10: "Calculates" framework-component diagram

3.2.1 Framework Implementor

The next four sections describe the four tasks of the framework designer in more detail.

3.2.1.1 Specify component signatures and behavior



Class and interface specifications come in two parts: the syntax (or signature) and the semantics (or behavior) of the class methods.

The syntax of each interface specification is a record type definition containing a field for each object method. Each method is a predicate, so each signature returns a boolean (`bool` below) and takes three arguments: a reference to the object data, the input parameters, and the output values. ("Static" class methods are similar but without any reference to an object.) The signature is declared as follows (see Section 1.4.2 to review the notation for types):

```
type iface_signature =
<|
   method1 = 'object_ptr
             -> ('input_state # 'input1_types)
             -> ('output_state # 'output1_types)
             -> bool;
   method2 = 'object_ptr
             -> ('input_state # 'input2_types)
             -> ('output_state # 'output2_types)
             -> bool;
   ...
|>
```

The semantics of any class or interface are written as a predicate over the methods. The predicate takes a single argument which is the record of all the methods contained in the class. Inside the predicate are statements about the relationships of the methods and/or their arguments, possibly over periods of time. The following is a general example:

```
iface_obligs class =
  ∀ iface_object_ptr state1 state2 state3 input1 output1 output2 .
  (input1 <= output1)
  ∧
  class.method1 iface_object_ptr (state1, input1) (state2, output1)
  ==>
  class.method2 iface_object_ptr (state2, output1) (state3, output2)
  ...
```

Simple interfaces are self-contained, needing no reference to other system components. However, more complex interfaces are specified in terms of other interfaces. This is done with an existential quantifier, such as the following where the interface is defined in terms of the behavior of the `library_component` interface.

```
iface_obligs class =
∃ lib_class::library_component .
∀ lib_object_ptr iface_object_ptr state1 state2 input1 output1 .
class.method1 iface_object_ptr (state1, input1) (state2, output1)
==>
lib_class.method1 lib_object_ptr (state1, input1) (state2, output1)
...
```

As an example, we begin the `Calculates` framework by specifying the responsibilities of the `Operates` component. We specify the framework second because its definition depends on some characteristics of the component.

The signature for `Operates` components is as follows:

```
public interface Operates {
  public int operate(int x, int y);
  public int identity();
  public int maximum();
}
```

Since any compiler can typecheck the signature, for this example we also require that the function throws an overflow error for any calculation outside the range of 0 to `maximum`. Following are the modified Java declarations [1] [2]:

```
public interface Operates {
  public interface PosNumber {}
  public class Number implements PosNumber {
    int value;
    public Number(int val) { value = val; }
    public int getValue() { return value; }
  }
  public class Overflow implements PosNumber {
    String message;
    public Overflow(string val) { message = val; }
    public String getMessage() { return message; }
  }

  public PosNumber operate(int x, int y);
  public int identity();
  public int maximum();
}
```

The remainder of this section presents the behavioral specifications for an `operates` component. First, because the `Operates` interface is defined in terms of the concrete classes `Number` and `Overflow`, the associated class data must be defined before other specifications. This is often the case since more complex datatypes often make use of simpler concrete classes to specify their meaning.

[1] We collect other interface and class declarations (such as `PosNumber`) inside the `Operates` class; this does not change the semantics but we must refer to those entities by prepending them with "`Operates.`" ("`Operates`"-"dot") in subsequent Java code.

[2] Overflows are usually handled by exceptions. Exceptions can be implemented here by adding another element to the state (see Section 4.2) such that each statement would be executed conditionally based on the current exception state. Although this example emulates exception behavior, it is contrived to demonstrate simple framework behavior and is not representative of good object-oriented design.

Following are the auxiliary `number_obj` and `overflow_obj` types along with their superclass `pos_number_obj`; only the first two hold data, and the latter is simply a wrapper around either one:

```
type number_obj = Number of num


type overflow_obj = Overflow of string


type pos_number_obj =
   Pos_Number_Number of number_obj
 | Pos_Number_Overflow of overflow_obj
```

After defining the data, the next step is to define the entire memory and pointer structures when writing class implementations (see Section 4.2). However, this example is still in the specification stage and it is good practice to leave the memory structure as general as possible until absolutely necessary, so the pointer structure definitions are postponed.

Next is the signature for the `operates` interface methods. The `identity` and `maximum` methods return single numbers. The `operate` method takes an object pointer, the state and two numbers (representing inputs), and the new state and a `pos_number_obj` (representing outputs) and returns true if all the arguments satisfy the specification:

```
type operates_iface_sig =
<|
  operate : 'operates_ptr
            -> ('state # num # num)
            -> ('state # pos_number_obj)
            -> bool;
  identity : 'operates_ptr -> num;
  maximum : 'operates_ptr -> num
|>
```

Note that the first parameter to each method need not be specified yet (as shown by the single quote (') mark in front of the type `'operates_ptr`); this way, multiple class objects can potentially implement this method and satisfy the obligations.

Semantically, this framework only cares that the `identity` and `operate` functions return a number lower than the component's `maximum` number. The `operate` function also guarantees that the result `z` is either a number under the maximum or it's an overflow[3]:

```
   operates_obligs (operates_class) =
   ∀ oper_ptr x y z state .
1)  ((operates_class.identity oper_ptr) ≤ (operates_class.maximum oper_ptr))
    ∧
2)  ((operates_class.operate oper_ptr (state, x, y) (state, z))
     ==>
3)   (XOR
4)    (∃ w . (z = (Pos_Number_Number (Number w)))
          ∧ (w <= (operates_class.maximum oper_ptr)))
5)    (∃ str . z = (Pos_Number_Overflow (Overflow str)))))
```

This specification for the `Operates` interface says in clause 1 that the value of the `identity` method is always less than the value of the `maximum` method. Clause 2 implies the rest of the formula; in other words, when the `operate` method is called, the result (`z`) obeys the constraints in clauses 3-5. Clause 4 says that the `PosNumber` result called `z` (the result of `operate` in clause 2) is a `Number` object with an internal value (`w`) less than the value of `maximum`. To the contrary, clause 5 says that it is an `Overflow` object containing some internal error message. So clauses 2-5 together state that after a call to `operate`, exactly one condition (clause 4) or the other (clause 5) is true, but not both.

[3]Note that numbers in HOL specifications are in no danger of overflow; they are defined logically as an infinite set.

3.2.1.2 Specify framework signatures and behavior

For our example, a class that `Calculates` can, in addition to applying a given operation once to a pair of numbers, repeatedly apply (or `fold`) the operation to a list of numbers. Following is the programmer's syntactic (Java) specification for a `Calculates` framework:

```
public interface Calculates {
  public Operates.PosNumber fold(Operates op, List yy);
}
```

The following is the syntactic declaration, where `fold` is a predicate over a `'calc_ptr`, three input parameters, and finally a `pos_number_obj`:

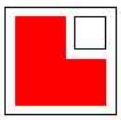
```
type calculates_iface_sig =
  <|
     fold : 'calculates_ptr
              -> ('state # 'operates_ptr # num list)
              -> ('state # pos_number_obj)
              -> bool
  |>
```

Following are the `Calculates` semantics; much like in the `Operates` component above, the result of `fold` is either an overflow value or a valid number not greater than the maximum integer value:

```
calculates_obligs calculates_class =
∃ operates_class::operates_obligs .
∀ calc_ptr oper_ptr state yy z .
(calculates_class.fold calc_ptr (state, oper_ptr, yy) (state, z))
==>
(XOR
 (∃ w . (z = Pos_Number_Number (Number w))
  ∧ (w <= operates_class.maximum oper_ptr))
 (∃ str . z = Pos_Number_Overflow (Overflow str)))
```

At this point we have introduced the key elements to framework verification: first were the component obligations, and finally the entire system obligations. Next comes the framework code in terms of the components, and its verification, but our most significant contribution is the technique for framework specification.

3.2.1.3 Implement framework with a predicate made of combinations of interfaces



The previous section shows how a specification is written in object-oriented terms for a framework; this section shows how the implementation is written using the approach in Chapter 4.

Just as in hardware circuitry modelled in logic, each method body becomes a predicate over the object data, machine state, input values, and output values where internal variable values are represented with existential variables. Also, since class implementations often depend on other library or component classes, we must supply the latter as arguments.

```
class_function (component_class) (object_ptr)
                (input_state, input1, input2) (output_state, output) =
∃ local_var1 local_var2 .
∃ inner_state1 inner_state2 component_ptr .
  primitive_method1 (input_state, input1)
  ∧
  component_class component_ptr (input_state, local_var1) (inner_state1)
  ...
```

Returning to the `Calculates` example, following is the implementation for a complete calculator:

```
public class Calculator implements Calculates {
  public Operates.PosNumber fold(Operates op, List yy) {
    if (yy.size() == 0) {
      return op.identity();
    } else {
      Operates.PosNumber rest = fold(op, yy.subList(1,yy.size()));
      if (rest instanceof Operates.Overflow) {
        return rest;
      } else {
        return op.operate(((Integer)yy.get(0)).intValue(),
                          ((Operates.Number)rest).value);
      }
    }
  }
}
```

The `Calculator` class contains no data so the object data definition is merely a constructor:

```
type calculator_obj = Calculator
```

The `Calculates` interface contains a single `fold` method. The implementation above translates to the following in HOL[4]:

```
(fold_function (operates_class) (calc_ptr:'calculates_ptr)
               ((state:'state), (oper_ptr:'operates_ptr), []) (state2, z) =
 ((state = state2)
  ∧
  (z = (Pos_Number_Number (Number (operates_class.identity oper_ptr))))))
∧
(fold_function (operates_class) calc_ptr
                (state, oper_ptr, CONS x xx) (state2, z) =
  (state = state2)
  ∧
  (∃ w .
   fold_function (operates_class) calc_ptr (state, oper_ptr, xx) (state, w)
   ∧
   (pos_number_obj_case
    (λ number .
     number_obj_case
     (λ num . operates_class.operate oper_ptr (state, x, num) (state, z))
     number)
    (λ message . Pos_Number_Overflow message = z)
    w)))
```

The idea is that the `fold_function` is a logic version of the Java code for the `fold` method in the `Calculates` class above. The details of that logic translation are not important for this discussion; that is the entire focus of Chapter 4.

Those are all the elements needed for the `Calculator` framework. When used in proofs or as a part of another system, it is referenced as a record in the following way;

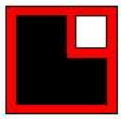
[4]The definition of `fold_function` must be manually proven to terminate with the list argument as shown in Appendix A.4.

note that it in turn references a component with the name `operates_class` since a `Calculates` component is a part of its implementation:

```
...  <| fold := fold_function operates_class |>  ...
```

The final system is an aggregate of these types of records, where the components used to implement them are often supplied when the system is deployed[5]. That is the essential technique of mapping framework implementations to HOL terms. Chapter 4 covers this topic in detail.

3.2.1.4 Verify framework behavior based on component specifications and programming logic



Verification is proof that our function implementations combined with the components satisfy the obligations.

Recall that `calculates_obligs` is a predicate over a class record definition; in this case, the class record contains a single `fold` function which is the `fold_function` above, coupled with an `Operates` component:

```
∀ operator::operates_obligs .
calculates_obligs <| fold := fold_function operator |>
```

There are three general steps to proving this and similar goals in this thesis:

- Rewrite with the specification and implementation definitions as much as possible. In other words, replace specification and component names with their

[5]It is possible to use the logical choice operator "ϵ" to name an implementation so that it stands alone as in the following example:

```
calculator_class =
<| fold := fold_function (ε operates_class::operates_obligs . T) |>
```

However, this mechanism is not useful in a larger system since the component ("`operates_class`") is not shared or connected with any other part of the system implementation.

meanings (such as `operates_obligs` in the logic of `fold_function` in the last section).

- Fill in the local variables (such as `j` in the last section) with values that help solve the goal.

- Use domain-specific facts to finish the proof. For our `Multiplier` example, this involves proof by arithmetic induction; for the EJB example in Section 5.3, this involves tactics which manipulate boolean conditions and implications.

The following is a synopsis of the proof in HOL. Appendix A.1 shows the HOL results at each of these steps.

1. Rewrite with the definitions from previous sections to get a goal in terms of `fold_function`:

2. Case-split on the variable `yy`, which is either an empty list or a list of at least one item:

   (a) For the case with the empty list, rewrite with the definitions of `fold_function`, `XOR`, and `operates_obligs`.

   Then rewrite with theorems about the relationships of the classes, such as that a `Number` does not equal an `Overflow`. At that point a value must be chosen for an instance of `w`, and since this is the case of an empty list, supply `operator_class.identity` to prove this case.

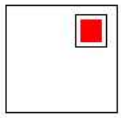
   (b) For the case with at least one item in the list, rewriting with the `fold_function` and then case splitting on its result yields the following two goals:

Each of these cases is solved by rewriting with the definitions and the properties of the `PosNumber` classes and then choosing the values for the existential variables that match existing variables.

3.2.2 Component Implementor

This section shows the four tasks of the framework deployer in more detail.

3.2.2.1 Implement component with a predicate



In general, component implementations are logically simpler than framework implementations since they are built from primitive programming constructs. Following is an `Checksum` component, built to implement the `Operates` interface[6]:

```
public class Checksum implements Operates {
  public int identity() {
    return 0;
  }
  public int maximum() {
    return 1000;
  }
  public Operates.PosNumber operate(int x, int y) {
    return new Operates.Number((x + y) % maximum());
  }
}
```

[6]We simplify this code by assuming that both `x` and `y` are positive arguments.

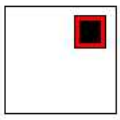
Those methods are translated to the following logic terms:

```
(checksum_identity checksum_ptr = 0)


(checksum_maximum checksum_ptr = 100)


(checksum_operate checksum_ptr (state, x, y) (state2, z) =
 (state = state2)
 ∧
 (z = Pos_Number_Number (Number ((x + y) MOD (checksum_maximum checksum_ptr)))))
```

The `Checksum` class does not rely on any other components, so the entire class can be defined independently:

```
checksum_class = <|
  identity := checksum_identity;
  maximum := checksum_maximum;
  operate := checksum_operate
|>
```

3.2.2.2 Verify component

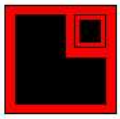


Following is the goal stating that the `checksum_class` satisfies the `Operates` specification;

```
⊢ operates_obligs checksum_class
```

The proof of this goal is a direct result of the properties of the "%" (modulo) operator.

3.2.2.3 Verify final system framework properties



Since the `Calculator` class framework was verified for any component that satisfies the `operates_obligs` obligations (Section 3.2.1.4), and since the `checksum_class`

was proven to satisfy those obligations (Section 3.2.2.2), the combination of the two correctly implements the `Calculates` functionality.

In logical terms, this final theorem is a logical consequence of those two previous theorems:

```
⊢ calculates_obligs <| fold := fold_function checksum_class |>
```

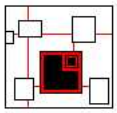
In fact, the following HOL script `VERIFY_FRAMEWORK` manipulates the framework and component correctness theorems and generates that final theorem automatically:

```
1)  fun VERIFY_FRAMEWORK framework_thm component_thm =
2)  MP
3)  (PART_MATCH
4)   (fst o dest_imp)
5)   (REWRITE_RULE [RES_QUAN_CONV (concl framework_thm)] framework_thm)
6)   (concl component_thm))
7)  component_thm
```

That script works as follows. Line 5 converts the framework theorem into an implication saying that the correctness of a component implies framework correctness. Lines 3-6 then match the actual component theorem results; the result of those lines says that the component already proven correct implies framework correctness. That result, combined with the component correctness theorem (line 7) using modus ponens (line 2), yields the framework correctness theorem.

This step is fully automatic.

3.2.2.4 Verify final system client properties



Finally, now that the client has verified their own incarnation of the framework system they can more easily verify the properties in which they are ultimately interested. This is not an automatic verification since these properties are in an arbitrary

form, but it is a very simple proof because it is a direct result of the framework properties which is the reason the framework was employed in the first place.

For example, the calculator with the checksum component might be part of a data transfer program that ensures the checksum value occupy no more than a byte in memory.

```
type transfer_sig =
  <|
     transfer : 'transfer_ptr
                -> ('state # 'calculator_ptr # num list)
                -> ('state # num)
                -> bool
  |>



transfer_obligs transfer_class =
∀ transfer_ptr state amounts checksum .
  transfer_class.transfer transfer_ptr (state, amounts) (state, checksum)
  ==> checksum < 256
```

The `Transfer` class extends the `Checksum` class with a `transfer` method that returns the computed checksum as a `char` value.

```
public class Transfer extends Checksum {
  public char transfer(List[] amounts) {
    Operates.PosNumber result = Calculator.fold(this, amounts);
    if (result instanceof Operates.Number) {
      return (char)((Operates.Number)result).getValue();
    } else {
      return 0;
    }
  }
}
```

That is translated to the following logic term:

```
transfer_transfer (calculator_class) transfer_ptr
                  (state1, calculator_ptr, amounts) (state2, checksum) =
∃ result .
calculator_class.fold calculator_ptr (state1, transfer_ptr, amounts)
                                        (state2, result)
∧
(checksum =
 (pos_number_obj_case
  (λ number . number_obj_case (λ num . num) number)
  (λ message . 0)
  result))
```

Is it straightforward to specify and implement the final framework since the initial framework abstractions clearly state the deployment requirements. It is also straightforward to prove the final properties because they follow directly from the framework properties proven automatically in the previous step. The goal is stated as follows:

```
transfer_obligs
<|
  transfer := transfer_transfer <| fold := fold_function checksum_class |>
|>
```

This chapter detailed each step of framework verification. Chapter 5 gives further examples of this process. However, there are specific technical issues that arise when translating an implementation into logic terms, and these are explained in Chapter 4.

Chapter 4

Translating Implementations to Logic

The sections in this chapter show how programming concepts can be translated into logic terms. The focus is on object-oriented concepts in imperative languages and how they map to logic terms in a classical higher-order logic. This is background for the examples in Chapter 5, where each implementation is generated as shown by this chapter.

The programming language we have chosen is Java and the formal logic is the Higher-Order Logic (HOL) theorem-proving system. While all the examples are done in these specific tools, the concepts are not tied to them. The approach and all the translation and proof details are applicable to object-oriented programming languages and higher-order logics in general.

Similarly, although most examples have been hand-crafted following the principles in this chapter, this process could be automated. The formulae in Section 4.2 were created with the help of automatic tools and the rest of the examples could have been generated much the same way.

Almost all the work in this section is original; exceptions are acknowledged. For other formal representations of programming languages in general and Java in particular, see the references in Section 2.3.

Section 4.1 introduces the fundamental programming concepts in the context of a simple verification. Section 4.2 goes into more detail about basic programming data types and expressions. Section 4.3 addresses remaining issues such as well-formedness of object data.

4.1 Basic Software Verification

This section is an introduction to the three verification steps of specification (4.1.1), implementation (4.1.2), and proof (4.1.3) for object-oriented programs. This section does not illustrate a framework program, but it does show how basic object-oriented constructs map to prediates and proofs.

4.1.1 Specify properties

First we show how an interface maps to a HOL predicate. Our first example is a function that multiplies two numbers, with the following interface in Java:

```
public interface Multiplies {
  public int operate(int m, int n);
}
```

That signature translates to a new HOL record type containing one element: `operate`, which is a function from an object reference and two numbers to a third number. In this work, each method is transformed into a predicate, which is a final type that ranges from the inputs and output to a boolean value[1] [2]:

[1] It is not absolutely necessary to convert every method to be a predicate since each interface- and class-level construct is also a predicate. However, we use this convention in order to conform to the standard way of defining functions in theorem-provers.

[2] More explanation of the object-reference parameter can be found in Section 3.2.1.1.

```
type multiplies_sig = <| operate: 'mult_ptr -> (num # num) -> num -> bool |>
```

The following is a HOL specification for the interface `Multiplies`:

```
multiplies_obligs multiplier =

  ∀ mult_ptr m n k . multiplier.operate mult_ptr (m,n) k = (k = m*n)
```

That says that a class with a function `operate` satisfies the `multiplies_obligs` property if and only if, for every call to `operate` which returns `k` for inputs of `m` and `n`, `k` has a value of `m*n`[3].

4.1.2 Implement and translate into logic terms

4.1.2.1 Implement system

The following is an implementation of `multiplies_obligs` where `operate` does its computation with a recursive call.

```
public class Multiplier implements Multiplies {
  public int operate(int m, int n) {
    if (n==0) {
      return 0;
    } else {
      int i = n-1;
      int j = operate(m, i);
      return m + j;
    }
  }
}
```

[3]This first example has no elements of a framework since this section is only for introductory purposes.

In HOL terms, our code looks like this:

```
1)  mult_operate mult_ptr (m, n) k =
2)    if (n = 0)
3)    then k = 0
4)    else let i = n-1 in
5)         ∧ (∃ j . mult_operate mult_ptr (m, i) j
6)                ∧ (k = m + j))


7)  multiplier_class = <| operate := mult_operate |>
```

Line 1 declares a function called `mult_operate` which takes three parameters corresponding to values from the programming code (ie. inputs `m` and `n` and output `k`). Since this method is a predicate, its result will be true or false based on whether the parameter values satisfy the relationship of the respective parameters in the programming code. Line 2 starts the conditional and line 3 is the first arm, which says that the output must be 0. Lines 4-6 make up the else clause, where `i` becomes the value of `n-1`, `j` becomes the output value of the recursive call to `mult_operate`, and the final output is the value of `m` added to the result of the recursive call.

Line 7 defines `multiplier` as a record with one element: an `operate` method which is the `mult_operate` function just described.

4.1.2.2 Implement with components

This section is a small diversion into how a program might be organized with components. Specifically, we modify the `Multiplier` class such that it is implemented with components.

A framework system is implemented in terms of partially specified (ie. abstract or interface) classes. Consider how such a class might fit into our `Multiplier` example; if the addition operation were made more general, it might be fashioned into an

interface and used in place of the + operator. The following is a sample interface for such a component:

```
public interface Adds {
  public int add(int m, int n);
}
```

The translation of this code into logic terms is precisely like that above. We start with the syntactic declaration of the Adds interface and then intended behavior:

```
type adds_sig = <| add: 'add_ptr -> (num # num) -> num -> bool |>


adds_obligs adder =
  ∀ add_ptr m n k . adder.add add_ptr (x,y) z = (z = x+y)
```

The syntax of the new Multiplies2 interface must be modified to accept the new component. There are better ways to do this, such as to have a set method for the component or to let each implementing class take the Adds component in their constructor; however, we do it as simply as possible for this first example.

```
public interface Multiplies2 {
  public int operate(Adds a, int m, int n);
}
```

Its logic syntax and behavior are also altered: the operate method is now implemented with a component that is supplied at execution time. This takes the form of a new add_class and add_ptr argument to the logic version of the operate method:

```
type multiplies2_sig = <|
  operate: adds_sig -> 'mult_ptr -> ('add_ptr # num # num) -> num -> bool
|>


multiplies2_obligs multiplier2 =
  ∀ add_class mult_ptr add_ptr m n k .
  multiplier2.operate add_class mult_ptr (add_ptr,m,n) k = (k = m*n)
```

The new `Multiplier2` implementation makes use of the `Adds` component:

```
public class Multiplier2 implements Multiplies2 {
  public int operate(Adds a, int m, int n) {
    if (n==0) {
      return 0;
    } else {
      int i = n-1;
      int j = operate(m, i);
      return a.add(m, j);
    }
  }
}
```

Besides the new `Adds` interface, the only change is the use of `a.add(m,j)` instead of `m+j`.

Finally, we translate the new `Multiplier2` code:

```
1) mult_operate2 add_class mult_ptr (a, m, n) k =
2)   if (n = 0)
3)   then k = 0
4)   else let i = n-1 in
5)        ∧ (∃ j . mult_operate2 mult_ptr (m, i) j
6)               ∧ (add_class.add a (m, j) k))


7) multiplier2 = <| operate := mult_operate2 adder_impl |>
```

That involves three changes to the code from the first example:

- Line number `2` now includes the two new arguments for the class definition `add_class` and the object argument `a`.
- Line number `6` now uses the interface method rather than the hard-coded `+` operation.
- Line number `7` now includes the term `adder_impl`; this is the adder class that the user wishes to use inside the `multiplier2` class.

With those changes the `Multiplier2` class is more general since it can be run with any component fulfilling the `Adds` interface.

4.1.3 Verify that implementation satisfies specification

Finally we verify the code by proving that the `multiplier_class` satisfies the `multiplies_obligs`. The primary conjecture is stated as follows:

```
⊢ multiplies_obligs <| operate := mult_operate |>
```

The other version of this goal that uses the more component-oriented example is as follows:

```
⊢ ∀ adder :: adder_obligs
  multiplies2_obligs <| operate := mult_operate2 adder |>
```

Note that the goal is to show that any `adder` component that meets the `adder_obligs`; once that is verified, any adder meeting those criteria can be substituted in the place of `adder`, and the resulting proof is simple as explained in Section 3.2.2.3.

Proving the primary goal is straightforward but requires intelligent selection of the proof sequence. Following are the proof steps to make that goal a theorem in HOL:

1. Rewrite with the definitions from Section 4.1.2 to get a goal showing the specific behavior expected from `mult_operate`.

2. Induct on the variable `n` to get a base case and induction step.

3. The base case is proven with the following steps:

   (a) Rewrite once with the definition of `mult_operate`.

   (b) Induct on `k` for another base case and induction step. Each of these are easily provable based on the properties of zero (with the HOL tactic `reduceLib.REDUCE_TAC`).

4. The induction step for `n` is proven with the following steps:

   (a) Rewrite once with the definition of `mult_operate`.

   (b) Simplify the phrase (`SUC n - 1`) to `n`, expand the `let` phrase by substituting `n` for `i` in the remainder of the goal, and eliminate the first arm of the `if` clause since (`SUC n`) is never 0.

   (c) Rewrite with the hypothesis for the induction step to get the following.

   (d) The remainder depends on properties of multiplication and division. Note that by instantiating (`m * n`) for `j`, the result is the following straightforward goal:

   ```
   (k = m + m * n) = (k = m * SUC n)
   ```

Appendix A.2 shows the HOL results at each of these steps. The entire HOL proof script to solve that theorem is found in Appendix A.3.

4.2 Handling State Naively

The most difficult verification issue is handling persistent state. This is mainly because the state variable is mutable, and also because function calls affecting one record may affect the data in other records accessed through object references. Another reason is that the state represents many data values, so elements of concern are accessed and updated indirectly which complicates proofs as well as specifications. These difficulties would be especially pronounced for raw pointers to memory addresses but they remain problematic for object references.

In this section, we show one possible encoding for object pointers and other persistent data and how it can deal with issues of static (class) variables and inheritance. This particular formulation is introductory but not comprehensive; we address its shortcomings in the next section.

State can be a tuple with a place for the list of objects of each class. Objects are then simple aggregates of object data. Object data may be primitive data (currently HOL booleans, numbers, or strings) or object references. Each class has two unique object reference constructors: one for the null value and one for object references. Each object reference is an offset into the state list for the associated class.

One significant limitation with this approach is that the classes (not interfaces) must be known at the time of encoding. A consequence is that any change to a class or even the addition of a new class requires that the entire program encoding be regenerated. We reiterate that this is consistent with the shallow nature of our logic embedding; we gain more understandable terms and HOL type-checking that enforces many constraints, and we lose general constructs and the power to prove general properties about statements and expressions.

4.2.1 Data Model

Memory can be represented with an array of the live objects. While one single array would be simple, it's possible to take advantage of HOL's typechecking with a little more work: each class will have its own list of objects of that type. So the entire state becomes a tuple of all types. The following is a state model for the calculator from the last section; first is the definition of the data object types, then the entire state type, then the object reference types:

```
type number_obj = Number of num;
type overflow_obj = Overflow of string;


type state = State of number_obj list # overflow_obj list


type number_ptr = Number_Ptr of num | Number_Null
type overflow_ptr = Overflow_Ptr of num | Overflow_Null
```

Accessor functions are simply array offsets:

```
number_lookup (Number_Ptr x) (State (numbers, overflows)) = EL x numbers


overflow_lookup (Overflow_Ptr x) (State (numbers, overflows)) = EL x overflows
```

The code "`EL x numbers`" returns the element of array `numbers` at index `x`.

Static variables and constants are not shown in this example nor in most of the following discussion. They can be handled at the global state level with their own separate entries from the lists of object data, and they would have their own access and assignment methods much like those for object data.

As shown in the previous examples, there are no data constructors defined for interfaces; the functions are written to take generic types (denoted by a prepended apostrophe, such as `'a`). This is so that any data type can be used to fulfill that interface. The class methods, when defined, will supply the appropriate type.

The class relationships for abstract and super-classes are straightforward:

```
type pos_number_obj =
   Pos_Number_Number of number_obj
 | Pos_Number_Overflow of overflow_obj


type operator_obj = Operator
```

Wherever a sub-class is used as a particular instance of a super-class it is wrapped with a logic constructor such as `Pos_Number_Number`; for any downward cast the sub-class is simply unwrapped.

Note that there is a logic constructor for each implementing class; the `pos_number_obj` interface has two implementing classes so there is one for each, but the `operator_obj` has no subclasses so there is a single logic constructor needed to make the type declaration even though it will never be used in a term. The method behavior takes more planning and will be discussed later. (To preview, the process is to create stub method definitions which invoke the actual methods up or down the hierarchy. Class casts works much the same way.)

4.2.2 Constructors

Creating a default new object means adding an element to the state with all fields set to default values. The return value of constructors is a reference to the new object, along with the new state[4]:

[4]While we use predicates over inputs and outputs in this thesis, in most places we could use functional types instead. For example, `new_number` above could take a state and return a pair of the new state and the new object reference. This would make many of the expressions more closely resemble programming statements and thus easier to understand. However, to remain consistent throughout this thesis we will continue to use predicates.

```
new_number (State (numbers, overflows)) (new_state, new_ptr) =
  let new_obj = Number 0
  in (new_state = State (APPEND numbers [new_obj], overflows)
     ∧
     (new_ptr = Number_Ptr (LENGTH numbers))

new_overflow (State (numbers, overflows)) (new_state, new_ptr) =
  let new_obj = Overflow ""
  in (new_state = State (numbers, APPEND overflows [new_obj])
     ∧
     (new_ptr = Overflow_Ptr (LENGTH overflows))
```

We do not discuss ways to reclaim memory or improve efficiency in this thesis.

Explicitly declared class constructors look similar to these above except that they would contain more arguments and possibly run code to affect the given state. For example, the following is a constructor declaration for the `calculator_class` from the last section that might store an `Operates` argument:

```
  public Calculator(Operates op) {.....}
```

The following is the HOL version, assuming there exist objects such as `adders` that fulfill the `Operates` interface:

```
calculator (State (calculators...), operator) (new_state, new_calc_ptr) =
∃ inner_state .
  let next_state = State (APPEND calculators [Calculator Operator_Null]...)
  in let inner_state = .....
     in (new_state = inner_state)
        ∧
        (new_calc_ptr = Calculator_Ptr (LENGTH calculators))
```

4.2.3 Fields

Field access and assignment for objects requires special processing since functional logics do not allow assignment. Getting field values is straightforward; setting field values returns a new state:

```
overflow_get_message_obj (Overflow message) result = (result = message)

overflow_get_message (Overflow_Ptr offset) (State (numbers, overflows)) result =
  overflow_get_message_obj (EL offset overflows) result

overflow_set_message_obj (Overflow message) new_message overflow_result =
  (overflow_result = (Overflow new_message))

overflow_set_message (Overflow_Ptr offset)
                     (State (numbers, overflows), new_message)
                     state_result =
  ∃ new_overflow .
  overflow_set_message_obj (EL offset overflows) new_message new_overflow
  ∧
  (state_result =
    State (numbers,
           (APPEND (FIRSTN offset overflows)
                   (CONS new_overflow (BUTFIRSTN (offset+1) overflows)))))
```

4.2.4 Expressions and Statements

Expressions and statements are fairly straightforward to model. Expressions without side effects look very similar; For example, integer operations are exactly the same. Most other primitives (including reals and strings) and their operators have some kind of analogy in a mature logic.

Expressions with side effects may affect a local variable or the entire state. In the former case a variable is updated, and in the latter the whole state must be updated.

These are all placed in conjuncts where the variables are assigned to new values, and any externally visible values are declared equivalent at the end.

The following is an example snippet of Java which both updates a local variable and creates a new state object:

```
... new Integer(a++)...
```

The HOL version looks like the following:

```
...
∧ (value, a') = (a, a+1)
∧ (new_integer (state, value) (state', obj_ptr))
...
∧ (output_a = a')
```

As for operator overloading, logics such as HOL do not allow ad hoc operator overloading, but there is always a mapping to a non-overloaded version. Since most object-oriented languages have strong typing, each expression type can be determined such that the right operators may be chosen or the subexpressions may be explicitly cast to the right types.

Statements are also usually straightforward to model in HOL. We list here the elements that are the most difficult (but we do not address these issues any further in the rest of this thesis):

- Loops, while simple to specify, are usually very difficult to verify for termination properties. The verifier must ensure that the conditions placed on each variable in preceding code are enough to prove termination. This is a requirement for defining the code used in an implementation, which happens even before a program can be proven correct.

For the sake of completeness we show a definition for the general case of a `for` loop. This version is only the inner loop; it takes a continuation `condition`, an `iter_step` statement to run after each loop, and a `body` to execute, but no initial statement to run. The general definition (`for_pred_aux`) cannot be made in HOL; a specific condition, iterative step, and body and possibly even restrictions on the state would have to be written for each instance in order to define this type of loop.

```
for_pred_aux (state, condition, iter_step, body) (state') =
(∃ (inner_state) (inner_state') inner_state'' continue .
 condition state (inner_state, continue)
 ∧
 (continue =>
     (body inner_state inner_state'
      ∧
      iter_step inner_state' inner_state''
      ∧
      for_pred_aux (inner_state'', condition, iter_step, body) state')
     |
     (state' = state)))
```

- Exceptions force a jump past all subsequent normal statements and outside functions until execution hits an exception-handling statement. A good way to handle this (as in [Lei95]) is with one extra element in the state which tells whether an exception has been thrown; then each statement conditionally executes based upon the exception status (including exception-handling statements). (This could usually be done in HOL with a higher-order function statement-handling function that would conditionally execute most statements based on the state.)

- Return statements do not do much: that is the point in execution at which the output values must reconcile with the internally computed values. (The special case is when a `return` executes conditionally in a block statement which is followed by more statements; see the next paragraph.)

- Other control-flow statements such as `break` or `continue` (or even a `return` that does not come last in a method) do much the same thing as exceptions. A general statement-reorganization scheme would be very difficult; one alternative approach would be to put another element in the global state which is set or cleared based on the statement's placement in the enclosing block.

4.2.5 Example involving state

The rest of this section steps through a specification and proof involving state. The example framework is another generic calculator, but this one remembers the previous result and can undo the previous operation.

4.2.5.1 Specify framework

The calculator will have one operation assigned to it and also contains the most recently calculated result. It is constructed with an element that `Inverts` and an initial value, and it has `operate` and `undo` methods. The signature for the `UndoableCalc` class is as follows; note how it interacts with an `Inverts` component (Figure 4.1):

```
public interface UndoableCalc {
  public UndoableCalc(Inverts op);
  public void set(int y);
  public int operate(int y);
  public int undo(int z);
}
```

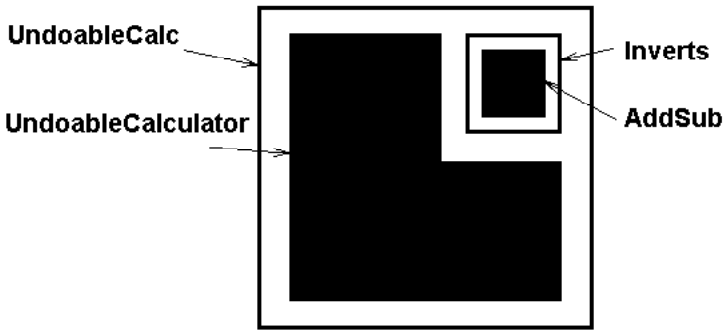


Figure 4.1: "Undoable" framework-component diagram

This framework will only store one datum in the object, so the `undo` function relys on the behavior of the `invertor`. Without that invert property, the framework would have to store at least two datum (for the last value and for the one before that). This is a contrived but valid example framework that exhibits state-dependant behavior.

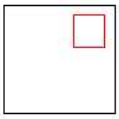
First we define the syntax of the operations:

```
type undoable_iface_sig =
<|
  set : 'undoable_ptr -> ('state # num) -> 'state -> bool;
  operate : 'undoable_ptr -> ('state # num) -> ('state # num) -> bool;
  undo : 'undoable_ptr -> ('state # num) -> ('state # num) -> bool
|>
```

Following is the behavioral specification for that interface:

```
undoable_obligs undoable_class =
∀ undo_ptr state state' state'' state''' x x' y z.
(undoable_class.set undo_ptr (state, x) (state'))
∧
(undoable_class.operate undo_ptr (state', y) (state'', z))
∧
(undoable_class.undo undo_ptr (state'', z) (state''', x'))
==>
(x' = x)
```

4.2.5.2 Specify components



The calculator relies on components that satisfy the `Inverts` interface which is defined as follows:

```
public interface Inverts {
  public int operate(int x, int y);
  public int invert_op(int z, int y);
}
```

The following is that syntax in logic terms:

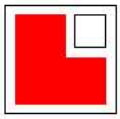
```
type inverts_iface_sig =
<|
  operate : 'inverts_ptr -> 'state # num # num -> num -> bool;
  invert_op : 'inverts_ptr -> 'state # num # num -> num -> bool
|>
```

When given the result and the second value from any previous call to `operate`, `invert_op` returns something equivalent to the first value from that call[5] .

```
inverts_obligs inverts_class =
∀ inv_ptr x y z x' state state' .
 (inverts_class.operate inv_ptr (state, x, y) z)
 ∧
 (inverts_class.invert_op inv_ptr (state', z, y) x')
 ==>
 (x' = x)
```

[5]Note that the interface functions are defined in terms of the generic types `'inverts_ptr` and `'state`. This makes it possible to define the interfaces before the program-specific classes, so the interfaces can be shared independant of the shallowly-embedded programs that use them.

4.2.5.3 Implement framework



The following is an implementation of `UndoableCalc`, assuming there exists a `AddSub` class that implements `Inverts`:

```
public class UndoableCalculator {

  private int prev;
  private Inverts op;

  public UndoableCalculator(Inverts op, int initial_value) {
    this.op = op;
    this.prev = initial_value;
  }

  public void set(int x) {
    prev = x;
  }

  public int operate(int y) {
    prev = op.operate(prev, y);
    return prev;
  }

  public int undo(int y) {
    prev = op.invert_op(prev, y);
    return prev;
  }

  public static void main(String[] args) {
    UndoableCalculator uncalc_ptr=new UndoableCalculator(new AddSub(),5);
    System.out.print("First result: ");
    System.out.println(uncalc_ptr.operate(4));
    System.out.print("Prev value: ");
    System.out.println(uncalc_ptr.undo(4));
  }

}
```

We now show the HOL version of that code with the complete state definitions. Following are the auxiliary type declarations:

```
type uncalc_obj = Uncalc of 'inverts_ptr # num



type state = State of 'inverts_ptr uncalc_obj list



type system_out_sig =
<| println: ('inverts_ptr state # 'outstream) -> ('inverts_ptr state) -> bool |>



type uncalc_ptr = Uncalc_Ptr of num | Uncalc_Null



type uncalc_class_sig =
<|
  uncalc : ('inverts_ptr state # 'inverts_ptr # num) -> 'inverts_ptr state
           -> bool;
  set : uncalc_ptr -> ('inverts_ptr state # num) -> 'inverts_ptr state
        -> bool;
  operate : uncalc_ptr -> ('inverts_ptr state # num)->('inverts_ptr state # num)
            -> bool;
  undo : uncalc_ptr -> ('inverts_ptr state # num) -> ('inverts_ptr state # num)
         -> bool;
  main : ('inverts_ptr state # string list) -> 'inverts_ptr state
         -> bool
|>
```

Following are the object constructor, getter, and setter methods:

```
uncalc_new (State uncalcs, op, initial_value)
           (new_state, new_ptr) =
let result_obj = Uncalc (op, initial_value)
in (new_state = State (APPEND uncalcs [result_obj])
   ∧
   (new_ptr = Uncalc_Ptr (LENGTH uncalcs))



uncalc_get_op_obj (Uncalc (op, prev)) result =
  (result = op)
uncalc_get_op (Uncalc_Ptr offset) (State uncalcs) result =
  uncalc_get_op_obj (EL offset uncalcs) result



uncalc_get_prev_obj (Uncalc (op, prev)) result =
  (result = prev)
uncalc_get_prev (Uncalc_Ptr offset) (State uncalcs) result =
  uncalc_get_prev_obj (EL offset uncalcs) result



uncalc_set_op_obj (Uncalc (op, prev)) new_op uncalc_result =
  (uncalc_result = Uncalc (new_op, prev))
uncalc_set_op (Uncalc_Ptr offset) (State uncalcs, new_op)
              state_result =
  ∃ new_uncalc .
  uncalc_set_op_obj (EL offset uncalcs) new_op new_uncalc
  ∧
  (state_result =
    State (APPEND (FIRSTN offset uncalcs)
                  (CONS new_uncalc (BUTFIRSTN (offset+1) uncalcs))))
```

```
uncalc_set_prev_obj (Uncalc (op, prev)) new_prev uncalc_result =
  (uncalc_result = Uncalc (op, new_prev))
uncalc_set_prev (Uncalc_Ptr offset) (State uncalcs, new_prev)
                state_result =
  ∃ new_uncalc .
  uncalc_set_prev_obj (EL offset uncalcs) new_prev new_uncalc
  ∧
  (state_result =
    State (APPEND (FIRSTN offset uncalcs)
                  (CONS new_uncalc (BUTFIRSTN (offset+1) uncalcs))))
```

Finally, following are the function definitions for the `Uncalc` class.

```
uncalc_undoablc_calc = uncalc_new


uncalc_set = uncalc_set_prev


uncalc_operate uncalc_ptr (state, y) (state_result, num_result) =
∃ op::inverts_obligs .
∃ inner_prev inner_op inner_result inner_state .
  uncalc_get_prev uncalc_ptr (state) (inner_prev)
  ∧
  uncalc_get_op uncalc_ptr (state) (inner_op)
  ∧
  op.operate inner_op (state, inner_prev, y) (inner_result)
  ∧
  uncalc_set_prev uncalc_ptr (state, inner_result) (inner_state)
  ∧
  uncalc_get_prev uncalc_ptr (inner_state) (num_result)
```

```
uncalc_undo uncalc_ptr (state, y) (state_result, num_result) =
∃ op::inverts_obligs .
∃ inner_prev inner_op inner_result inner_state .
  uncalc_get_prev uncalc_ptr (state) (inner_prev)
  ∧
  uncalc_get_op uncalc_ptr (state) (inner_op)
  ∧
  op.invert_op inner_op (state, inner_prev, y) (inner_result)
  ∧
  uncalc_set_prev uncalc_ptr (state, inner_result) (inner_state)
  ∧
  uncalc_get_prev uncalc_ptr (inner_state) (num_result)
```

Note the omission of the `main` method; it is not needed to prove the `Undoable` obligations but it is listed below for the interested reader.

Note also that `main` requires the `AddSub` class definitions since an instance must be constructed. Unfortunately, constructors are often not included in interface definitions; for example, Java does not allow any constructors in interfaces. This is probably a good decision because an interface should not specify how an object should be created; if object creation is an important function of the interface then a `getInstance` method or a class factory should be specified.

Regardless, the example below includes the `add_sub` constructor in the type definition and shows how to relate the interface obligations for the `AddSub` class (named `add_sub_obligs`) to other obligations (such as `inverts_obligs` below; others could be included as well). (The addition of a new class mandates modifications to the above `state` and the addition of getter, setter, and constructor methods.)

```
type add_sub_class_sig =
<|
  add_sub : state -> state # add_sub_ptr;
  operate : add_sub_ptr -> state # num # num -> num -> bool;
  invert_op : add_sub_ptr -> state # num # num -> num -> bool
|>



add_sub_obligs (add_sub_class:add_sub_class_sig) =
  inverts_obligs (y where operate = add_sub_class.operate
                    where invert_op = add_sub_class.invert_op)
```

The last method is `main`:

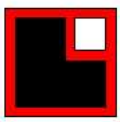
```
uncalc_main (state, args) (state_result) =
∃ System_out:system_out_sig outstream_ptr .
∃ (add_sub::add_sub_obligs) add_sub_ptr.
∃ uncalc_ptr
  inner_result inner_result'
  inner_state inner_state' inner_state'2 inner_state'3
  inner_state'4 inner_state'5 inner_state'6 inner_state'7
  .
  add_sub.add_sub (state) (inner_state, inner_add_sub_ptr)
  ∧
  uncalc_new (inner_state, inner_add_sub_ptr, 5) (inner_state', uncalc_ptr)
  ∧
  System_out.println outstream_ptr (inner_state', "First result: ")
                                   (inner_state'2)
  ∧ ...
```

```
  uncalc_operate uncalc_ptr (inner_state'2, 4)
                             (inner_state'3, inner_result)
  ∧
  System_out.println outstream_ptr (inner_state'3, num_str(inner_result))
                                   (inner_state'4)
  ∧
  System_out.println outstream_ptr (inner_state'4, "Prev value: ")
                                   (inner_state'5)
  ∧
  uncalc_operate uncalc_ptr (inner_state'5, 4)
                             (inner_state'6, inner_result')
  ∧
  System_out.println outstream_ptr (inner_state'6, num_str(inner_result'))
                                   (inner_state'7)
  ∧
  (state_result = inner_state'7))
```

Finally, the `undoable` framework class is defined in terms of those methods:

```
undoable_calc_class =
<|
  undoable_calc = uncalc_undoable_calc;
  set = uncalc_set;
  operate = uncalc_operate;
  undo = uncalc_undo;
  main = uncalc_main
|>
```

4.2.5.4 Verify framework

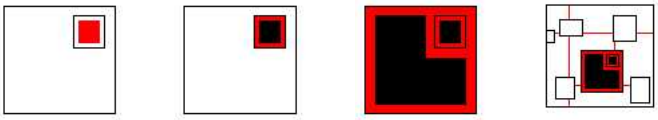


The following is the verification goal, declaring that the `UndoableCalculator` class satisfies the undoable interface obligations `undoable_obligs`:

```
⊢ undoable_obligs undoable_calc_class
```

The proof of this goal follows the same structure as that in Section 3.2.1.4.

4.2.5.5 Components and verification



Since our emphasis is on the framework itself, we do not develop the last four stages of component implementation, component proof, overall verification result, and final system deployment.

4.3 Handling State Correctly

The preceding section showed a trivial formalism and proof of machine state involving a single component. However, it did not address the problems of showing what object data remains unchanged after method calls nor guaranteeing class invariants for all object data. This same problem occurs when verifying that a particular data model maps to a more abstrct data model.

Since this thesis primarily addresses the composition of components into verified frameworks, the complete task of mapping between state formalisms and proving data invariants is left to future work. However, this section discusses the issues involved and their analogies in hardware verification, then shows how a "well-formed" predicate can be used for interfaces to map between the concrete and abstract state models.

Restriced access (or rather "well-defined interfaces") is the means to tractable verification involving state and component hierarchies. Here's how: the state model at each level of abstraction models data appropriately for that level; the mapping to a higher level of abstraction is done method by method, such that an entire method implementation at the more concrete level corresponds to a single atomic action at the more abstract level.

To show how this would work, we explain how state abstractions are handled in the verified microprocessor Uinta [Win94] and then how a similar approach would

work for this work.

Uinta was based on the concept of a "generic interpreter" which sequentially executes a stream of instructions which change state and generate output. This was used to verify the correctness of a macro-instruction machine built from simple components with two abstraction levels in-between; in other words, the basic components were verified to implement a micro-instruction machine, and that was verifed to implement another more abstract machine, and that was verified to implement the final machine.

The key innovation that applies here is the abstraction method: each level defines more powerful operations than the lower one by combining multiple steps into one, ie. by temporal abstraction. The following is part of the conclusion of the formal statement of correctness for an instantiation, where a lower-level instruction interpreter is mapped into a higher-level one:

```
let f t = sync gi (s' t) (e' t)
in ((select gi (s t) (e t) = k) ∧ (f t)
    ==> ∃ c . Next f (t,t+c)
               ∧ (instructions gi k (s t) (e t) = (s (t + c))))
```

Although the background and details of that formula are not given, it illustrates how the more abstract function `f` is verified at time `t` and then skips forward some number of lower-level clock cycles using a `Next` function to be verified at the next applicable time `t+c`. It is not verified at every juncture; it is only verified at the points in which it should synchronize with the lower implementation level. (The synchronization function in the above formula is `sync`.) This is a good way to map from one state model to a more abstract model.

Software formalisms are a bit different and potentially more complicated; although the memory references and values are still stored in a global state, they are created (and deleted) at will. However, they are also simpler in some ways; while the micro-

processor clock is involved in every state change, program states can be completely hidden at the abstract level and there is no need to map the sequence of state changes. However, mapping from the lower-level data to that in the more abstract state is still a necessary step.

In order to relate the state of memory at any two points in time we write a "well-formed" predicate that specifies the behavior expected from each particular object. This predicate is like our other behavioral specifications up to this point since it is defined in terms of the method results. The difference is that it only applies at one particular point in time; the rest of the specification is written in terms of this "well-formed" definition.

In other words, the theory obligations (ie. class invariants) in this section are broken into two parts: a "well-formed" predicate (named "impl_for_..._ptr") for the requirements for a particular object reference in a state, and the obligations (named "..._obligs") which say that every object is "well-formed" in every state. As a result, the state can be verified between successive method calls until the entire, more abstract methods are verified.

The remainder of this section gives an example formal specification and verification involving state and illustrates what special definitions and abstractions might be used in a larger system. Again, being focused on framework verification, most details of state formalisms, mappings, and verification are left to future work.

4.3.1 Specify Framework

This framework example is based on the role of a consumer. The `consumes` interfaces includes two simple functions: `eat` takes a single number representing an arbitrary token and `eaten` returns a count of how many tokens have been consumed. The `balancer` framework is built as a wrapper around two consumers such that it

gives each of them an equal number of tokens. In other words, it implements the `balances` interface with two functions: one accessor method `retrieve` for the consumers and the method `delegate` which takes a token and passes it to one of those consumers.

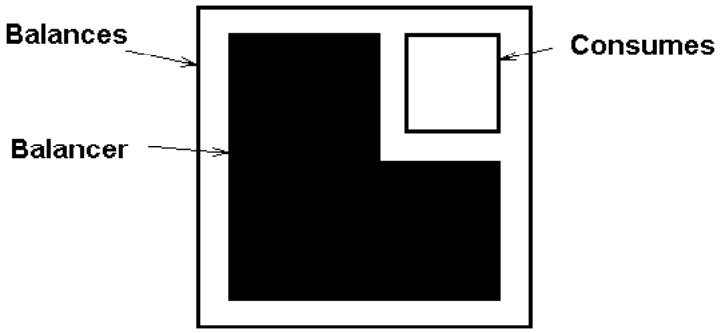


Figure 4.2: "Balances" framework-component diagram

Any `balances` framework has the following two methods:

```
type balances_iface_sig =
<|
  delegate : 'balances_ptr -> ('state # num) -> ('state) -> bool;
  retrieve : 'balances_ptr -> ('state)
              -> ('state # 'consumes_ptr # 'consumes_ptr) -> bool
|>
```

The `balances` specification says that the two consumers accessed from a balancer will have an `eaten` count that is no more than one different from each other:

```
impl_for_bal_ptr bal_class bal_ptr state =
∀ (consumes_class:('con_ptr, 'state) consumes_iface_sig)::consumes_obligs .
∀ con1_ptr con2_ptr state' .
∀ count1 count2 state'' state''' .
(bal_class.retrieve bal_ptr (state) (state', con1_ptr, con2_ptr))
∧
(consumes_class.eaten con1_ptr (state') (state'', count1))
∧
(consumes_class.eaten con2_ptr (state') (state''', count2))
==>
(state' = state)
∧
(state'' = state')
∧
(state''' = state')
∧
((count1 = count2)
 ∨
 (count1 = count2+1)
 ∨
 (count1+1 = count2))



balances_obligs balances_class =
 ∀ bal_ptr state .
 (impl_for_bal_ptr balances_class bal_ptr state)
```

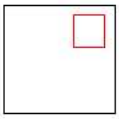
There is one more interface necessary for the implementation below: since Java does not allow for constructors in interfaces, we must have a `consumes_factory` with a method for generating consumers:

```
type consumes_factory_iface_sig =
<|
  generate : 'consumes_factory_ptr -> ('state) -> ('state # 'consumes_ptr)
             -> bool
|>
```

The `consumes_factory` specification says that `generate` returns a consumer that has `eaten` zero tokens:

```
impl_for_con_fac_ptr con_fac_class con_fac_ptr state =
∀ consumes_class .
∀ con_fac_ptr con_ptr state state' state'' how_many .
(con_fac_class.generate con_fac_ptr (state) (state', con_ptr))
==>
((impl_for_con_ptr consumes_class con_ptr state')
 ∧
 (consumes_class.eaten con_ptr (state') (state'', how_many))
 ∧
 (state'' = state')
 ∧
 (how_many = 0))



consumes_factory_obligs
 (consumes_factory_class:('con_fac_ptr,'state,'con_ptr)consumes_factory_iface_sig)=
∀ (con_fac_ptr:'con_fac_ptr) (state:'state) .
(impl_for_con_fac_ptr consumes_factory_class con_fac_ptr state)
```

4.3.2 Specify Components



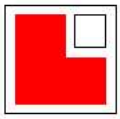
The `consumes` component has two methods:

```
type consumes_iface_sig =
<|
  eat : 'consumes_ptr -> ('state # num) -> ('state) -> bool;
  eaten : 'consumes_ptr -> ('state) -> ('state # num) -> bool
|>
```

The `consumes` specification says that after a token is passed to `eat`, the result of `eaten` is one greater than it would have been before the call to `eat`:

```
impl_for_con_ptr con_class con_ptr state =
(∀ state' state'' n .
 (con_class.eaten con_ptr (state') (state'', n))
 ==>
 (state'' = state'))
∧
(∀ state' state'' state''' token n m .
 (con_class.eat con_ptr (state, token) (state'))
 ∧
 (con_class.eaten con_ptr (state) (state'', n))
 ∧
 (con_class.eaten con_ptr (state') (state''', m))
 ==>
 (m = n+1))



consumes_obligs consumes_class =
∀ con_ptr state .
(impl_for_con_ptr consumes_class con_ptr state)
```

4.3.3 Implement Framework



Following is the Java code for each interface:

```
public interface Consumes {
    public void eat(int token);
    public int eaten();
}

public interface ConsumesFactory {
    public Consumes generate();
}

public interface Balances {
    public void delegate(int token);
    public Consumes[] retrieve();
}
```

The following is the code for the `Balancer` class:

```
public class Balancer implements Balances {

    public Consumes cons1, cons2;
    public boolean inner_cons1_last;
    public Consumes getCons1() { return cons1; }
    public Consumes getCons2() { return cons2; }

    public Balancer(ConsumesFactory cf) {
        cons1 = cf.generate();
        cons2 = cf.generate();
    }
    ...
```

```
    public void delegate(int token) {
        if (inner_cons1_last) {
            cons1.eat(token);
        } else {
            cons2.eat(token);
        }
        inner_cons1_last = !inner_cons1_last;
    }

    public Consumes[] retrieve() {
        Consumes[] result = { cons1, cons2 };
        return result;
    }
}
```

The preceding code is for illustrative purposes. Since the logical translation closely follows earlier examples, it is listed in the Appendix A.5 for interested readers.

There are a few additional predicates that might be necessary for the final goal. Some of these may be part of an entire state theory that includes concepts of aliveness, reachability, disjointedness, etc. such as defined in [PH97] (see Appendix A.7). They might be defined directly by a person along with the shallow embedding. Each of them could also be derivative theorems proven from the properties of the particular state model, particularly if it included an abstract model of memory references with a concept of reachability.

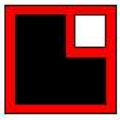
Following are two examples of such theorems. The first states that an object with a different memory reference still satisfies the `balances` obligations after a particular method call:

```
∀ bal2_ptr bal_ptr state state' token (balances_class::balances_obligs) .
balances_class.delegate bal_ptr (state, token) (state')
∧
 (bal2_ptr = bal_ptr)
==>
bal_impl_for_ptr balances_class bal_ptr state')
```

The next theorem is more powerful; it says that the object data is totally unaffected by a particular method call:

```
∀ bal2_ptr bal_ptr state token state' .
balances_class.delegate bal_ptr (state, token) (state')
∧
 (bal2_ptr = bal_ptr)
==>
balances_lookup bal2_ptr state' = balances_lookup bal2_ptr state
```

4.3.4 Verify Framework

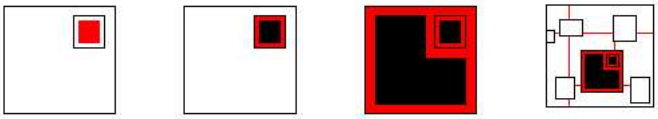


In this case, we need to prove that the behavior is satisfied after other methods (like the constructor) as well as for each interface method. This is the reason for a different specification predicate for a particular object reference. The following goal is much like previous goals, but it includes the fact that a balancer object reference is well-formed rught after being constructed:

```
let balancer_class =
 <|
    delegate := balancer_delgate;
    retrieve := balancer_retrieve
  |>
in
∀ state con_fac_ptr state' bal_ptr .
balancer_balancer (state, con_fac_ptr) (state', bal_ptr)
==>
((wf_bal_obj balancer_class bal_ptr state')
 ∧
 balances_obligs balancer_class)
```

We include the proof for this goal in the appendix (A.5).

4.3.5 Components and Verification



As in the last section, we do not develop the last four stages of component implementation, component proof, overall verification result, and final system deployment.

Chapter 5

Examples of Framework Verification

This section combines our method of framework verification from Section 3 with the logic representation of Java from Section 4 to verify some larger examples that occur in the literature. Since every framework we found involves code-generation or introspection, we simplified each example in ways that preserve the essential framework characteristics.

Section 5.1 shows a formalism for the Visitor design pattern. Since design patterns are by definition very general principles, we have created a specific set of Visitor classes to demonstrate. In other words, we show how a typical example behaves as a framework and how it is translated to logic terms. The only drawback to this example is that it does not fit the second criterion of section 1.2.2: its behavior cannot be defined with the mechanisms developed in this thesis since the behavior always depends on the structure of the data. Therefore we stop after defining the framework and translating the implementation into logic terms.

Section 5.2 illustrates the FileReader framework. A FileReader takes user-defined

classes and creates functions that write and read specific objects of those classes into and out of textual representations. Since we cannot view the structure of a class, we force the framework to accept a different model of a class: an array containing representations for each class datum, each of which may model a different class. This recursive model of the class structure will be the component that completes the framework. For this example, we proceed through the specification and framework proof but do not include a component implementation and proof.

Section 5.3 verifies transaction attributes for an Enterprise JavaBean (EJB) framework (or "*container*") and component (or "*bean*"). This framework automatically handles issues such as data persistence and remote procedure calls for user-defined classes, so EJB tools introspect and generate code like the other examples. In order to analyze an EJB framework we use a single bean which has a single method; it would be simple to extend this process to an arbitrary number of beans each having a different number of methods. We show how all steps of our formal analysis apply to this entire example.

5.1 Visitors

Our first complex example is an application of the visitor design pattern. For any hierarchically structured document such as the parse tree of an XML page, a useful way to extract information from it is to allow visitor objects to traverse the document tree [GHJV95]. This way, a variety of customized visitors can be used to gather specific information. In addition, all the code for each conversion is contained in one place even though it may range over many related data objects. See [FF97] for a more comprehensive explanation of visitors.

This example is different from others in this thesis because the visitor protocol is too complex to specify in general. This section is meant to illustrate the structure of a complex framework. Section 5.1.1 gives an example implementation, followed by

Section 5.1.2 which shows the structure of a visitor specification. In other words, we introduce this example in reverse order starting with the framework and component implementations followed by the specifications in the next section in order to illustrate how this non-trivial framework might be specified in logic terms; no verification is included (see Sections 5.1.2 and 5.1.3 for more explanation).

In terms of our framework diagram, the visitor would be the component interface that the class structure employs (Figure 5.1).

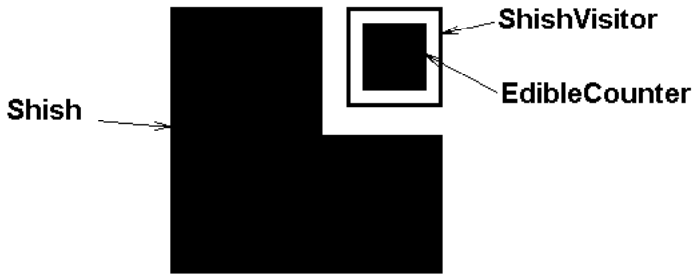
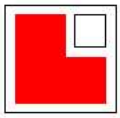


Figure 5.1: Visitor framework-component diagram

5.1.1 Implement Framework



A `Shish` object represents a linked list of items beginning with zero or more foods and ending with a skewer. Since each edible item must be attached to a shich-kebab, they all include a link to the remainder of the kebab. The skewer always comes last.

We define an interface `Shish` with a single method `accept` followed by a visitor class `ShishVisitor` which executes some operation on any given `Shish`. The purpose of the mandatory `accept` method of the `Shish` interface is to allow any `ShishVisitor` to be invoked on any `Shish` structure. Note that this code presupposes the existence of a `Result` class:

```
public interface Shish {
    public Result accept(ShishVisitor sv);
}
```

The following Java code shows four possible implementing classes. There is one skewer and three edible items:

```
public class ShishSkewer implements Shish {
  public ShishSkewer() {}
  public Result accept(ShishVisitor v) {return v.visitSkewer();}
}
public class ShishOnion implements Shish {
  private Shish restOfKebab;
  public ShishOnion(Shish restOfKebab){this.restOfKebab = restOfKebab;}
  public Result accept(ShishVisitor v){return v.visitOnion(restOfKebab);}
}
public class ShishLamb implements Shish {
  private Shish restOfKebab;
  public ShishLamb(Shish restOfKebab){this.restOfKebab = restOfKebab;}
  public Result accept(ShishVisitor v){return v.visitLamb(restOfKebab);}
}
public class ShishTomato implements Shish {
  private Shish restOfKebab;
  public ShishTomato(Shish restOfKebab){this.restOfKebab = restOfKebab;}
  public Result accept(ShishVisitor v){returnv.visitTomato(restOfKebab);}
}
```

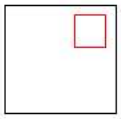
The `ShishVisitor` interface has a method for each class that implements `Shish`:

```
public interface ShishVisitor {
    public Result visitSkewer();
    public Result visitOnion(Shish s);
    public Result visitLamb(Shish s);
    public Result visitTomato(Shish s);
}
```

Finally, we write a visitor which tells how many edible items are on any given shish-kebab. For a skewer, the count is 0; for any other item, the result is the result of the rest of the kebab plus 1. We assume there is an available `Result` class with an `add` method to accumulate the total:

```
public class EdibleCounter implements ShishVisitor {
    public Result visitSkewer() {
        return new Result(0);
    }
    public Result visitOnion(Shish s) {
        return s.accept(this).add(1);
    }
    public Result visitLamb(Shish s) {
        return s.accept(this).add(1);
    }
    public Result visitTomato(Shish s) {
        return s.accept(this).add(1);
    }
}
```

5.1.2 Specify Framework & Components

Since the visitor pattern is a cookbook structure for programmers to use in particular situations, the code must be written (or generated) for each purpose. There is no general framwork or component specification that fits all visitors, and consequently behavioral specifications must also be custom built. Assuming one could build a single abstraction for a node or a tree structure, potential specifications might include guarantees that all nodes get visited or that they get visited in a particular order.

So this example is only meant to show the structure of a visitor specification. We begin with the `Shish` interface which has a single method to accept visitors:

```
type shish_iface_sig = <| accept: 'shish -> 'visitor -> 'result -> bool|>;
```

Each implementation of the `ShishVisitor` interface has a method for each shish-kebab classes:

```
type shish_visitor_iface_sig =
  <|
    visit_skewer: 'visitor -> 'result -> bool;
    visit_onion:  'visitor -> 'shish -> 'result -> bool;
    visit_lamb:   'visitor -> 'shish -> 'result -> bool;
    visit_tomato: 'visitor -> 'shish -> 'result -> bool
  |>
```

Only the `Shish` classes have any data, so the entire structure for the class data is as follows:

```
type shish = Skewer | Onion of shish | Lamb of shish | Tomato of shish
```

Now we can define the class methods. The `accept` method must be implemented by each subclass of `shish`[1] [2]:

```
(shish_obligs tree_funs =
  (∃ visitor_funs::shish_visitor_obligs .
   (∀ vis_obj result . tree_funs.accept (Skewer) vis_obj result =
    visitor_funs.visit_skewer vis_obj result)
   ∧
   (∀ vis_obj s result . tree_funs.accept (Onion s) vis_obj result =
    visitor_funs.visit_onion vis_obj s result)
   ∧ ...
```

[1] In HOL, we can define all four functions inside one definition since they share the same name and take parameters of the same base datatype.

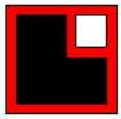
[2] Note that this definition uses `shish_visitor_obligs` which is defined below but whose definition in turn makes use of this one, meaning that the two definitions are mutually recursive and must be defined concurrently.

```
   (∀ vis_obj s result . tree_funs.accept (Lamb s) vis_obj result =
    visitor_funs.visit_lamb vis_obj s result)
   ∧
   (∀ vis_obj s result . tree_funs.accept (Tomato s) vis_obj result =
    visitor_funs.visit_tomato vis_obj s result)
   ))
```

The visitor implements each of the four `visit` methods from the `shish_visitor_iface_sig` interface, one of which (the `Skewer`) calculates a result independantly, and the other three invoke `accept` on the `Shish` objects they receive as arguments. Note that the following does not contain class details (such as `add(1)` from the Java code) since it is concerned only with the obligations for the visitor interface and not specific classes; to instantiate this definition with a class such as `EdibleCounter` we would replace the `result_fun` with the analogy of `add(1)`:

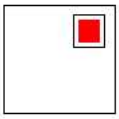
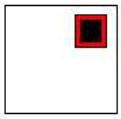
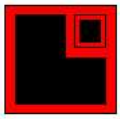
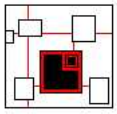
```
(shish_visitor_obligs visitor_funs =
  (∃ shish_funs::shish_obligs .
   (∀ vis_obj .
    (∃ result . visitor_funs.visit_skewer vis_obj result)
    ∧
    (∀ shish result . visitor_funs.visit_onion vis_obj shish result =
     (∃ result2 result_fun . shish_funs.accept shish vis_obj result2
                  ∧ result = result_fun result2)
    ∧
    (∀ shish result . visitor_funs.visit_lamb vis_obj shish result =
     (∃ result2 result_fun . shish_funs.accept shish vis_obj result2
                  ∧ result = result_fun result2)
    ∧
    (∀ shish result . visitor_funs.visit_tomato vis_obj shish result =
     (∃ result2 result_fun . shish_funs.accept shish vis_obj result2
                  ∧ result = result_fun result2)
    )))
```

5.1.3 Verify Framework



The visitor pattern has proven extremely useful for some tasks. Unfortunately, since each application of this pattern is human-intensive, its analysis is out of the range of current verification techniques and beyond the scope of this thesis. So we stop at the framework specification above, which is interesting in its own right.

5.1.4 Components and Verification

As just explained, we do not develop the last four stages of component implementation, component proof, overall verification result, and final system deployment.

5.2 The File Reader

In this section, we model a framework which Woolf calls a "File Reader" [Woo99]. It takes a tree object structure and constructs one or more objects with that structure from data in an input stream.

For example, consider a `Person` and an `Address` object. `Person` contains the string fields `name` and `occupation` along with an `Address` field which contains five more string fields. The data for a given `Person` could be flattened into seven fields; a `FileReader` could then be used to reconstruct the original `Person` data from those seven fields. Going further, given simply an input stream that uses a delimiter to separate fields, a `FileReader` could reconstruct a list of `Person` objects with nested `Address` objects, one for each seven fields from the stream.

The original `FileReader` is fairly complex, employing language features such as introspection to set field values. It must be modified to create formal definitions. In fact, the methods below barely resemble the original framework for two reasons: we do not have a rich enough logic to reason about introspection, and we want to show example components that have well-defined interfaces and behaviors.

So where the published file reader works with files or input streams, our version converts string arrays into objects (and vice-versa). And where the published version determines the structure of the objects mechanically by introspection, our version requires components which convert a single object to a string (and vice-versa).

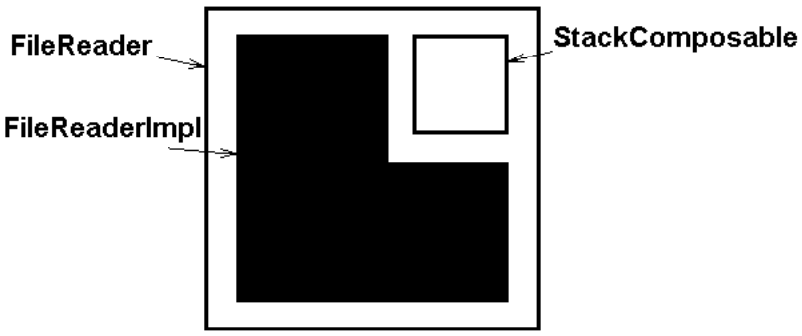


Figure 5.2: "FileReader" framework-component diagram

Components of the `StackComposable` interface require two methods:

- `construct` takes a list of strings and a list of objects, each of which is treated as a stack: it pops values from the string list and possibly the object list, creates an object out of the values, and pushes the result object onto the object list. It returns the new version of both lists.

- `destruct` also accepts and returns a list of strings and a list of objects but does the opposite: it takes the first object and converts it back into string and object values which it pushes onto the respective list.

The real-world `FileReader` does not require the opposing `destruct` operator; it is included in this formalism in order to be able to state a formal correctness condition. Without that, there is only an operation that converts strings to objects without any mechanism to tell whether that operation works correctly.

The final framework takes a list of `StackComposable` objects along with the string list of values and creates the hierarchical object structure.

5.2.1 Specify Framework

The `FileReader` has two operations. The method `of_list` takes a list of `StackComposable` objects and creates an object from a string list. The method `to_list` does the opposite: it converts an object into a flattened list of string values.

```
public interface FileReader {
    public Object to_list(StackComposable[] stack_composables,
                          String[] strings);
    public String[] of_list(StackComposable[] stack_composables,
                            Object obj);
}
```

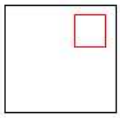
This behavioral specification says that these two are essentially inverses of each other[3].

```
type file_reader_iface_sig =
<|
  of_list: (('obj)stack_composable_iface_sig list # string list) -> ('obj) -> bool;
  to_list: (('obj)stack_composable_iface_sig list # 'obj) -> (string list) -> bool
|>
```

[3]The inputs to `to_list` may contain extra elements (represented by `morestr`); this is becaus it will finish and return an object as soon as all the `StackComposers` have been applied successfully. Also note that the `to_list` requires the `StackComposable` objects in reverse order; this was an implementation decision which may not be necessary for proper functionality.

```
file_reader_obligs file_reader =
(∀ stack_composers strlist objlist morestr moreobj .
 file_reader.of_list (stack_composers, APPEND strlist morestr) (obj)
 =
 file_reader.to_list (REVERSE stack_composers, obj) (strlist))
```

5.2.2 Specify Components



`StackComposable` objects operate on two stacks, a string stack and an object stack, each being a list of elements.

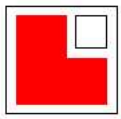
The input and output stacks are combined in the `StringObjectStacks` class:

```
public interface StackComposable {
    public class StringObjectStacks { Vector strings, objects; }

    public StringObjectStacks construct(StringObjectStacks stacks);
    public StringObjectStacks destruct(StringObjectStacks stacks);
}
```

As for their behavior, a `construct` operation always causes the string list to shrink in size (specified by the first two conjuncts of `stack_composable_obligs`). A `construct` followed by a `destruct` restores the string and object stacks (which is the last part of `stack_composable_obligs`).

```
type stack_composable_iface_sig =
<|
  construct : (string list # 'Object_with_create list)
               -> (string list # 'Object_with_create list) -> bool;
  destruct  : (string list # 'Object_with_create list)
               -> (string list # 'Object_with_create list) -> bool
|>

stack_composable_obligs stack_composer =
(∀ strlist objlist strlist2 objlist2 .
 stack_composer.construct (strlist, objlist) (strlist2, objlist2)
 ==>
 LENGTH strlist2 < LENGTH strlist)
∧
(∀ strlist objlist strlist2 objlist2 .
 stack_composer.destruct (strlist2, objlist2) (strlist, objlist)
 ==>
 LENGTH strlist2 < LENGTH strlist)
∧
(∃ strlist objlist objlist2 .
 stack_composer.construct (strlist, objlist) ([], objlist2)
 ∧
 stack_composer.destruct ([], objlist2) (strlist, objlist)
 ∧
 (LENGTH objlist2 = 1))
```

5.2.3 Implement Framework



This framework is relatively simple to implement since the `StackComposer` components do most of the work. The most difficult part is keeping the roles in mind. Each `StackComposer` has its own version of `construct` that pops from the

`StringObjectStacks` to create and push another `Object` onto the stack, and each `destruct` moves in the other direction; this framework's job is simply to delegate the actions:

```
public class FileReaderImpl implements FileReader {

    public Object of_list(StackComposable[] stack_composables,
                          String[] strings) {
        Object[] objects = {};
        StringObjectStacks stacks =
            new StringObjectStacks(objects, strings);
        int i = 0;
        while (stacks.objects.size() > 0) {
            stacks = stack_composables[i++].construct(stacks);
        }
        return stacks.objects.elementAt(i);
    }

    public String[] to_list(StackComposable[] stack_composables,
                            Object obj) {
        Object[] objects = { obj };
        String[] strings = {};
        StringObjectStacks stacks =
            new StringObjectStacks(objects, strings);
        int i = 0;
        while (stacks.objects.size() > 0) {
            stacks = stack_composables[i++].destruct(stacks);
        }
        return (String[])stacks.strings.toArray(new String[0]);
    }

}
```

Following is the logical definition of `of_list_fun` . The workhorse is `of_list_fun2` which always succeeds for the base case of no `stack_composers`, and which applies the `construct` method and recurses through the rest of the list when there are more to

apply [4] [5] . The `of_list_fun` method sets up `of_list_fun2` with the right parameters.

```
(of_list_fun2 ([], strlist, objlist:'a list)
                (strlist2:string list, objlist2:'a list)
  = T)
∧
(of_list_fun2 (CONS stack_composer stack_composers, strlist, objlist)
                (strlist2, objlist2) =
 (stack_composable_obligs stack_composer
  ==>
  (∃ strlist' objlist' .
   (stack_composer.construct (strlist, objlist) (strlist', objlist'))
   ∧
   (of_list_fun2 (stack_composers, strlist', objlist') (strlist2, objlist2)))))

(of_list_fun (stack_composers, strlist) (obj) =
 (∃ strlist' objlist' .
  of_list_fun2 (stack_composers, strlist, []) (strlist', objlist')
  ∧
  (obj = HD objlist')))
```

Following is the logical definition of `to_list_fun`. It works much the same way: by calling the recursive `to_list_fun2` with the right parameters it forces a `destruct` on each memer of the list of objects.

[4] The definition of `of_list_fun2` cannot be created automatically; the following measurement function can be used with HOL's `Defn` library:

```
list_meas ((a:'a stack_composable_iface_sig list, b:string list, c:'a list),
x:string list, y:'a list)
= LENGTH a
```

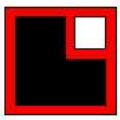
[5] Note the different syntax to restrict all `stack_composer` components to satisfy `stack_composable_obligs`. Inside definitions, an implies (⟶) must be used because restricted quantification (::) is only available with the forall (∀) and exists (∃) bindings.

```
(to_list_fun2 ([], strlist, objlist:'a list)
              (strlist2:string list, objlist2:'a list)
  = T)
∧
(to_list_fun2 (CONS stack_composer stack_composers, strlist, objlist)
                (strlist2, objlist2) =
 (stack_composable_obligs stack_composer
  ==>
  (∃ strlist' objlist' .
   (stack_composer.destruct (strlist, objlist) (strlist', objlist'))
   ∧
   (to_list_fun2 (stack_composers, strlist', objlist') (strlist2, objlist2)))))

(to_list_fun (stack_composers, obj) (strlist:string list) =
 (∃ strlist' .
  to_list_fun2 (stack_composers, [], [obj]) (strlist', [])))
```

The `FileReaderImpl` class is made of these two methods as follows:

```
file_reader_class = <|
  of_list := of_list_fun;
  to_list := to_list_fun
|>
```

5.2.4 Verify Framework

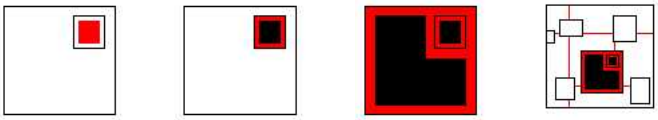


The theorem for the File Reader follows the familiar pattern:

```
⊢ file_reader_obligs file_reader_class
```

The proof of this theorem proceeds much like earlier proofs (see Section 3.2.1.4).

5.2.5 Components and Verification



For this example, we do not develop the last three stages of component implementation, component proof, overall verification result, and final system deployment.

5.3 EJB transactions

Finally, this section presents a framework that manages transactions for Enterprise JavaBeans (EJBs).

As a standard for independantly deployable components, the EJB specification defines roles for the different parties that participate and spells out actions that occur from development through deployment and maintenance. However, we are only interested in the technical details of EJBs, especially how their properties can be formally defined and verified.

The EJB specification version 1.1 is 558 pages in size with additional errata documents [MH99]. For this example, we chose one area where the code could be treated as a framework with well-defined behavioral specifications, then verify code involving reusable components. Out of all the issues addressed in the documentation, only two appear to be defined well enough for formal analysis: security and transactions. However, only transactions appear to have functionality that is modular enough to be organized into some type of component-framework. Throughout this section, the framework formalisms are developed at the same time the framework is explained.

5.3.1 Specify Framework

The system is structured as an EJB framework that delegates user requests to component methods which do the work. Each method call can have an associated transaction assigned to it that guarantees atomicity and consistency; it maintains the connection to the database and ensures that all data changes happen or they all fail,

and that other processes viewing the data see all the changes if they see any at all.

Component designers have the option of maintaining all their own transactions with the database and explicitly controlling when changes commit or not. However, they can choose "container-managed" transactions where the framework manages some of the database details so the component designers do not have to. For example, the component can be marked as always needing a new transaction for its work so that its actions are not combined with any other methods' database actions (except those it explicitly invokes). This is exactly the behavior that the "transaction demarcation" attributes specify; these are set with a "deployment descriptor" in a location separate from programming code. That particular behavior is named `RequiresNew`.

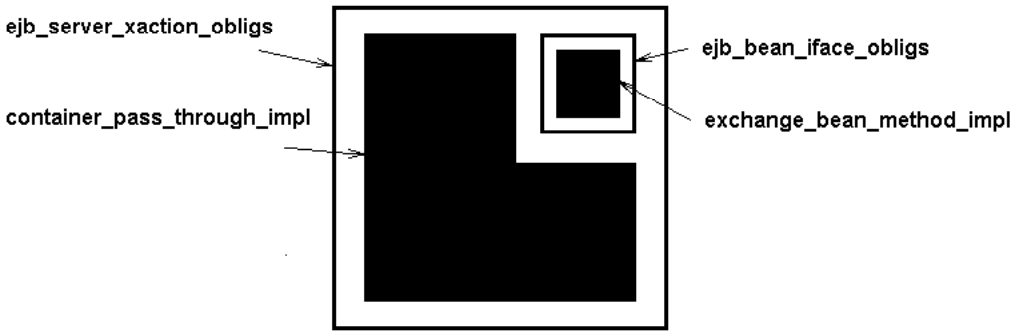


Figure 5.3: EJB framework-component diagram

This section specifies how the example container framework and component together preserve transaction demarcation attributes. We begin with Table 5.1 showing the effect of each attribute. We then specify the behavior a transaction, whose syntax is essential to the framework code and whose semantics are essential to the framework validation (Section 5.3.1.1).

The framework and component specifications come directly from the "Transaction attribute summary" table on pages 357-8 of [MH99]. It summarizes the transaction made available to the business method and the framework "resource managers", or

Transaction attribute	Client's transaction	Transaction associated with business method	Transaction associated with resource managers
NotSupported	none	none	none
	T1	none	none
Required	none	T2	T2
	T1	T1	T1
Supports	none	none	none
	T1	T1	T1
RequiresNew	none	T2	T2
	T1	T2	T2
Mandatory	none	error	N/A
	T1	T1	T1
Never	none	none	none
	T1	error	N/A

Table 5.1: Transaction Attribute Behavior Summary

framework subsystems that control the database connections. The label `T1` is the transaction passed with the client request, while `T2` is a transaction initiated by the framework.

For example, line 2 states that when a component deployed with a "Required" attribute is called by a process that does not already have a transaction, the resource managers create a new transaction which is used by the business logic.

First we formally specify what a transaction component does since that is critical for defining the correct operation of the framework. Then we give the formal specification of the framework.

5.3.1.1 A Transaction Component

A transaction is defined in terms of the following actions:

- A `new` returns a transaction object which modifies the database, often called a `Connection` in programming code.
- A `modify` updates a particular database table entry with a new value datum. For this thesis, the table and entry identifiers are strings and the written value is a number.
- A `read` retrieves the current value of a particular table entry. Again, the table and entry are strings and the accessed value is a number.
- A `commit` permanently commits updates from `modify` commands.
- A `rollback` undoes any transaction changes since the `new` or the most recent `commit`.

The following is the type signature for that interface:

```
type transaction_iface_sig = <|
  new : 'state -> 'state # xaction_ptr -> bool;
  modify : xaction_ptr -> 'state # string # string # num -> 'state # bool ->bool;
  read : xaction_ptr -> 'state # string # string -> 'state # num -> bool;
  commit : xaction_ptr -> 'state -> 'state -> bool;
  rollback : xaction_ptr -> 'state -> 'state -> bool
|>
```

A transaction has the following two behaviors (spanning lines 1-3 and 4-14 respectively in the specification below):

- The `new` method (line 2) creates a non-null transaction object.

- A `modify` (line 6) causes a subsequent `read` (line 7) to return the same data. In addition, calls to `rollback` (lines 8-11) and `commit` (lines 12-14) work as follows:

  - A `rollback` (line 8) causes a subsequent `read` (line 9) to return the same result as a `read` (line 10) before creating the transaction (ie. the state before the `new` named `state'pre_begin`).

  - A `commit` (line 12) causes a subsequent `read` (line 13) to return the same result as the `read` (line 7) before creating the transaction.

```
     transaction_iface_obligs xaction =
1)    (∀ trans_ptr state'pre_begin state'post_begin .
2)     (xaction.new (state'pre_begin) (state'post_begin, trans_ptr)
3)      ==> ¬(trans_ptr = Xaction_Null)))
      ∧
4)    (∀ trans_ptr state'pre_begin state'post_begin
       state'pre_end state'post_roll state'post_commit
       table row data success .
5)     (¬(trans_ptr = Xaction_Null))
        ∧
6)      (xaction.modify trans_ptr (state'post_begin, table, row, data)
                                  (state'pre_end, success)))
       ==>
7)     (xaction.read trans_ptr (state'pre_end, table, row)
                               (state'pre_end, data)
        ∧
        ...
```

```
8)      ((xaction.rollback trans_ptr (state'pre_end) (state'post_roll)
          ∨ ¬ success)
         ==>
         (∃ data2 data3 .
9)        xaction.read trans_ptr (state'post_roll, table, row)
                                 (state'post_roll, data3)
          ∧
10)       xaction.read trans_ptr (state'pre_begin, table, row)
                                 (state'pre_begin, data2)
          ∧
11)       (data2 = data3)))
        ∧
12)     ((xaction.commit trans_ptr (state'pre_end) (state'post_commit)
          ∧ success)
         ==>
         (∃ data2 .
13)       xaction.read trans_ptr (state'post_commit, table, row)
                                 (state'post_commit, data2)
          ∧
14)       (data2 = data)))))
```

5.3.1.2 An EJB Container Framework

In EJB terminology, the framework is a "*container*" and the components are "*beans*". The container framework acts as an intermediary between the requestor and the bean component. It intercepts calls to the component and manages server resources, such as transactions in this example. In particular, the framework in this section will create and pass transactions to the bean according to the selected transaction attribute (Table 5.1).

Like the other industrial examples in this thesis, this framework is simplified. A framework can normally delegate any number of messages to any number of compo-

nents, but that requires code generation tools and reasoning which is not available in logic. Plus, that would be overly complex for this illustration. This example shows a framework with a single method which it delegates to a single component.

First we declare the available transaction attributes:

```
type xaction_attr =
  NotSupported | Required | Supports | RequiresNew | Mandatory | Never
```

Now we are ready to specify the transactional EJB framework behavior. A framework has a single `container_method` which accepts incoming requests to delegate to the bean component. Besides the component and framework object on which it operates, the method takes four arguments: the memory (of type `'state`), the transaction attribute (of type `xaction_attr`), the transaction object of the client (of type `xaction_ptr`), and any arguments for the business logic (of type `'args`). It returns four arguments: the new memory (of type `'state`), the two transation objects supplied to the business method component and resource managers respectively (of type `xaction_ptr`), and the return value of the component method (of type `'val`). Following is the type signature of this method:

```
type ejb_server_sig = <|
  container_method :
    container_ptr
    -> 'state # xaction_attr # xaction_ptr # 'args
    -> 'state # xaction_ptr # xaction_ptr # 'val
    -> bool
|>
```

The overall obligations are a formal statement of the EJB transation attribute summary (Table 5.1) before and after the method call[6]. They begin with a call to the `container.container_method` on the phrase marked `1` below.

[6]The name of `trans4` is correct; `trans3` is used later in the implementation, where there is an internal transaction between `trans2` and `trans4`.

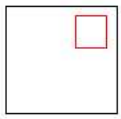
The phrases marked 2 through 7 express the behavior of each of the entries in Table 5.1. For example, phrase 3 says that if the transaction attribute is "Required" then there are two cases: the transaction from the client may be null, in which case a new transaction is created for both the component and resource manager; otherwise there is a transaction from the client which both the component and the resource manager use.

```
    ejb_server_xaction_spec (server:('state, 'args, 'val) ejb_server_sig)
    =
    ∃ xaction::transaction_iface_obligs .
    ∀ container_ptr state'pre state'post
      trans1 trans2 trans4 trans_attr
      (x:'args) (y:'val) .
1)  server.container_method container_ptr
      (state'pre, trans_attr, trans1, x)
      (state'post, trans2, trans4, y)
    ==>
2)  ((trans_attr = NotSupported) ==>
      ((trans2 = Xaction_Null)
      ∧ (trans4 = trans2)))
    ∧
3)  ((trans_attr = Required) ==>
      (((trans1 = Xaction_Null)
        => ((trans4 = trans2)
            ∧ (∃ (state'any_pre_bean:'state) state'any_pre_bean2 .
                xaction.new (state'any_pre_bean)
                            (state'any_pre_bean2, trans2)))
        | (trans2 = trans1)
          ∧ (trans4 = trans1))))
    ∧
    ...
```

```
4)  ((trans_attr = Supports) ==>
      ((trans2 = trans1)
       ∧ (trans4 = trans1)))
    ∧
5)  ((trans_attr = RequiresNew) ==>
      ((trans4 = trans2)
       ∧ (∃ state'any_pre_bean state'any_pre_bean2 .
            xaction.new (state'any_pre_bean)
                        (state'any_pre_bean2, trans2))))
    ∧
6)  ((trans_attr = Mandatory) ==>
      ((¬(trans1 = Xaction_Null))
       ∧ (trans2 = trans1)
       ∧ (trans4 = trans1)))
    ∧
7)  ((trans_attr = Never) ==>
      ((trans1 = Xaction_Null)
       ∧ (trans2 = trans1)
       ∧ (trans4 = trans1)))
```

Note that the definition depends on a class that satisfies the transaction obligations. This is done with the internal reference to the `transaction_iface_obligs`.

5.3.2 Specify Components



Our component is a single EJB bean. Since most of the critical component code is simple and does not change between components, introspection tools are used to generate most of the code so that component developers need not worry about the details of databases and maintaining transactions. However, automated code-generation is difficult to specify let alone reason about. So the behavior we specify for this example is exactly the behavior that the EJB tools generate automatically for

the developers. We realize that this particular example will not encourage developers to verify their own component behavior, but it may be useful for EJB tool writers as well as being good for this framework illustration.

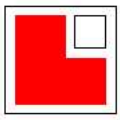
While in the real world a bean component may have any number of methods and any kind of signatures, it is not possible to deal with all those variations in this thesis. The property with which we are concerned is the state of the transaction before and after the method call, which must conform to the table above. So a component is simplified to contain one method which takes a transaction and other arguments and returns the transaction and any other arguments.

```
type ejb_bean_iface_sig = <|
  bean_method :
    bean_ptr -> state # xaction_ptr # 'args -> state # xaction_ptr # 'val -> bool
|>
```

The bean component obligation states that the transaction remains unchanged after the method call. This is a simple obligation, but it is a critical fact in the proof that the entire framework acts correctly:

```
ejb_bean_iface_obligs bean =
∀ bean_ptr state'pre_bean state'post_bean trans1 trans2 x y .
   bean.bean_method bean_ptr (state'pre_bean, trans1, x)
                             (state'post_bean, trans2, y)
   ==> (trans1 = trans2)
```

5.3.3 Implement Framework



The implementation we studied is the JOnAS EJB container framework and its set of bean component generation tools. We are focused only on the transactional aspect of method calls to the component, so we do not include incomplete code snippets or

large blocks of unrelated code. The following framework implementation is restricted to the details of the transaction, but it accurately captures the entire transaction behavior across method boundaries.

The implementation of method `pass_through_method` is a bit different from previous implementations because it takes the bean component as one of the arguments. An EJB framework must be configured with all the component information, which is usually done through introspection and/or property files. Here this process is emulated by including the component as an argument to the framework's entry method.

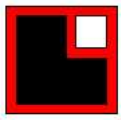
The semantics below show how the framework implementation interacts with the component. There are a total of four transaction variables throughout this implementation: `trans1` and `trans4` are the transactions shared with the client process at the beginning and end of the call to `pass_through_method` respectively; `trans2` and `trans3` are the transactions shared with the component process at the beginning and end respectively, and they may or may not be different from the `trans1` and `trans4` transaction depending on what transaction attribute was chosen at deployment.

For example, phrase `3` implements the "Required" functionality for a transaction which is described in the specification above. Note how the correctness of this case depends on the behavior of the `bean.bean_method`, which is where the relationship between `trans2` and `trans3` is defined.

```
    container_pass_through_impl
      (trans_class, bean_class)
      container_ptr
      (state'pre_skel:'state, trans_attr, trans1, w)
      (state'post_skel, trans2, trans4, z)
    =
    ∃ bean_ptr state'pre_bean state'post_bean
      state'any_pre_bean:'state state'any_pre_bean2
      trans3 .
1)  (bean_class.bean_method bean_ptr
      (state'pre_bean, trans2, w)
      (state'post_bean, trans3, z))
    ∧
2)  ((trans_attr = NotSupported) ==>
      ((trans2 = Xaction_Null)
       ∧ (trans4 = Xaction_Null)))
    ∧
3)  ((trans_attr = Required) ==>
      (((trans1 = Xaction_Null)
        => (trans2 = trans1)
            ∧ (trans4 = trans3)
            ∧ (trans_class.new state'any_pre_bean
                              (state'any_pre_bean2, trans2))
        | (trans2 = trans1)
          ∧ (trans4 = trans3))))
    ∧
4)  ((trans_attr = Supports) ==>
      ((trans2 = trans1)
       ∧ (trans4 = trans3)))
    ∧
    ...
```

```
5)  ((trans_attr = RequiresNew) ==>
      ((trans4 = trans3)
       ∧ (trans_class.new state'any_pre_bean
                          (state'any_pre_bean2, trans2))))
    ∧
6)  ((trans_attr = Mandatory) ==>
      ((¬(trans1 = Xaction_Null))
       ∧ (trans2 = trans1)
       ∧ (trans4 = trans3)))
    ∧
7)  ((trans_attr = Never) ==>
      ((trans1 = Xaction_Null)
       ∧ (trans2 = trans1)
       ∧ (trans4 = trans3)))
```

5.3.4 Verify Framework

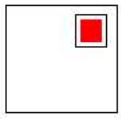


The correctness statement resembles earlier goals, where the customizable interfaces are each written with restricted universal quantification; here, the component interfaces are the transaction and bean class variables:

```
⊢ ∀ (trans_class::transaction_iface_obligs)
        (bean_class::ejb_bean_iface_obligs) .
  ejb_container_xaction_obligs
  <| container_method := container_pass_through_impl (trans_class, bean_class) |>
```

Again, the proof of this theorem resembles earlier proofs. The entire HOL proof script is found in Section A.6.1.

5.3.5 Implement Component



For this last example we complete the process for the client.

The bean component depends on three other components: the `customer_class` and `seller_class` have identifiers (accessed by "`getId()`") which are used to find their rows in their respective tables, and the `stock_class` includes a stock price which will be the value used to update each entry.

Since our examples do not deal with exceptions (see Section 4.2.4, we rely on a boolean "`success`" values to tell whether the updates succeed.

```
public class StockInterfaceBean {
  ...
  public static boolean exchange(Connection xaction,
                                 CustomerAccount account,
                                 StockHolder seller,
                                 StockInfo stock) {
1)  boolean success1 = true;
    try {
      xaction.executeStatement...
    } catch (Exception e) {
      success1 = false;
    }

2)  boolean success2 = true;
    try {
      xaction.executeStatement...
    } catch (Exception e) {
      success2 = false;
    }

3)  if (!(success1 /\ success2)) {
4)    xaction.rollback();
    } else {
5)    xaction.commit();
    }
  }
  ...
}
```

The following logic version of that Java code requires some explanation. Lines 1-4 are parameters of the implementation method: line 1 is the list of external com-

ponents; line 2 is the object reference for the component upon which this method is called; line 3 is the list of inputs, including the transaction which is made available by code-generation tools; and line 4 is the list of outputs.

In the body of the Java class above, phrases 1 and 2 correspond with lines 11 and 12 below respectively. Note that variables such as the `customer_id` and `state'mid1` are linked to other phrases where those variables are set or accessed. Finally, lines 3, 4, and 5 above correspond to lines 16, 17-18, and 19-20 below respectively.

It may help to describe the intent of the logic.

- Line 5 states that the transaction pointer remains unchanged (since the output value is the same as the input value).

- Lines 6-8 retrieve the customer and seller IDs and the stock price from their respective objects into local variables.

- Lines 9-10 access the database table entries which may be modified later; this is to ensure that the initial values remain unchanged upon a rollback.

- Lines 11-12 execute the update (or "`modify`") statement to fill the database tables with new values.

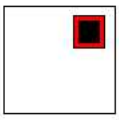
- Lines 13-14 relate the input and output machine states to the internal states.

- Line 15 checks whether the table updates were successful. Upon success of both, lines 16-17 show that the `stock_price` is successfully written to each table; however, upon failure of either, lines 18-19 show that the data originally in each table remains as if nothing happened.

```
    exchange_bean_method_impl
1)    (trans_class, customer_class, seller_class, stock_info_class)
2)    bean_ptr
3)    (state'pre, trans_attr, trans_ptr, customer_ptr, seller_ptr, stock_info_ptr)
4)    (state'post, trans_ptr2) =
    ∀ data1 data2 success1 success2 .
    ∃ state'mid1 state'mid2 state'mid3 customer_id seller_id stock_price .
5)  (trans_ptr2 = trans_ptr)
    ∧
6)  customer_class.get_cust_id customer_ptr (state'pre, Void)
                                                (state'pre, customer_id)
    ∧
7)  seller_class.get_sell_id seller_ptr (state'pre, Void)
                                             (state'pre, seller_id)
    ∧
8)  stock_info_class.get_price stock_info_ptr (state'pre, Void)
                                                    (state'pre, stock_price)
    ∧
9)  trans_class.read trans_ptr (state'pre, "customer", customer_id)
                                  (state'pre, data1)
    ∧
10) trans_class.read trans_ptr (state'pre, "seller", seller_id)
                                  (state'pre, data2)
    ∧
11) trans_class.modify trans_ptr (state'mid1, "customer", customer_id,stock_price)
                                    (state'mid2, success1)
    ∧
12) trans_class.modify trans_ptr (state'mid2, "seller", seller_id, stock_price)
                                    (state'mid3, success2)
    ∧
    ...
```

```
13) (state'mid1 = state'pre)
    ∧
14) (state'mid3 = state'post)
    ∧
15) if (success1 ∧ success2)
    then
16)      (trans_class.read trans_ptr (state'post, "customer", customer_id)
                                      (state'post, stock_price)
         ∧
17)      trans_class.read trans_ptr (state'post, "seller", seller_id)
                                     (state'post, stock_price))
    else
18)      (trans_class.read trans_ptr (state'post, "customer", customer_id)
                                      (state'post, data1)
         ∧
19)      trans_class.read trans_ptr (state'post, "seller", seller_id)
                                     (state'post, data2))
```

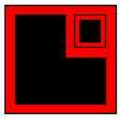
5.3.6 Verify Component



Following is the goal to prove this bean component correct. It states that when the method `exchange_bean_method_impl` is the implementation for a component method, it satisfies the `ejb_bean_iface_obligs` in conjunction with the other (`trans_class`, `customer_class`, etc.) components.

```
⊢ ∀ (trans_class::transaction_iface_obligs)
  customer_class seller_class stock_info_class .
  ejb_bean_iface_obligs
  <| bean_method := exchange_bean_method_impl
        (trans_class, customer_class, seller_class, stock_info_class)
   |>
```

5.3.7 Final System Framework Properties

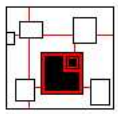


The goal of a verified EJB server is stated as follows: when deployed with a bean component implemented by the `exchange_bean_method_impl` method, a framework implememented by the `container_pass_through_impl` method satisfies the framework obligations `ejb_container_xaction_obligs`.

```
⊢ ∀ (trans_class::transaction_iface_obligs)
  customer_class seller_class stock_info_class .
  ejb_container_xaction_obligs
  <| container_method :=
      container_pass_through_impl
        (trans_class,
         <| bean_method :=
              exchange_bean_method_impl
               (trans_class, customer_class, seller_class, stock_info_class)
          |>)
   |>
```

As always, the final system properties follow automatically from the framework and component proofs.

5.3.8 Final System Client Properties



Finally, the end users are not so much interested in the details of transaction safety as they want to know that their transaction completely succeeds or it competely fails. The following specification states that the customer's purchase either all succeeded or all failed.

- Line 1 is all the external components that contribute to this specification. Line 2 is the argument representing the class being validated.

- Line 3 shows how the framework's `exchange` method is invoked. Lines 4-6 state

that the transaction attribute must not allow a null transaction.

- Lines 7-9 access the customer's and seller's IDs and the stock price for later use.
- Lines 10-12 state the crucial facts about the final state of the transaction: either the new stock price has been updated to both tables (line 10), or both tables still have the old values (lines 11-12).

```
    type exchange_gateway_sig =
    <| exchange : container_ptr
                  -> ('state # xaction_attr # xaction_ptr
                      # (customer_ptr # seller_ptr # stock_info_ptr))
                  -> ('state # xaction_ptr # void)
                  -> bool
    |>

    exchange_spec
1)    (trans_class, customer_class, seller_class, stock_info_class)
2)    exchange_class =
    ∀ trans_attr
      container_ptr trans_ptr customer_ptr seller_ptr stock_info_ptr
      state'pre state'post .
3)  (exchange_class.exchange
     container_ptr
     (state'pre, trans_attr, trans_ptr, customer_ptr, seller_ptr, stock_info_ptr)
     (state'post, trans_ptr, Void)
     ∧
4)   ¬(trans_attr = Supports)
     ∧
5)   ¬(trans_attr = NotSupported)
     ∧
     ...
```

```
6)   ¬(trans_attr = Never)
     ==>
     (∃ customer_id seller_id stock_price .
7)    (customer_class.get_cust_id customer_ptr (state'pre, Void)
                                                  (state'pre, customer_id)
       ∧
8)     seller_class.get_sell_id seller_ptr (state'pre, Void)
                                              (state'pre, seller_id)
       ∧
9)     stock_info_class.get_price stock_info_ptr (state'pre, Void)
                                                    (state'pre, stock_price))
      ==>
10)   ((trans_class.read trans_ptr (state'post, "customer", customer_id)
                                    (state'post, stock_price)
        ∧
        trans_class.read trans_ptr (state'post, "seller", seller_id)
                                    (state'post, stock_price))
         ∨
11)     (∃ data1 data2 .
        trans_class.read trans_ptr (state'post, "customer", customer_id)
                                    (state'post, data1)
        ∧
        trans_class.read trans_ptr (state'post, "seller", seller_id)
                                    (state'post, data2)
        ∧
12)     trans_class.read trans_ptr (state'pre, "customer", customer_id)
                                    (state'pre, data1)
        ∧
        trans_class.read trans_ptr (state'pre, "seller", seller_id)
                                    (state'pre, data2)))))
```

The final theorem says that the class constructed with the container framework

methods from Section 5.3.1.2 and the bean component methods from Section 5.3.1.2 satisfy those exchange_spec obligations:

```
⊢
∀ trans_class::xaction_obligs .
∀ bean_class::ejb_bean_xaction_obligs .
∀ customer_class seller_class stock_info_class .
exchange_spec (trans_class, customer_class, seller_class, stock_info_class)
<|
  exchange :=
    container_pass_through_impl
      (trans_class,
       <| bean_method :=
            exchange_bean_method_impl
              (trans_class, customer_class, seller_class, stock_info_class
        |>))
|>
```

Chapter 6

Conclusions and Future Work

This work identifies frameworks as a specific type of high-level system whose concrete implementations can be formally analyzed. It shows how component interfaces can be used to build abstract theories of framework systems such that particular behaviors can be verified, then demonstrates how to formally verify some example frameworks.

This thesis incorporates hardware verification principles into object-oriented software verification. Specifically, it shows how to apply predicate specifications and abstract theories to interfaces and classes. Predicates showing the relationships between interface functions are well-suited for the abstract theories expressed by frameworks. The operational semantics are straightforward, often very similar to programming constructs, and are shown to apply to programs involving subtyping and mutually recursive functions.

Most of the effort in this work was involved in specifying each problem. This was the most significant problem we encountered, and that is the biggest surprise in this work: although designing a formal model for each behavior proved difficult, by far the most challenging task was simply organizing each system into manageable pieces such that the behavior of each part could be formally defined. This was true even

for the frameworks carefully analyzed in previous publications. We see this as one of the biggest barriers to widespread sharing of verification results, along with the organization it takes to share these carefully crafted specifications. Each programming language and library has slightly different semantics, usually for good reasons, and although humans are adept at handling these problems, they make shared verification efforts intractible if not impossible. Thus we emphasize again that the only proper subjects for widespread verification are framework systems that are already stable and well-understood.

While this work is useful for most object-oriented language features, we found a large class of frameworks that this approach does not handle: many frameworks generate class code or use object introspection to incorporate the components into the framework. In fact, all the large systems we studied used one of these features to some extent in order to use the components. Reasoning about such systems will require a much more advanced set of tools.

Finally, we formally specified transaction attributes found in the EJB specification and verified that a particular combination of server and bean satisfies it. It is our hope that others will find this useful for further examination of EJB behavior or transactions.

6.1 Future Work

This thesis was an exploration into software verification using many of the same theorem proving tools found in hardware verification. It shows their effectiveness and extends their methods for large, component-based systems. Although we have shown the potential for system-wide verification, we believe the evolution of software verification will follow the route taken by the hardware community: the first tools to become popular handle common low-level mistakes such as bad memory references; system-level abstractions are still difficult enough to be left to specialists in very

focused, proprietary endeavors. We discovered in this work that the effort of writing component and system specifications is almost prohibitive. So the next steps along this line of research should address the following points:

- Is there a programming language and tool independant way to share component abstractions and specifications?
- How are specifications for common ideas (eg. transactions and security) written such that they are easy to understand and share?
- How can the actions of code-generation tools be expressed formally? How can the properties of those tools or their output be specified?
- We found that our specifications evolved together with the framework and component implementations and verifications. Is this inherent in the nature of large systems? Is there a point of stability at which the formal definitions cover most of the customers' needs? If not, would it be beneficial to share multiple formalisms of a framework?

While everyday verification of high-level abstractions remains future work, some of the more complex low-level details may be good material for automatic verification by model-checkers: object reachability, liveness, and even equivalence. Theorem-provers would be more useful if they handled some of these problems by delegating them to symbolic model-checking or symbolic trajectory evaluation tools.

This work dealt with state in an ad-hoc way for each example. There are more comprehensive theories such as [PH97] to handle object references; these could ease verification involving more complex object data structures. In fact, [PH97] contains a theory of object data which could be used as a standard and build upon for more abstract methods such as this work.

Furthermore, the formalisms of this thesis are not complete; although every issue of framework verification is addressed, we have shown in some areas that further work is required to make the specifications and proofs entirely accurate (eg. the state of memory). We hope this thesis gives a useful overview and procedure for framework verification effort as a whole.

Appendix A

Definitions and Proofs for Examples

A.1 `Calculates` Framework Proof Steps

Following is a copy of the general description of the proof steps for the goal in Section 3.2.1.4, along with the intermediate logic (HOL) results:

1. Rewrite with the definitions from previous sections to get a goal in terms of `fold_function`:

```
∀ yy z.
  fold_function operates_class calc_ptr (state,oper_ptr,yy) (state,z) ==>
  XOR
    (∃ w.
       (z = Pos_Number_Number (Number w)) ∧
       w <= operates_class.maximum oper_ptr)
    ∃ str. z = Pos_Number_Overflow (Overflow str)
```

2. Case-split on the variable `yy`, which is either an empty list or a list of at least one item:

```
∀ h z.
  fold_function operates_class calc_ptr (state,oper_ptr,h::yy) (state,z) ==>
  XOR
    (∃ w.
       (z = Pos_Number_Number (Number w)) ∧
       w <= operates_class.maximum oper_ptr)
    ∃ str. z = Pos_Number_Overflow (Overflow str)

∀ z.
  fold_function operates_class calc_ptr (state,oper_ptr,[]) (state,z) ==>
  XOR
    (∃ w.
       (z = Pos_Number_Number (Number w)) ∧
       w <= operates_class.maximum oper_ptr)
    ∃ str. z = Pos_Number_Overflow (Overflow str)
```

(a) For the case with the empty list, rewrite with the definitions of `fold_function`, `XOR`, and `operates_obligs` to get the following:

```
(∀ oper_ptr x y z state.
   operates_class.identity oper_ptr <= operates_class.maximum oper_ptr ∧
   (operates_class.operate oper_ptr (state,x,y) (state,z) ==>
    XOR
      (∃ w.
         (z = Pos_Number_Number (Number w)) ∧
         w <= operates_class.maximum oper_ptr)
      ∃ str. z = Pos_Number_Overflow (Overflow str))) ==>
   ...
```

```
(∀ z.
  (z = Pos_Number_Number (Number (operates_class.identity oper_ptr))) ==>
  ((∃ w.
     (z = Pos_Number_Number (Number w))
     ∧
     w <= operates_class.maximum oper_ptr)
   ∧
   ¬ (∃ str. z = Pos_Number_Overflow (Overflow str)))
  ∨
  (¬ (∃ w.
       (z = Pos_Number_Number (Number w))
       ∧
       w <= operates_class.maximum oper_ptr)
   ∧
   ∃ str. z = Pos_Number_Overflow (Overflow str)))
```

Rewrite with theorems about the relationships of the classes, for example that a `Number` does not equal an `Overflow`. At that point a value must be chosen for an instance of `w`, and since this is the case of an empty list, supply `operator_class.identity` to prove this case.

(b) For the case with at least one item in the list, rewriting with the `fold_function` and then case splitting on its result yields the following two goals:

```
fold_function operates_class calc_ptr (state,oper_ptr,yy)
                                          (state,Pos_Number_Number n)
∧
pos_number_obj_case
  (λ number.
     number_obj_case
       (λ num. operates_class.operate oper_ptr (state,h,num) (state,z))
       number) (λ message. Pos_Number_Overflow message = z)
  (Pos_Number_Number n) ==>
XOR (∃ w.
      (z = Pos_Number_Number (Number w))
      ∧ (w <= operates_class.maximum oper_ptr))
    (∃ str. z = Pos_Number_Overflow (Overflow str))

fold_function operates_class calc_ptr (state,oper_ptr,yy)
                                          (state,Pos_Number_Overflow o')
∧
pos_number_obj_case
  (λ number.
     number_obj_case
       (λ num. operates_class.operate oper_ptr (state,h,num) (state,z))
       number) (λ message. Pos_Number_Overflow message = z)
  (Pos_Number_Overflow o') ==>
XOR (∃ w.
      (z = Pos_Number_Number (Number w))
      ∧ (w <= operates_class.maximum oper_ptr))
    (∃ str. z = Pos_Number_Overflow (Overflow str))
```

Each of these cases is solved by rewriting with the definitions and the properties of the `PosNumber` classes and then choosing the values for the existential variables that match existing variables.

A.2 Multiplies Proof Steps

Following is a copy of the general description of the proof steps for the goal in Section 4.1.3, along with the intermediate logic (HOL) results:

1. Rewrite with the definitions from Section 4.1.2 to get a goal showing the specific behavior expected from `mult_operate`:

```
∀ n mult_ptr m k. mult_operate mult_ptr (m,n) k = (k = m * n)
```

2. Induct on the variable `n` to get a base case and induction step:

```
∀ mult_ptr m k. mult_operate mult_ptr (m,0) k = (k = m * 0)



∀ mult_ptr m k. mult_operate mult_ptr (m,n) k = (k = m * n)
⟹
∀ mult_ptr m k. mult_operate mult_ptr (m,SUC n) k = (k = m * SUC n)
```

3. The base case is proven with the following steps:

   (a) Rewrite once with the definition of `mult_operate`:

```
∀ mult_ptr m k.
  (if 0 = 0 then
     k = 0
   else
     (let i = 0 - 1 in
      ∃ j. mult_operate mult_ptr (m,i) j ∧ (k = m + j))) =
  (k = m * 0)
```

   (b) Induct on `k` for another base case and induction step. Each of these are easily provable based on the properties of zero (with the HOL tactic `reduceLib.REDUCE_TAC`):

```
(if 0 = 0 then
   k = 0
 else
   (let i = 0 - 1 in ∃ j. mult_operate mult_ptr (m,i) j ∧ (k = m + j))) =
(k = m * 0)
⟹
(if 0 = 0 then
   SUC k = 0
 else
   (let i = 0 - 1 in
      ∃ j. mult_operate mult_ptr (m,i) j ∧ (SUC k = m + j))) =
(SUC k = m * 0)



(if 0 = 0 then
   0 = 0
 else
   (let i = 0 - 1 in ∃ j. mult_operate mult_ptr (m,i) j ∧ (0 = m + j))) =
(0 = m * 0)
```

4. The induction step for `n` is proven with the following steps:

   (a) Rewrite once with the definition of `mult_operate`:

```
∀ mult_ptr m k. mult_operate mult_ptr (m,n) k = (k = m * n)
⟹
∀ mult_ptr m k.
  (if SUC n = 0 then
     k = 0
   else
     (let i = SUC n - 1 in
      ∃ j. mult_operate mult_ptr (m,i) j ∧ (k = m + j))) =
  (k = m * SUC n)
```

(b) Simplify the phrase `SUC n - 1` to `n`, expand the `let` phrase by substituting `n` for `i` in the remainder of the goal, and eliminate the first arm of the `if` clause since `SUC n` is never 0:

```
∀ mult_ptr m k. mult_operate mult_ptr (m,n) k = (k = m * n)
⟹
∀ k.
  (∃ j. mult_operate mult_ptr (m,n) j ∧ (k = m + j)) =
  (k = m * SUC n)
```

(c) Rewrite with the hypothesis for the induction step to get the following:

```
∀ k. (∃ j. (j = m * n) ∧ (k = m + j)) = (k = m * SUC n)
```

(d) The remainder depends on properties of multiplication and division. Note that by instantiating `m * n` for `j`, the goal becomes `(k = m + m * n) = (k = m * SUC n)` which is straightforward to solve.

A.3 `Multiplies` Proof Script

Following is the full HOL proof script for the goal found in Section 4.1.3. `MULTIPLIES_CLASS_TAC` is the tactic that proves the entire theorem.

```
val MULT_SUC =
    (CONJUNCT2 o CONJUNCT2 o CONJUNCT2 o CONJUNCT2 o CONJUNCT2 o SPEC_ALL)
    arithmeticTheory.MULT_CLAUSES;
```

```
val MULTIPLIES_CLASS_MULT_CASES =
EQ_TAC
THENL [
 STRIP_TAC
 THEN FIRST_ASSUM (fn th => UNDISCH_TAC (concl th))
 THEN ASM_REWRITE_TAC []
 THEN STRIP_TAC
 THEN ASM_REWRITE_TAC []
 THEN REWRITE_TAC [MULT_SUC]
 ,
 STRIP_TAC
 THEN ASM_REWRITE_TAC []
 THEN EXISTS_TAC (Term`m * n`)
 THEN REWRITE_TAC []
 THEN REWRITE_TAC [MULT_SUC]];


val MULTIPLIES_CLASS_TAC =
REWRITE_TAC [db.theorem "-" "multiplies_obligs_def",
             db.theorem "-" "multiplier_class_def",
             db.theorem "-" "multiplies_accupds"]
THEN GEN_TAC
THEN GEN_TAC
THEN GEN_TAC
THEN SPEC_TAC (Term`m:num`, Term`m:num`)
THEN SPEC_TAC (Term`mult_ref:'a`, Term`mult_ref:'a`)
THEN SPEC_TAC (Term`n:num`, Term`n:num`)
THEN Induct
THENL [
 ONCE_REWRITE_TAC [mult_impl_def]
 THEN GEN_TAC
 THEN GEN_TAC
 THEN Induct
 THEN reduceLib.REDUCE_TAC
 ,
 ONCE_REWRITE_TAC [mult_impl_def]
 THEN GEN_TAC
 THEN GEN_TAC
 THEN REWRITE_TAC [arithmeticTheory.SUC_SUB1]
 THEN CONV_TAC (TOP_DEPTH_CONV Let_conv.let_CONV)
 THEN REWRITE_TAC [GSYM arithmeticTheory.SUC_NOT]
 THEN ASM_REWRITE_TAC []
 THEN POP_ASSUM (fn thm => ALL_TAC)
 THEN Induct
 THEN MULTIPLIES_CLASS_MULT_CASES];
```

A.4 fold_function Definition Proof

Following is the well-foundedness proof that enables the definition of the fold_function from Section 3.2.1.3:

```
val guess =
    (Term`measure((LENGTH o SND o SND o FST o SND o SND)
                  : ('operates_ptr, 'state) operates_iface_sig
                  # 'calculates_ptr
                  # ('state # 'operates_ptr # num list)
                  # ('state # pos_number_obj)
                  -> num)`);

Defn.tprove (fold_function_defn,
             (EXISTS_TAC guess
              THEN TotalDefn.TC_SIMP_TAC [] []
              THEN REWRITE_TAC [listTheory.LENGTH]
              THEN REPEAT STRIP_TAC
              THEN numLib.ARITH_TAC));
```

A.5 `Balances` Definitions and Proof

Following are the full HOL definitions, the goal, and the proof script for the `Balances` example with all the state details (from Section 4.3).

```
(*** Specifications ***)

load"bossLib";
open bossLib;


Hol_datatype‘
consumes_iface =
<|
  eat : 'consumes_ref -> ('state # num) -> ('state) -> bool;
  eaten : 'consumes_ref -> ('state) -> ('state # num) -> bool
|>
‘;

Define‘
wf_con_obj con_class con_ref state =
(! state' state'' n .
 (con_class.eaten con_ref (state') (state'', n))
 ==>
 (state'' = state'))
/\
(! state' state'' state''' token n m .
 (con_class.eat con_ref (state, token) (state'))
 /\
 (con_class.eaten con_ref (state) (state'', n))
 /\
 (con_class.eaten con_ref (state') (state''', m))
 ==>
 (m = n+1))
‘;

Define‘
consumes_obligs consumes_class =
! con_ref state .
(wf_con_obj consumes_class con_ref state)
‘;
```

```
Hol_datatype‘
consumes_factory_iface =
<|
  generate : ’consumes_factory_ref -> (’state) -> (’state # ’consumes_ref)
             -> bool
|>
‘;

Define‘
wf_con_fac_obj con_fac_class con_fac_ref state =
! consumes_class .
! con_fac_ref con_ref state state’ state’’ how_many .
(con_fac_class.generate con_fac_ref (state) (state’, con_ref))
==>
((wf_con_obj consumes_class con_ref state’)
 /\
 (consumes_class.eaten con_ref (state’) (state’’, how_many))
 /\
 (state’’ = state’)
 /\
 (how_many = 0))
‘;


Define‘
consumes_factory_obligs
 (consumes_factory_class
  :(’con_fac_ref,’state,’con_ref)consumes_factory_iface) =
! (con_fac_ref:’con_fac_ref) (state:’state) .
(wf_con_fac_obj consumes_factory_class con_fac_ref state)
‘;




Hol_datatype‘
balances_iface =
<|
  delegate : ’balances_ref -> (’state # num) -> (’state) -> bool;
  retrieve : ’balances_ref -> (’state)
              -> (’state # ’consumes_ref # ’consumes_ref) -> bool
|>
‘;
```

```
load "res_quanLib";

Define‘
wf_bal_obj bal_class bal_ref state =
! (consumes_class:('con_ref, 'state) consumes_iface)::consumes_obligs .
! con1_ref con2_ref state' .
! count1 count2 state'' state''' .
(bal_class.retrieve bal_ref (state) (state', con1_ref, con2_ref))
/\
(consumes_class.eaten con1_ref (state') (state'', count1))
/\
(consumes_class.eaten con2_ref (state') (state''', count2))
==>
(state' = state)
/\
(state'' = state')
/\
(state''' = state')
/\
((count1 = count2)
 \/
 (count1 = count2+1)
 \/
 (count1+1 = count2))
‘;


Define‘
balances_obligs balances_class =
 ! bal_ref state .
 (wf_bal_obj balances_class bal_ref state)
‘;
```

```
(*** Implementation ***)
(* state helper functions *)

Hol_datatype`
balancer_obj = Balancer of 'consumes_ref # 'consumes_ref # bool;
balancer_ref = Balancer_Ptr of num | Balancer_Null`;

Hol_datatype`
state = State of ('consumes_ref balancer_obj) list # 'other_objs`;

Define`
balancer_helper_new (State (bals, bal_facs)) cons1 cons2 cons1_last =
  let result_obj = Balancer (cons1, cons2, cons1_last)
  in (State (APPEND bals [result_obj], bal_facs),
      (Balancer_Ptr (LENGTH bals)))
`;

Define`
(balancer_helper_get_cons1_obj (Balancer (cons1, cons2, cons1_last)) =
 cons1)
`;

Define`
(balancer_helper_get_cons1 (Balancer_Ptr offset) (State (bals, bal_facs))=
  balancer_helper_get_cons1_obj (EL offset bals))
`;

Define`
(balancer_helper_get_cons2_obj (Balancer (cons1, cons2, cons1_last)) =
 cons2)
`;

Define`
(balancer_helper_get_cons2 (Balancer_Ptr offset) (State (bals, bal_facs))=
  balancer_helper_get_cons2_obj (EL offset bals))
`;

Define`
(balancer_helper_get_cons1_last_obj (Balancer (cons1, cons2, cons1_last))=
  cons1_last)
`;

Define`
(balancer_helper_get_cons1_last (Balancer_Ptr offset)
                                (State (bals, bal_facs)) =
  balancer_helper_get_cons1_last_obj (EL offset bals))
`;
```

```
load "rich_listTheory";

Define‘
(balancer_helper_set_cons1_obj
  (Balancer (cons1, cons2, cons1_last)) new_cons1 =
  Balancer (new_cons1, cons2, cons1_last))‘;

Define‘
(balancer_helper_set_cons1 (Balancer_Ptr offset)
        (State (bals, bal_facs)) new_cons1 new_state =
 ? new_balancer .
 (new_balancer = balancer_helper_set_cons1_obj(EL offset bals)new_cons1)/\
 (new_state =
   State (APPEND (FIRSTN offset bals)
                 (CONS new_balancer (BUTFIRSTN (offset+1) bals)),
          bal_facs)))‘;

Define‘
(balancer_helper_set_cons2_obj
  (Balancer (cons1, cons2, cons1_last)) new_cons2 =
  Balancer (cons1, new_cons2, cons1_last))‘;

Define‘
(balancer_helper_set_cons2 (Balancer_Ptr offset)
       (State (bals, bal_facs)) new_cons2 new_state =
 ? new_balancer .
 (new_balancer = balancer_helper_set_cons2_obj(EL offset bals)new_cons2)/\
 (new_state =
   State (APPEND (FIRSTN offset bals)
                 (CONS new_balancer (BUTFIRSTN (offset+1) bals)),
          bal_facs)))‘;

Define‘
(balancer_helper_set_cons1_last_obj
 (Balancer (cons1, cons2, cons1_last)) new_cons1_last =
  Balancer (cons1, cons2, new_cons1_last))‘;

Define‘
(balancer_helper_set_cons1_last (Balancer_Ptr offset)
       (State (bals, bal_facs)) new_cons1_last new_state =
 ? new_balancer .
 (new_balancer =
   balancer_helper_set_cons1_last_obj (EL offset bals) new_cons1_last) /\
 (new_state =
  State (APPEND (FIRSTN offset bals)
                (CONS new_balancer (BUTFIRSTN (offset+1) bals)),
         bal_facs)))‘;
```

```
(* method translations *)

Define‘
balancer_balancer (state, consumes_factory_ref) (state', balancer_ref)
  =
  (? (con_fac_class::consumes_factory_obligs) .
   ? inner_state inner_state' cons1 cons2 .
   (con_fac_class.generate consumes_factory_ref state (inner_state,cons1))
   /\
   (con_fac_class.generate consumes_factory_ref inner_state
                                                    (inner_state', cons2))
   /\
   ((state', balancer_ref)
    =
    ((balancer_helper_new inner_state' cons1 cons2 F))))‘;

Define‘
balancer_delegate (balancer_ref:'consumes_ref balancer_ref)
  (state, token) (new_state:('consumes_ref, 'other_objs) state) =
  ? cons_class::consumes_obligs .
  ? inner_cons1_last inner_state inner_state' .
  (inner_cons1_last = balancer_helper_get_cons1_last balancer_ref state)
  /\
  (if (inner_cons1_last)
  then (? inner_cons .
        (inner_cons = balancer_helper_get_cons2 balancer_ref state)
        /\
        (cons_class.eat inner_cons (state, token) inner_state))
  else (? inner_cons .
        (inner_cons = balancer_helper_get_cons1 balancer_ref state)
        /\
        (cons_class.eat inner_cons (state, token) inner_state)))
  /\
  (balancer_helper_set_cons1_last
    balancer_ref
    state
    (~(balancer_helper_get_cons1_last balancer_ref state))
    (inner_state'))
  /\
  (new_state = inner_state')‘;

Define‘
balancer_retrieve (balancer_ref:'consumes_ref balancer_ref)
  (state:('consumes_ref, 'other_objs) state)
  (new_state, cons1, cons2) =
  (cons1 = balancer_helper_get_cons1 balancer_ref state) /\
  (cons2 = balancer_helper_get_cons2 balancer_ref state) /\
  (new_state = state)‘;
```

```
(*** Goal and Proof ***)

g‘
let balancer_class =
 <|
    delegate := balancer_delgate;
    retrieve := balancer_retrieve
  |>
in
! state con_fac_ref state’ bal_ref .
  balancer_balancer (state, con_fac_ref) (state’, bal_ref)
  ==>
  (wf_bal_obj balancer_class bal_ref state’
   /\
   balances_obligs balancer_class)
‘;


load"Let_conv";
open Let_conv;
load"res_quanLib";
open res_quanLib;
load"arithLib";
val RES_QUAN_CONV =
    TRY_CONV (CHANGED_CONV (DEPTH_CONV RESQ_FORALL_CONV))
    THENC TRY_CONV (CHANGED_CONV (DEPTH_CONV RESQ_EXISTS_CONV))
    THENC Rewrite.REWRITE_CONV [res_quanTheory.RES_SELECT,
                                res_quanTheory.RES_ABSTRACT];
val RES_QUAN_TAC = CONV_TAC RES_QUAN_CONV;


val BALANCER_CLASS_TAC =
REWRITE_TAC [db.theorem"-""balancer_balancer_def"]
THEN CONV_TAC let_CONV
THEN REWRITE_TAC [db.theorem"-""balances_iface_accupds"]
THEN REWRITE_TAC [db.theorem"-""wf_bal_obj_def"]
THEN RES_QUAN_TAC
THEN REWRITE_TAC [
                  db.theorem"-""balances_iface_accupds",
                  db.theorem"-""consumes_obligs_def",
                  db.theorem"-""wf_con_obj_def",
                  db.theorem"-""consumes_factory_obligs_def",
                  db.theorem"-""wf_con_fac_obj_def",
                  db.theorem"-""balancer_retrieve_def"
                  ]
...
```

```
THEN REPEAT STRIP_TAC
THENL [
       ASM_REWRITE_TAC [],
       RES_TAC,
       RES_TAC,
       RES_TAC
       THEN ONCE_ASM_REWRITE_TAC[]
       THEN REPEAT STRIP_TAC
       THEN CONV_TAC arithLib.ARITH_CONV,
       REWRITE_TAC [
                    db.theorem"-""balances_obligs_def",
                    db.theorem"-""wf_bal_obj_def",
                    db.theorem"-""balances_iface_accupds"
                    ]
       THEN RES_QUAN_TAC
       THEN REWRITE_TAC [
                         db.theorem"-""consumes_obligs_def",
                         db.theorem"-""wf_con_obj_def",
                         db.theorem"-""balancer_retrieve_def"
                         ]
       THEN REPEAT STRIP_TAC
       THENL [
              ASM_REWRITE_TAC [],
              RES_TAC,
              RES_TAC,
              RES_TAC
              THEN ONCE_ASM_REWRITE_TAC []
              THEN CONV_TAC arithLib.ARITH_CONV
              ]
       ];

e BALANCER_CLASS_TAC;
```

A.6 EJB Proofs

A.6.1 Framework

Following is the proof script for the EJB framework verification goal (for Section 5.3).

```
load "res_quanLib";
open res_quanLib;

val RES_QUAN_CONV =
    TRY_CONV (CHANGED_CONV (DEPTH_CONV RESQ_FORALL_CONV))
    THENC TRY_CONV (CHANGED_CONV (DEPTH_CONV RESQ_EXISTS_CONV))
    THENC Rewrite.REWRITE_CONV [res_quanTheory.RES_SELECT,
                                res_quanTheory.RES_ABSTRACT];
val RES_QUAN_TAC = CONV_TAC RES_QUAN_CONV;

fun FIRST_CH [] g = NO_TAC g
  | FIRST_CH (tac::rst) g = CHANGED_TAC tac g
                            handle HOL_ERR _ => FIRST_CH rst g;
val CHECK_CH_ASSUME_TAC :thm_tactic = fn gth =>
    FIRST_CH [CONTR_TAC gth, ACCEPT_TAC gth, DISCARD_TAC gth,
              ASSUME_TAC gth];
val STRIP_CH_ASSUME_TAC = REPEAT_TCL STRIP_THM_THEN CHECK_CH_ASSUME_TAC;
fun RES_CH_TAC g =
    RES_THEN (Thm_cont.REPEAT_GTCL IMP_RES_THEN STRIP_CH_ASSUME_TAC) g
    handle HOL_ERR _ => ALL_TAC g;

REWRITE_TAC [DB.theorem "-" "EJB_server_xaction_demarc_obligs_def"]
THEN REWRITE_TAC [db.theorem "-" "ejb_server_accupds"]
THEN RES_QUAN_TAC
THEN GEN_TAC
THEN CONV_TAC (TOP_DEPTH_CONV RIGHT_IMP_FORALL_CONV)
THEN REPEAT GEN_TAC
THEN REWRITE_TAC [DB.theorem "-" "EJB_bean_xaction_obligs_def"]
THEN REWRITE_TAC [DB.theorem "-" "EJB_container_xaction_obligs_def"]
THEN RES_QUAN_TAC
THEN CONV_TAC (TOP_DEPTH_CONV LEFT_IMP_FORALL_CONV)
THEN EXISTS_TAC (Term`(xaction :transaction)`)
THEN DISCH_THEN (fn th => ASSUME_TAC
            (CONV_RULE (TOP_DEPTH_CONV RIGHT_IMP_FORALL_CONV) th))
THEN FIRST_ASSUM (fn th => UNDISCH_TAC (concl th))
THEN CONV_TAC (TOP_DEPTH_CONV LEFT_IMP_FORALL_CONV)
...
```

```
THEN MAP_EVERY EXISTS_TAC
[(Term‘(container_ptr :container_ptr)‘),
 (Term‘(bean :('a, 'b) ejb_bean)‘),
 (Term‘(bean_ptr :bean_ptr)‘),
 (Term‘(state'pre :state)‘),
 (Term‘(state'post :state)‘),
 (Term‘(state'pre_bean :state)‘),
 (Term‘(state'post_bean :state)‘),
 (Term‘(state'any_pre_bean :state)‘),
 (Term‘(state'any_pre_bean2 :state)‘),
 (Term‘(trans1 :xaction_ptr)‘),
 (Term‘(trans2 :xaction_ptr)‘),
 (Term‘(trans3 :xaction_ptr)‘),
 (Term‘(trans4 :xaction_ptr)‘),
 (Term‘(trans_attr :xaction_attr)‘),
 (Term‘(x :'a)‘),
 (Term‘(y :'b)‘)]
THEN DISCH_TAC
THEN MAP_EVERY EXISTS_TAC
[(Term‘(bean_ptr :bean_ptr)‘),
 (Term‘(state'pre_bean :state)‘),
 (Term‘(state'post_bean :state)‘),
 (Term‘(trans2 :xaction_ptr)‘),
 (Term‘(trans3 :xaction_ptr)‘),
 (Term‘(x :'a)‘),
 (Term‘(y :'b)‘)]
THEN DISCH_TAC
THEN DISCH_TAC
THEN RES_TAC
THEN ASSUM_LIST(fn thl => MAP_EVERY (fn th => UNDISCH_TAC (concl th)) thl)
THEN DISCH_TAC
THEN POP_ASSUM_LIST (fn thl => ALL_TAC)
THEN REPEAT DISCH_TAC
THEN REPEAT STRIP_TAC
THEN RES_TAC
THENL [
       ASM_REWRITE_TAC [],
       ASM_REWRITE_TAC [],
       FIRST_ASSUM (fn th => UNDISCH_TAC (concl th))
       THEN BOOL_CASES_TAC (Term‘trans1 = Xaction_Null‘)
       THEN REWRITE_TAC []
       THEN ASM_REWRITE_TAC []
       THEN STRIP_TAC
       THEN ASM_REWRITE_TAC [],
       ASM_REWRITE_TAC []
       THEN FIRST_ASSUM (fn th => UNDISCH_THEN (concl th)
                                  (fn th => REWRITE_TAC [GSYM th]))
...
```

```
       THEN ASM_REWRITE_TAC [],
       ASM_REWRITE_TAC [],
       ASM_REWRITE_TAC []
       THEN POP_ASSUM (fn th => ALL_TAC)
       THEN FIRST_ASSUM (fn th => UNDISCH_THEN (concl th)
                                  (fn th => REWRITE_TAC [GSYM th]))
       THEN ASM_REWRITE_TAC [],
       ASM_REWRITE_TAC []
       THEN FIRST_ASSUM (fn th => UNDISCH_THEN (concl th)
                                 (fn th => REWRITE_TAC [GSYM th]))
       THEN FIRST_ASSUM (fn th => UNDISCH_THEN (concl th)
                                  (fn th => REWRITE_TAC [GSYM th]))
       THEN ASM_REWRITE_TAC []
       ];
```

A.6.2 Component

Following is the component (bean) proof (for Section 5.3):

```
REWRITE_TAC [
             db.theorem "-" "EJB_bean_xaction_obligs_def",
             db.theorem "-" "bean_class_def",
             db.theorem "-" "ejb_bean_accupds",
             db.theorem "-" "a_bean_method_def"
             ]
THEN (REPEAT STRIP_TAC)
THEN (ASM_REWRITE_TAC [])
```

A.7 State Functions

Following are the elements Poetzsch-Heffter uses [PH97] to model and reason about objects and their states during program execution. A mature methodology that finishes the work begun in Chapter 4 would have to include these ideas in order to be complete.

- The following functions are analogous to elements described in in Chapter 4 of this thesis (see [PH97], pp 43-44):

  `isvoid`, `obj`, `loc`, `init`, `static`, `update`, `read`, `alloc`, `new`

- The following predicates are elements that require attention in a mature methodology (see [PH97] on the pages listed next to each):

  `alive` (44, also 21), `disjoint` (54, also 22), `dirpart` (25, also 114-5), `part` (26), `wfL` (27, also 103, 115-6, 31, 96, 126) `reach` (52-3), $reach_n$ (52-3), `oreach` (54), `x-equivalent` (55), `less-alive` (55-6), `treach` (106), `alien` (112), `extern` (113, also 124)

- The rest of the functions he uses are not necessary for this thesis but might be useful when reasoning about the Java language itself, such as in [vO01] (see [PH97] on the pages listed next to each):

  `aL` (18), `inv` (98, also 27), `typ` and `ltyp` and `dtyp` and `rtyp` (43-44)

Bibliography

[Abr96] Jean-Raymond Abrial. *The B-Book : Assigning Programs to Meanings.* Cambridge University Press, August 1996.

[AF99] J Alves-Foss, editor. *Formal Syntax and Semantics of Java.* Number 1523 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, June 1999.

[AG02] Program Analysis and Verification Group. Rapide, January 2002.

[AL98] Martin Abadi and K. Rustan M. Leino. A logic of object-oriented programs. Technical Report 161, Digital Systems Research Center, Palo Alto, CA, September 1998.

[Ala98] V S Alagar. *Specification of Software Systems.* Springer Verlag, 1998.

[All97] Robert J. Allen. *A Formal Approach to Software Architecture.* PhD thesis, Carnegie Mellon University, May 1997.

[Bac80] R. J. R. Back. Correctness preserving program refinements: proof theory and applications. *MC Tracts*, 131, 1980. Mathematisch Centrum, Amsterdam.

[BDBS99] Eerke Boiten, John Derrick, Howard Bowman, and Maarten Steen. Constructive consistency checking for partial specification in z. *Science of Computer Programming*, 35(1):29–75, September 1999.

[BDHS01] G Barthe, G Dufay, M Huisman, and S Sousa. Jakarta: a toolset for reasoning about javacard. In *E-smart*, number 2140 in Lecture Notes in Computer Science, pages 2–18. Springer-Verlag, 2001.

[BDJ+01] Gilles Barthe, Guillaume Dufay, Line Jakubiec, Bernard P. Serpette, and Simao Melo de Sousa. A formal executable semantics of the javacard platform. In *European Symposium on Programming*, pages 302–319, 2001.

[BJLW97] Annette Bunker, Michael D. Jones, Trent N. Larson, and Phillip J. Windley. Alexandria: Libraries of abstract, verified hardware modules. In *2nd Workshop on Libraries, Component Modeling, and Quality Assurance*, Toledo, Spain, April 1997.

[BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[DFH+93] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The coq proof assistant user's guide version 5.8. Technical Report 154, INRIA, May 1993.

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report 159, Compaq SRC, 1998.

[EHY99] A H Eden, Y Hirshfeld, and A Yehudai. Towards a mathematical foundation for design patterns. Technical Report 1999-004, Department of Information Technology, Uppsala University, 1999.

[EKW92] D W Embley, B D Kurtz, and S N Woodfield. *Object-Oriented Systems Analysis*. Yourdon Press, 1992.

[FF97] Matthais Felleisen and Daniel P. Friedman. *A Little Java, A Few Patterns*. MIT Press, December 1997.

[FF00] Cormac Flanagan and Stephen N. Freund. Type-based race detection for java. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 219–232, 2000.

[FKN+92] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: A framework for integrating multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering*, 2(1):31–58, March 1992.

[Flo67] R. W. Floyd. Assigning meanings to programs. In *Proceedings of the American Mathematical Society Symposia in Applied Mathematics*, volume 19, pages 19–31, 1967.

[FSJ99] Mohamed E Fayad, Douglas C Schmidt, and Ralph E Johnson, editors. *Implementing Application Frameworks: Object-Oriented Frameworks at Work*. Wiley Computer Publishing, 1999.

[GC94] G. Birtwistle B. Graham and S.-K. Chin. *new_theory 'HOL';; An Introduction to Hardware Verification in Higher Order Logic*, August 1994.

[Ge93] M. J. C. Gordon and T. F. Melham (eds.). *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[GHG+93] John V. Guttag, James J. Horning, Stephen J. Garland, Kevin D. Jones, Andros Modet, and Jeannette M. Wing. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag Texts and Monographs in Computer Science, 1993.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley Pub Co, October 1995.

[GM00] Joseph A Goguen and Grant Malcolm, editors. *Software Engineering with OBJ: Algebraic Specification in Action.* Kluwer Academic Publishers, 2000.

[GPZ94] John D Gannon, Games M Purtilo, and Marvin V Zelkowitz. *Software Specification: a comparison of formal methods.* Ablex Publishing Company, 1994.

[Gro01a] Composable Software Systems Research Group. Composable software systems, June 2001. http://www.cs.cmu.edu/afs/cs.cmu.edu/project/compose/www/.

[Gro01b] Software Composition Group. Software composition group, June 2001. http://www.iam.unibe.ch/ scg/.

[GS93] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering*

and Knowledge Engineering, pages 1–39, Singapore, 1993. World Scientific Publishing Company.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.

[HP00] Klaus Havelund and Tom Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4), April 2000.

[Jac95] Daniel Jackson. Structuring z specifications with views. *ACM Transactions on Software Engineering and Methodology*, 4(4):365–389, October 1995.

[Jon90] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990. ISBN 0-13-880733-7.

[Lea95] Ted Lewis and et al. *Object Oriented Application Frameworks*. Manning Publications, 1995.

[Lei95] K. Rustan M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, Pasadena, California, January 1995.

[Lei97] K. Rustan M. Leino. Ecstatic: An object-oriented programming language with an axiomatic semantics. In *Fourth International Workshop on Foundations of Object-Oriented Languages (FOOL 4)*, January 1997.

[LG86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.

[LSS99] K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. Technical Report #1999-002, Compaq SRC, Palo Alto, USA, 1999.

[Mel89] T. F. Melham. Abstraction mechanisms for hardware verification. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 267–291. Springer-Verlag, 1989.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1997.

[MH99] Vlada Matena and Mark Hapner. Enterprise javabeans specification, v1.1. Technical report, Sun Microsystems, December 1999.

[Nip01] Tobias Nipkow. Verified bytecode verifiers. In M. Miculan F. Honsell, editor, *Foundations of Software Science and Computation Structures (FOSSACS 2001)*, volume 2030 of *LNCS*, pages 347–363. Springer, 2001.

[Nor96] Michael Norrish. Derivation of verification rules for c from operational definitions. In Jim Grundy Joakim von Wright and John Harrison, editors, *Supplementary Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics: TPHOLs'96*, TUCS General Publications. Turku Centre for Computer Science, August 1996.

[NTea95] Oscar Nierstrasz, Dennis Tsichritzis, and et al. *Object-Oriented Software Composition.* Prentice Hall, 1995.

[ON99] David von Oheimb and Tobias Nipkow. Machine-checking the java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax*

and Semantics of Java, volume 1523 of *LNCS*, pages 119–156. Springer, 1999.

[ORR+96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in Lecture Notes In Computer Science. Springer Verlag, 1994.

[PH97] Arnd Poetzsch-Heffter. *Specification and Verification of Object-Oriented Programs*. PhD thesis, Technical University of Munich, January 1997.

[PHM99] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *European Symosium un Programming (ESOP '99)*, volume 1576, pages 162–176. Springer-Verlag, 1999.

[PSSD00] D. Park, U. Stern, J. U. Skakkebaek, and D. L. Dill. Java model checking. In *First International Workshop on Automated Program Analysis, Testing, and Verification*, 2000.

[Pus98] Cornelia Pusch. Formalizing the java virtual machine in isabelle/hol. Technical Report TUM-I9816, Institut für Informatik, Technische Universiät München, 1998.

[PvJ00] Erik Poll, Joachim van den Berg, and Bart Jacobs. Specification of the JavaCard API in JML. In *Fourth Smart Card Research and Advanced Application Conference (IFIP Cardis)*. Kluwer Academic Publishers, 2000.

[Qia97] Zhenyu Qian. A formal specification of java(tm) virtual machine instructions. Technical report, FB Informatic, September 1997.

[RJ96] Don Roberts and Ralph Johnson. Frameworks evolve to domain-specific languages. In John Vlissides, Robert Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Programs*. University of Illinois, Addison-Wesley, September 1996.

[Rog97] Gregory F. Rogers. *Framework-based software development in C++*. Prentice Hall PTR, 1997.

[SB79] William Swartout and Robert Balzer. On the inevitable intertwining of specification and implementation. *Communications of the ACM*, 25(7):438–440, July 1979.

[SG99] Joao Pedro Sousa and David Garlan. Formal modeling of the enterprise javabeans component integration framework. In Davies Wing, Woodcock, editor, *Proceedings of FM'99, World Congress on Formal Methods in the Development of Software Systems*, volume 1709 of *Lecture Notes In Computer Science*, pages 1281–1300. Springer Verlag, 1999.

[SS98] E. Sekerinski and K. Sere, editors. *Program Development by Refinement : Case Studies Using the B Method*. Formal Approaches to Computing and Information Technology. Springer Verlag, London, December 1998.

[Stu93] Object-Oriented Specification Case Studies. *Kevin Lano & Howard Haughton eds.* Prentice Hall, 1993.

[vO01] David von Oheimb. Hoare logic for java in isabelle/hol. *Concurrency and Computation: Practice and Experience: Formal Techniques for Java Programs*, 13(13):1173–1214, 2001.

[Wah98] Matthew Wahab. *Object Code Verification*. PhD thesis, Department of Computer Science, University of Warwick, Coventry, UK, December 1998.

[Win94] P. J. Windley. A theory of generic interpreters. In G. J. Milne and L. Pierre, editors, *Correct Hardware Design and Verification Methods: IFIP WG10.2 Advanced Research Working Conference: Proceedings*, number 683 in Lecture Notes in Computer Science. Springer-Verlag, 1994.

[Woo99] Bobby Woolf. Frameworks development using patterns: Developing the file reader. In Mohamed E Fayad, Douglas C Schmidt, and Ralph E Johnson, editors, *Implementing Application Frameworks: Object-Oriented Frameworks at Work*. Wiley Computer Publishing, 1999.

[Zam97] Vincent Zammet. A comparative study of coq and hol. In Elsa Gunter, editor, *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics*, New Jersey, August 1997. Springer Verlag.