An Open Framework for the Specification and Execution of a Testable Requirements Model

by

David Arnold

B.C.S., M.C.S.

A Thesis submitted to the

Faculty of Graduate Studies and Research

in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

Ottawa-Carleton Institute for Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario

April, 2009

Your file Votre référence
ISBN: 978-0-494-52057-4
Our file Notre référence
ISBN: 978-0-494-52057-4

Abstract

Bertolino [34] remarks: "The leading idea [in model-based testing] is to use models defined in software construction to drive the testing process, in particular to automatically generate the test cases." In practice, however, we observe that Model-Based Testing (MBT) typically suffers from a gap between such models and the Implementation Under Test (IUT). More precisely, a model contains a specification that must be transformed into test cases that can be executed by a specific IUT. Such transformation is seldom automated and thus the resulting executable test cases may no longer correspond to the ones obtained from the model. Furthermore, current MBT approaches generally compound this problem by not addressing the problem of test case instrumentation, in particular of the ones pertaining to scenario testing. In this work, we present a solution that eliminates both of these problems by automatically bridging from a Testable Requirements Model (TRM) to a concrete implementation.

Through the course of our research we have designed and implemented a Validation Framework (VF) supporting the specification of a TRM using a text-based contract specification language grounded in the notion of responsibilities and scenarios. The VF compiles this requirements model along with a set of bindings that bridge the gap between model and implementation. The result is a model that is directly executed against the IUT, resulting in the production of a contract evaluation report. It is the operation of this framework and its application to the MBT domain that we address in this dissertation: specifically how our VF bridges the gap between a TRM and an IUT allowing for automated validation of the implementation against the model.

Acknowledgements

I am grateful for the encouragement, enthusiasm, and inspiration provided by my thesis supervisor: Jean-Pierre Corriveau. Without your help and support, the thesis and implementation would never have left the whiteboard in your office.

Special thanks go to Alain Forget, my roommate these past five years: you served as a sounding board, helped me procrastinate, and made countless trips to Tim Hortons even though you don't even drink coffee.

I am indebted to Carleton University, the Government of Ontario, and the Government of Canada for the monetary support provided during the course of my research. I definitely would not have been able to focus on my research without their contributions.

My mother, my father, and my sister have always been a source of encouragement: they kept me focused on my work during tough times at home. I know they will never understand exactly what I do, but they will always be there to encourage me.

Finally, during the course of this endeavor I met someone who has become very special to me. My thesis has indirectly allowed me to fall in love. I love you Mary, and appreciate all of the caring, compassion, and support you have provided. This thesis is dedicated to you.

Thank you all.

Table of Contents

List of Acronyms

Acronym	Definition
ACL	Another Contract Language
AGEDIS	Automated Generation and Execution of Test Suites for Distributed Component-Based Software
AML	AGEDIS Modeling Language
API	Application Programming Interface
ASM	Abstract State Machine
ASML	Abstract State Machine Language
ASN.1	Abstract Syntax Notation One
ASP	Active Server Pages
CADP	Caesar Aldebaran Development Package
CER	Contract Evaluation Report
CIL	Contract Intermediate Language
CLP	Constraint Logic Programming
CORBA	Common Object Request Broker Architecture
DbC	Design-by-Contract
DCOM	Distributed Component Object Model
DLL	Dynamic Link Libraries
EBNF	Extended Backus-Naur Form
EFSM	Extended Finite State Machine
FSE	Foundations of Software Engineering
FSM	Finite State Machine
GFT	Graphical Format for TTCN
GRL	Goal-oriented Requirement Language
IDE	Integrated Development Environment
IDL	Interface Definition Language
IOLTS	Input Output Labeled Transition System
IR	Intermediate Representation
ISE	Interactive Software Engineering
IUT	Implementation Under Test
LTS	Labeled Transition System
MDA	Model Driven Architecture
MDD	Model Driven Development
MSC	Message Sequence Charts
MSIL	Microsoft Intermediate Language
MVC	Model View Controller
OCL	Object Constraint Language
OSI	Open Systems Interconnection
PE	Portable Executable
PHACT	Philips Automated Conformance Tester
PIXIT	Protocol Implementation Extra Information For Testing
RDK	Research Development Kit
RE	Requirements Engineering
SDK	Software Development Kit
SDL	Specification and Description Language
STL	Standard Template Library
TFT	Tabular Format for TTCN
TRM	Testable Requirements Model

Acronym	Definition
TTCN	Tree and Tabular Combined Notation
UML	Unified Modeling Language
URN	User Requirements Notation
VF	Validation Framework

List of Figures

List of Tables

1 Introduction

The current chapter will define the problem and present a brief summary of our research. We begin with an overview of the problem: addressing the gap between stakeholders and developers of a computer-based system with respect to validation. It is this gap that serves as the context and motivation for our work. The introduction will continue with an overview of our solution and provide a closer look at model-based testing and its relation to our approach. Next, we present the scope of our work. The scope outlines elements included within our framework, and where the framework supports domain-specific extension through its openness. We then define the validation methodology used to determine the correctness of the implementation and the usefulness of our approach. Finally the chapter will conclude with a summary of the contributions made by our research and provide an outline for the remainder of the dissertation.

1.1 Overview of the Problem

It is widely accepted that Requirements Engineering (RE) aims at providing a bridge between the stakeholders and the developers of a computer-based system, each having their own specific viewpoints and concerns [157]. From a practical standpoint, this bridge must be an operational one, rooted in the key notion of quality [36]. More specifically, the needs of stakeholders must be *validated* [132] against the *actual* behaviour of an Implementation Under Test (IUT).

The act of validating stakeholders' needs requires determining if the IUT satisfies the requirements provided by each stakeholder. These requirements include both functional and non-functional aspects of the software system. Functional Requirements (FRs) [171] specify tasks and activities that define the behaviour of the desired system, whereas Non-Functional Requirements (NFRs) [171] represent quality attributes or 'quality of service requirements' of the desired system. In general a FR defines *what* a system is supposed to *do*, whereas a NFR defines *how* a system is supposed to *be*. Implementation of each type of requirement by the software system is vital from the stakeholders' point of view, thus both functional and non-functional requirements must be included within the validation process.

Current approaches for validation fall into two categories: code-centric and model-centric. A code-centric approach to validation, such as Test-Driven Design (TDD) [31], uses test cases written at the implementation level to drive development. A TDD approach begins by creating a test case representing one or more requirements of the system. The test case is executed against the software system usually resulting in a failure. Code is then added to the implementation until the test case passes. The process repeats until all of the requirements have been satisfied. Such test cases do not explicitly model the needs of stakeholders and seem to be limited to a developers' viewpoint. Thus, code-centric approaches pertain more to white-box testing [36] than to validation, which treats the IUT as a black box [36]. That is, code-centric approaches are marred with implementation details making them difficult for stakeholders to understand and use.

In contrast, in a model-centric approach, such as Model-Based Testing (MBT) [34], the needs of stakeholders are captured in models from which tests are extracted to drive validation [36]. These models, either in a graphical or text-based format, present a view of the software system that is abstract and usually independent of implementation details. Thus, such models permit a better understanding, by the stakeholders, of the system being developed. Recently, there is a resurgence of research in the MBT domain, especially in the area of automated generation of test cases from abstract models. However such work largely remains theoretical: industrial adoption is low [34].

The lack of industry adoption seems due to the gap that results from generating test cases from a model. Let us elaborate: as the model resides at a level of abstraction that is above the implementation, test cases generated from the model also conceptually reside above the implementation. Thus, there is no direct link to the IUT. Typically, this link is established using one of three approaches.

The first approach is to use *glue code* to bridge the gap between the abstract test cases and an actual implementation. Such glue code requires a programmer to express the abstract test cases into code. The latter, (which includes test drivers for methods, instances, and scenarios) must be built but it may not correspond to the abstract test cases. Moreover, the glue code is implementation-specific and can only be used for a single IUT. Furthermore, the creation of such glue code is typically a manual exercise that is expensive and possibly just as error prone as development of the original IUT.

The second approach is the use of a model capturing detailed behaviour capable of generating test cases on the same level as the implementation. Adding such detailed behaviour transforms the model from one that was abstract and comprehensible by stakeholders into one that is implementation-specific and thus much less readable and understandable for stakeholders, as is also the case with code-centric approaches. In addition, models tightly coupled to specific implementation details and technology cannot be used for the generation of multiple implementations operating on different platforms, as is the goal of Model-Driven Development (MDD) [170]. Technology such as the Object Constraint Language (OCL) [188] can be used to provide such detailed behavioural constraints to models specified using the Unified Modeling Language (UML) [189]. However, the OCL is implementation-specific and often makes direct reference to variables within the IUT. Moreover, specialized compilers and code generators are required to translate OCL constraints into code-specific to each IUT [11]. In addition, the OCL is limited in the types of tests it can express, especially when it comes to non-functional aspects of a system [12].

Finally, a state-based approach can be used. State-based approaches are dominant within current MBT tools [76] and often rely on global states that are problematic with respect to scalability and traceability between model and IUT. To generate tests the state machine is traversed and code is generated. The generated code is either too abstract to be used directly (i.e., the state machine does not contain implementation-specific behavioural details), in which case a programmer must still provide glue code;

or the model is tightly-coupled to a specific implementation technology and suffers from the drawbacks discussed in the previous paragraph.

Furthermore, in his explanation of the very limited adoption of his industrial-strength state-based model-testing tools at Microsoft, Grieskamp [76] proposes several possible reasons for this situation. In particular, he remarks that developers and testers alike significantly prefer the intuitive nature of scenarios [168] to the semantics underlying more formal (not only logic-based and set theoretic, but also state-based) approaches.

Indeed, the use of scenarios (i.e., UML use cases [189]) as a method for requirements capture is commonly accepted [201]: several software design paradigms are grounded in the notions of *responsibilities* and *scenarios* [109]. This is the case, in particular, for object-oriented reactive systems. Thus, not surprisingly, scenarios (such as Use Case Maps (UCMs) [6] and Binder's extended Use Cases [36]) have also recently been used (sometimes in combination with state machines [167]) for MBT.

Even in such proposals for scenario-driven testing, as in the vast majority of existing work on MBT, there is still a considerable gap between the test cases generated from models, and their corresponding executable counterparts. More precisely, the generated test cases (which take the form of paths through scenarios) are specifications that must be transformed into test cases executable in a specific IUT. Such transformation is seldom automated and thus the resulting executable test cases may not correspond to the ones obtained from the models. Furthermore, current MBT

approaches generally compound this problem by not addressing the executability of such paths. That is, the model generates paths without any regard of how the validation of such paths is to be carried out with respect to the IUT.

To summarize, the problem we are addressing here is the lack of a bridge between the requirements of a system and a candidate implementation of this system for automated validation of the IUT against the system requirements. Such a bridge requires a requirements model that can be expressed at a level of abstraction that is stakeholder-friendly and implementation independent, yet executable with respect to the validation of concrete implementations.

1.2 Overview of Our Solution

We begin by looking at how we can capture the requirements of a system into a unified model representing both functional and non-functional requirements. System requirements are derived from two different sources. The first source is a predefined specification. Examples of such specifications include programming language compilers [98, 99], wireless network specifications [182, 183], and Application Programming Interface (API) specifications [187, 200]. The second source is the systems' stakeholders, who are translating a user need or marketing goal into a software product [87]. Regardless of where the system requirements come from, they need to be captured in a Requirements Model (RM).

1.2.1 Capturing a Requirements Model

The use of scenarios as a method for requirements capture is not new. Several software design paradigms focus on the use of scenarios for the elicitation of

requirements [5, 29, 109]. One of the more well known methods for the capture of scenarios is through UML *use cases* [189]. Use cases allow for the capture of functional requirements. They describe the interaction between a primary actor and the system itself. Such interaction is captured as a sequence of simple steps. Use case steps are specified using natural language, and are not directly executable [37].

Scenarios alone do not represent a complete requirements model. Scenarios are used to specify functional requirements; they do not readily specify non-functional requirements. In addition, the ambiguity associated with the use of natural language does not aid in the construction of a requirements model that can be automatically validated against an IUT.

Our work seeks to capture functional requirements specified in the form of scenarios, along with non-functional requirements, as well as system constraints not easily specified through the use of scenarios, to create an executable *Testable Requirements Model* (TRM) that is used for the automated validation of a candidate IUT. Let us elaborate.

1.2.2 Testable Models

In order to achieve automated validation of a requirements model, we need a model that is *testable* [36] as defined by Binder. Binder's definition specifies the following requirements for the creation of a testable model [36]:

- It is a complete and accurate reflection of the kind of implementations to be tested. The model must represent all requirements to be exercised.

- It abstracts detail that would make the cost of testing prohibitive.
- It preserves detail that is essential for revealing faults and demonstrating conformance.
- It represents all events, so that a tester can generate these events.
- It represents all actions, so that a tester can determine whether a required action has been produced.
- It represents state so that we have an executable means to determine what state has (or has not) been achieved.

Clearly from the first requirement, the model must capture both functional and non-functional requirements. However there does not appear to be a general agreement on the semantics used to create such a TRM. For example, in the case of the software radio specification [190], UML class and sequence diagrams are used to capture the model. Other system specifications are captured through the aforementioned use cases. Some researchers have argued for notations and semantics beyond the UML. Examples of such research include Yu's work on i* [205] and the Goal-oriented Requirement Language (GRL) [152], and Amyot's work on the User Requirements Notation (URN) [6] which has recently been standardized by the International Telecommunication Union (ITU) [106]. Regardless of notation or semantics, there does not appear to be a single approach for the creation of a general purpose TRM. In specialized domains where validation is actively practiced, it appears that the specialized domain leads to a specialized approach for validation. Examples of

such specialized approaches will be presented in Chapter 2. In contrast, we wish to create a TRM that is general in nature and is not tied to a specific domain.

It is our position that, regardless of the methods used to specify and evaluate a TRM, the model that we use for automated validation must be *testable*. By testable, we mean that the validation approach must provide a systematic way to generate checks from the model that can be automatically executed against a candidate IUT. As Binder emphasizes there is no testing without a TRM [36]. Furthermore, from his viewpoint, few of the models used in object-oriented and component-based development are testable. Our work first adds to Binder's requirements for a TRM, then defines a syntax and semantics for such a model, and finally designs and implements an underlying *Validation Framework* (VF) allowing for the automated execution of a TRM against a candidate IUT without the need for glue code or other type of bridge. The task of determining if the semantics resulting from creation of such a TRM are complete with respect to accuracy and the correct level of abstraction are beyond the scope of this work. We are only providing a syntax and a semantics for such a TRM within our VF.

1.2.3 Candidate Implementations

The premise of our approach is that the TRM is decoupled from any particular implementation so that a single TRM can be executed against *several* candidate implementations. By candidate, we mean one that meets all requirements defined by the TRM. This is in sharp contrast with code-centric approaches in which implementation code is the starting point for obtaining tests. Our research supports applying a single TRM to several candidate implementations one at a time, without

requiring modification to the TRM. The results generated by executing the TRM against each candidate IUT can be used for IUT comparison. As an example, consider several vendors applying for certification against a standard. The standard would be represented by a single TRM that would be validated against each of the candidate implementations.

Depending on the type of validation being performed, implementation source code may not be available. That is, a developer, who could be a different entity than the one doing validation, may not divulge the implementation source code to protect her intellectual property. Similarly, in the case where offshore software outsourcing is used, the source code may not be available, and validation could be used to determine completion of the outsourced component [52]. To aid with such a constraint, the TRM should be executed against a binary candidate IUT. That is, our validation process does not require source code of the implementation to operate. Figure 1 provides a graphical representation of how a TRM is derived and applied to several candidate implementations.

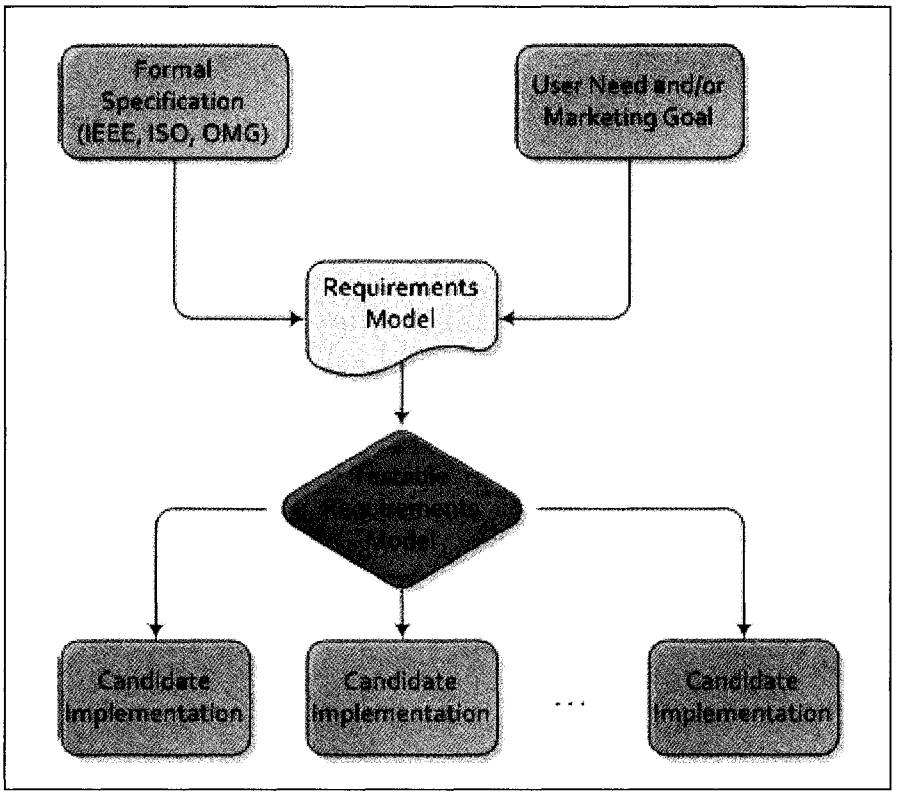

Figure 1 - Illustration of Validation

As illustrated in Figure 1 our vision is to have a requirements model that is more than just a requirements specification, but rather is testable. With such a TRM we strive for the creation of a VF that accepts the TRM as input along with a candidate IUT. The framework is then able to infer a set of mappings between elements of the TRM and the IUT to create a *binding*. It is this binding that eliminates the need for the manual specification of glue code. Thus, by simply clicking a button our VF is able to execute the candidate IUT against the TRM to produce a validation report. When the IUT is executed by our framework, the IUT is running test cases. These test cases are defined and implemented by the behaviour of the IUT. That is, our work does not focus on test case generation (more on this in the future work section in Chapter 5). Our contribution

is such a TRM as well as the implementation of an underlying VF supporting this vision. We will now provide a closer look at our TRM and define its requirements.

1.3 Testable Requirements Models

In this section we examine the characteristics of a Testable Requirements Model (TRM) used for validation. We begin by defining our requirements for such a model, then how scenarios and responsibilities can be used to construct such a model, and finally how we can augment our model for the capture of non-functional requirements.

1.3.1 Requirements for a Testable Model

A large body of work exists in the domain of requirements engineering [118]. In order for the requirements of a system to be testable, they must be captured in such a way as to yield a testable model, that is, a model from which *tests*[1] can be extracted in a systematic way. These tests form the *contract* of the system, a contract to which each realization of the requirements model must adhere. Furthermore, such tests can be viewed as queries on the structure and behaviour of a system. For the purposes of our research we will refer to the structural queries as static checks, and behavioural queries as dynamic checks. If these checks cannot be extracted in a systematic way from the requirements model, then we will necessarily suffer from the so-called *correspondence problem*: We cannot rule out that a semantic gap (that is a lack of *traceability* [36, 59]) may exist between the requirements model and the actual checks used during validation. That is, if the checks are not traceable back to requirements, then we cannot be certain that the checks *do* address the system requirements.

[1] Tests are used to determine the valid and invalid behaviour of an IUT. That is, the TRM specifies tests (such as scenarios), that are embedded in the IUT by the VF as checks which constitute the realization of these tests.

In our work, we adopt a strong interpretation of testability: the automated generation of checks from a system requirements model. Our motivation is obvious: if the production of checks is automated from a requirements model, then such checks are expected to systematically correspond to this model. In other words, the automated generation of tests from a requirements model considerably reduces the risk of a lack of correspondence between the requirements and the checks. Such a claim is very similar to the position that automatic generation of code from models, which lies at the heart of a model-driven approach [58] to software development, reduces the possibility of a lack of correspondence between models and implementation. It should be noted however that the correspondence problem never completely disappears unless the correctness of the generation process can be demonstrated.

As previously stated, we also wish to have a testable model that is *decoupled* from any particular implementation so that the contract of a system can be tested against *several* candidate implementations.

In the context of validation, demanding that the production of a TRM from a requirements model be automated is not sufficient: if such a generated TRM remains disconnected from an IUT, then we still have a *traceability* problem. That is, it is still possible that the *actual* checks to be run against an IUT do not correspond to the TRM produced from the requirements model. Consequently, our VF must support an even stronger interpretation of testability, one that insists that the TRM obtained from the

requirements model be *executable*. Such an executable model has two important repercussions:

First, the automated generation of executable checks from a TRM provides, in our opinion, a mechanism for the *objective* assessment of validation of an IUT against the TRM. Objectivity here stems directly from the ability to execute the generated checks. That is, the success or failure of a check is not a rhetorical exercise open to debate. Instead, it consists in the outcome of the execution of a check against a candidate implementation. When such validation is at the basis of a business relationship, then such objectivity is essential.

Secondly, automated generation of executable checks from a TRM entails the creation of a VF that is capable of inputting a requirements model, generating executable checks from it, and then executing such checks on one or more candidate IUTs. The postulate of our work developing such a VF is that *scenarios* [168] constitute the semantic foundation for the requirements modeling of a system [109, 168, 197]. We will now briefly discuss scenarios as they apply to our research.

1.3.2 Scenarios and Responsibilities

Scenarios are a method for requirements capture that places emphasis on system flow and business rules [109]. Scenarios are conceptualized as *temporal flows of responsibilities* [44]. Each responsibility represents a simple action or task, such as the saving of a file, or the firing of an event. Intuitively, a responsibility is either bound to a method (i.e., procedure) within an implementation, or the responsibility is to be

decomposed into a sub-grammar of responsibilities. Complementing the grammar of responsibilities, or scenario execution grammar, is a set of Design-by-Contract (DbC) [130] elements. DbC elements are typically used to express constraints on the state of the IUT before and after the execution of a scenario or responsibility: Preconditions specify constraints on the state of the IUT before the scenario can be executed. Post-conditions specify constraints on the IUT's state following a successful scenario execution.

When a scenario is executed by an IUT, the specified grammar of responsibilities must hold. That is, the responsibilities that compose the scenario must be executed in such an order that satisfies the grammar. If the scenario cannot be executed, or actions that are not defined by the scenario are executed, then the IUT does not match the TRM.

Even simple software systems are composed of multiple scenarios. As such, executing our TRM must include not only the execution of individual scenarios, but also the execution of multiple, possibly interleaving, scenarios. Such scenario integration requires specialized constructs and operators to support temporal ordering, concurrency, and distribution. Theoretical work on such operators exists in SCENT by Ryser and Glinz [167]. Our VF operationalizes portions of their work.

Scenarios aim to capture *paths* of execution and each path constitutes one specific *context* of execution. Each path through a scenario would explicitly define an exact sequence of responsibilities. The pre- and post-conditions mentioned earlier would be

associated with the individual responsibilities of a path. In order for the scenarios to compose an executable TRM, the scenarios themselves must be executable. That is, execution of the IUT must be observed in such a way that the execution (or non-execution) of a scenario can be deterministically established. In order for such execution to take place:

1. Clearly, the responsibilities would have to be eventually bound to actual methods since methods, not responsibilities, are executable.
2. A *path generation* and *selection algorithm* would be required to generate, out of each originating scenario, a set of paths to use, and then select which ones to execute. This task involves selection because it is not usually the case that we want to execute all the possible paths contained within a scenario. In fact, typically, applying the technique of *equivalence partitioning* [36], we will want to avoid selecting paths that, from a validation viewpoint, are equivalent.
3. After eventually binding responsibilities to actual methods, a path will consist of a specific sequence of actual method calls. In order for these methods to be executable, each actual method will need to be supplied with the parameters it requires. In particular, each method will need to obtain its receiver, that is, the instance on which it executes. We refer to this as *path instantiation*. Typically, such a task is greatly complicated by the fact that the receiver of each method may have to be put into a specific state [36].

In a simple example, all of the method calls pertaining to the responsibilities of a scenario would be performed on the same instance. Thus, all of the checks associated with these methods would pertain to the same instance. Generally, however a path will involve methods pertaining to several instances of different classes. Consequently, the task of path instantiation or sensitization will be obviously more difficult than in the case of a single instance. For example, consider customers waiting in distinct queues at a grocery store. Furthermore, assume such queues are monitored by a manager who decides when to open or close them. Here, there is a multitude of instances, some belonging to the same class (i.e., customer queues), some being singletons (i.e., the manager). As there may be different types of queues (i.e., *express* versus *normal*), it may also be necessary to associate a number of items-to-purchase with each customer. Does the manager contain a set of queues or is it each queue that has a pointer to this manager? Does each customer need to contain a reference to the queue that contains it? The point is that path instantiation is intimately tied to an actual implementation.

We also remark that once responsibilities are bound to actual methods, a path comes to constitute in itself a dynamic check: it is an expected sequence of method calls to be matched against the actual behaviour of the system. Consequently, a VF must be able to perform such matches. However, intuitively, the feasibility of such matches becomes increasingly difficult as the semantic richness of the formalism used to capture scenarios increases. For example, the use of an *abort* in a use case map [197], or of a *coregion* in a message sequence chart [129] offers significant semantic advantages with

respect to modeling, but also considerably complicates the task of matching the scenario in which such operators appear in the actual sequence of method calls.

Understanding how complex path instantiation and path matching can be leads to a fundamental issue: how much *coverage* [36] of the testable model is required for a candidate IUT to be validated against it. We will now briefly elaborate on coverage.

In practice, it is commonly accepted that testing may be an endless task [36]. In particular, a scenario-based approach to testing generally suffers from having a path generalization/selection algorithm that possibly generates an intractable number of paths. Also both path instantiation and path matching may be extremely difficult to achieve depending on the complexity of the semantics used in specifying the scenarios. Consequently, it is unrealistic to demand that all possible paths from all scenarios be selected, instantiated, tested, and matched [36].

Returning to Binder's definition of a testable model, we notice that such a model must be *complete* [36]. That is, it must document the handling of both valid and invalid paths. We stress that the complexity of path selection and of path instantiation clearly increases when one does not limit these tasks to only valid paths. We emphasize again that such path instantiation is intimately linked to an actual implementation. This observation leads to a fundamental postulate of our work: assessing how much work is involved in the instantiation of a set of selected paths is best performed by the developer and/or tester of the implementation.

As our framework operates using a binary IUT, the tasks of path selection, equivalence partitioning, and path instantiation will remain outside the scope of our research. That is, our framework will execute a Testable Requirements Model (TRM) against a candidate Implementation Under Test (IUT) whose path of execution is determined by the behaviour contained within the IUT. It is up to the developer of the IUT to determine the paths that are to be tested, how to instantiate those paths, and to decide on the required coverage level. Automatic path selection, equivalence partitioning, and path instantiation is left for future work.

1.3.3 Metric Capture and Analysis

Static checks, dynamic checks, and scenarios provide a mechanism for testing functional elements of a system. Non-functional elements, such as performance, security, and usability are difficult to capture using scenarios. As the requirements for a system include both functional and non-functional elements, we argue that a testable model must contain semantics for both functional and non-functional requirements in order to be *semantically complete*. Non-functional requirements can be validated by gathering metrics during the execution of scenarios. Such metrics include a broad range of values from performance timers to counting the number of mouse clicks (for usability analysis). Once a metric value has been gathered, it needs to be analyzed, and ultimately, trade-off analysis must take place [113]. Metric analysis is an extremely domain-specific activity [202]. To support multiple domains within our VF, we argue that our framework must be *open* (see Section 4.7). Such openness will allow for not only domain-specific metric analysis, but also specialized static and dynamic checks.

1.3.4 Discussion

In this section, we have argued that validation requires a Testable Requirements Model (TRM) capable of automatically generating executable checks and supporting the following:

- Capture of functional and non-functional requirements.
- Testability of the requirements model.
- Executability of checks.
- Semantics rooted in the notions of responsibilities and scenarios.
- Abstraction of the TRM over several candidate implementations.

Current approaches to validation [51] typically do not offer a TRM with the above characteristics. For this reason, we have created an *open* VF for the *specification* and *execution* of such a testable model for the purpose of validating a candidate IUT against the TRM. The following section will provide an overview of our solution.

1.4 Scope of Our Work

Before we begin to define the features and operation of our Validation Framework (VF), it is important to note where our framework begins and ends with respect to scope. Figure 2 provides a graphical representation of the elements contained within the scope of our research (green), and the elements that lie outside (grey).

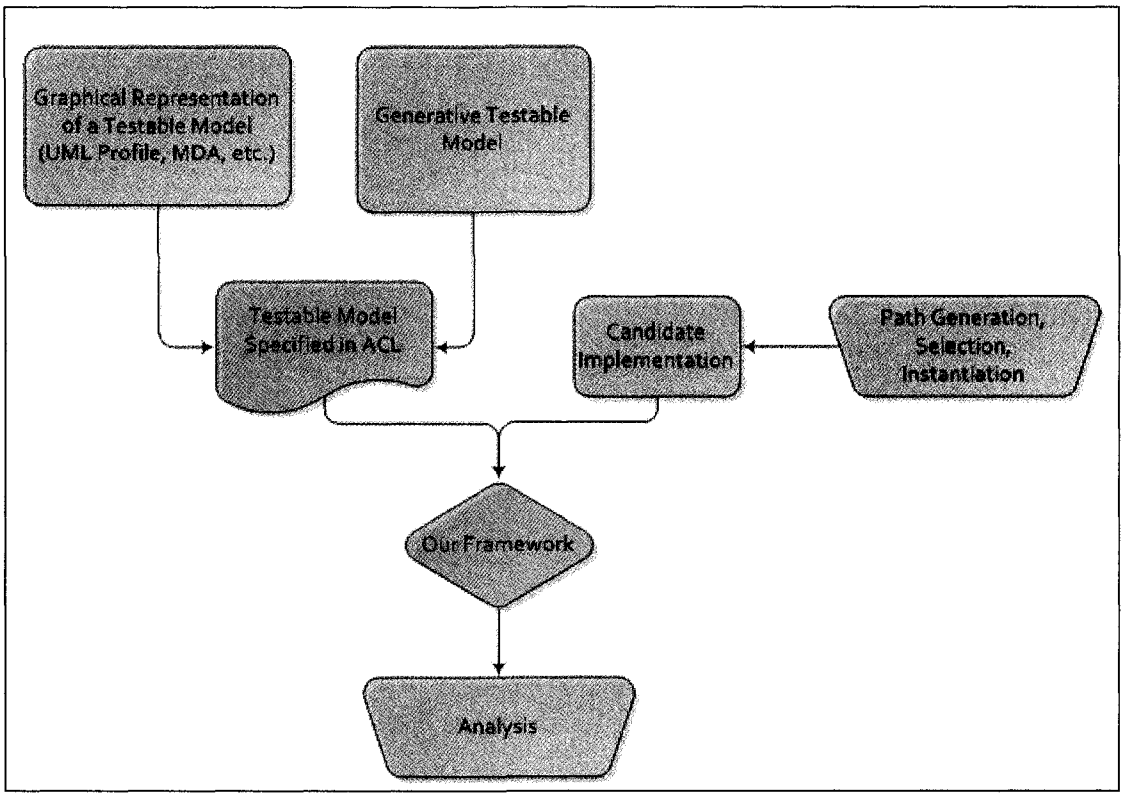

Figure 2 - Validation Framework Scope

Our VF operates on a TRM that is specified using a general purpose high-level contract language called Another Contract Language (ACL) [8] that we have created. The ACL will be presented in Chapter 3. For now, the ACL can be viewed as a text-based specification language. For the purposes of our research the ACL is directly specified by the user, however it could also be generated using one of two distinct methods.

The first method is through the use of a graphical notation that either directly represents, or is able to generate the Testable Requirements Model (TRM). As shown in Figure 2 such graphical notation lies outside the scope of our work, however additions to existing notations used to model systems are, in our opinion, feasible. An example would be the creation of a UML profile [189] that contains a graphical representation of

the constructs found within the ACL. Such a model would then be compiled, under the context of the Model Driven Architecture (MDA) [58] paradigm to generate ACL code.

The second method is modeled after work done by Czarnecki and Eisenecker in the domain of generative programming [57] where complete software systems are generated from a set of reusable components that cover an entire domain. That is, a family of systems is modeled, and then a single member of the family is generated from the available components. Applying the generative programming paradigm to testable models, we envision a set of reusable TRM components that can be assembled to build a specific TRM. That is, each of the reusable components are used in the construction of an implementation also has a reusable TRM component that can be assembled to create a TRM corresponding to the generated implementation. In order to aid in the construction and use of generative TRMs, the ACL supports generic contracts, an example of which is presented in Chapter 3.

Along with the specification of a TRM, our framework also requires at least one candidate IUT against which the TRM will be executed. The construction of candidate IUTs is outside the scope of our work. As previously stated, the VF operates on a binary candidate IUT that is able to execute the *requested scenarios*. By requested scenarios we refer to the execution paths that were selected by the tester. The tasks of path generation, selection, and instantiation lie beyond the scope of our work. Our VF operates on an actual execution path and behaviour generated by the execution of the provided candidate IUT.

Once execution of a TRM against a candidate IUT is complete, our VF produces an evaluation report known as the Contract Evaluation Report (CER). The CER indicates where the candidate IUT matches the TRM, and where any deviations from the TRM were observed. The VF presents the report in a graphical notation (see Chapter 4). Several quality control and analysis methods could be used to analyze the generated CER and apply their findings to the software development process, or calculate information important to management and other stakeholders. Such methods lie beyond the scope of our work.

Now that we have defined the boundaries of our VF, we present the features and operation of our framework. The VF provides an open architecture for the specification and execution of a TRM against a candidate IUT. The TRM specifies static checks, dynamic checks, responsibilities, scenarios, metric captures, and metric evaluators. The VF has been integrated into Microsoft's Visual Studio [137] for ease of use and integration with existing development processes (which, as emphasized by Grieskamp [76], improves adoption). The following subsections provide a brief overview of our VF; details are presented in Chapter 4.

1.4.1 Validation Framework Input

The VF operates on three input elements. The first element is the Testable Requirements Model (TRM). Recall that the TRM captures a set of contracts that are executed against a candidate IUT for validation. We have created a contract language Software Development Kit (SDK) [13] allowing for the creation of domain-specific contract specification languages. The use of multiple domain-specific contract

specification languages creates the ability to define a TRM that is syntactically similar to the problem domain. For example, a TRM for an accounting system would require different constructs from one modeling a game. For the purposes of our research we have defined the ACL [8]: a high-level general purpose contract language. The ACL is closely tied to requirements by defining constructs for the representation of goals, beliefs [152], scenarios [168], and several lower-level constructs, such as pre and post-conditions [130]. Additional domain-specific constructs can also be added to the ACL, via modules known as plug-ins. Plug-ins will be discussed in the next section.

The second input element is the candidate IUT to which the TRM will be executed against. The VF supports testing multiple IUTs against a single TRM. The framework accepts implementations in binary form. That is, source code for the IUT is not required. Using Microsoft's Phoenix Research Development Kit (RDK) [144], the framework is able to open .NET managed executables, C++ binaries that contain debugging information, and Active Server Page (ASP) .NET web applications. The debugging information is used to resolve the original symbol names during the binding process. Details regarding the Phoenix RDK are presented in Chapter 4.

Bindings represent the third and final input element required by the VF. Before a TRM can be executed the types, responsibilities, and observability [36] requirements used by the TRM must be bound to concrete implementation artifacts located within the IUT. Phoenix is used to open the binary IUT for the purposes of obtaining a structural representation. Our binding tool, which is part of the VF, uses this structural

representation to map elements from the TRM to types and methods defined within the candidate IUT. Such a binding is crucial for two reasons. First, it allows the contract to be independent of implementation details, as specific type and method names used with the candidate IUT *do not* have to exist within the TRM. Second, binding allows several candidate IUTs to be executed against a single TRM.

As will be illustrated in Chapter 3, and discussed in detail in Chapter 4, our binding tool is able to automatically infer most of the bindings required between a TRM and an IUT. That is, a set of bindings can be automatically generated by the VF, eliminating the need for manual specification.

1.4.2 Plug-ins

In addition to the openness provided by the contract language SDK, plug-ins allow for the inclusion of user-specified static checks, dynamic checks, and metric evaluators into our VF. Plug-ins are analogous to C++'s Standard Template Library (STL) [172] or plug-ins contained in Adobe's Photoshop [2]. Plug-ins can be invoked by any contract specification language, including the ACL, to provide additional framework functionality. The creation of plug-ins is targeted toward check developers, rather than the author of the testable requirements model. We have created specialized plug-in SDKs to aid in the creation of static checks [18], dynamic checks [15], and metric evaluators [16]. Examples illustrating the creation of each type of check and evaluator are provided in Chapter 4. The following subsections will define each of the three plug-in types.

1.4.2.1 *Static Checks*

Static checks perform a check on the IUT that can be accomplished without execution. Examples of static checks include: tests on inheritance depth, and the correct use of structural design patterns [72]. Static checks are invoked by code placed in the *structure* section of a contract. The various sections that compose a contract will be illustrated in Chapter 3. A static check can be viewed as an operation. Each check has a return type and may accept a fixed number of parameters. All static checks are guaranteed to be side-effect free.

1.4.2.2 *Dynamic Checks*

A dynamic check is used to perform a check on the IUT during execution. That is, a dynamic check can only be evaluated while the IUT is being executed. Examples of dynamic checks include: testing the value of a variable at a given point, ensuring a given state exists within an object, and validating data sent between two different objects. Dynamic checks are invoked by placing code in the *observability, invariant, responsibility,* or *scenario* sections of a contract. For details see Chapter 3. As with static checks, dynamic checks can be viewed as an operation with a return type and parameter set. The execution of a dynamic check is also guaranteed to be side-effect free.

1.4.2.3 *Metric Evaluators*

Metric evaluators are used to analyze and report on the metrics gathered while the candidate IUT was executing. Metric gathering is performed by the Validation Framework (VF). Once metric gathering is complete and the IUT has concluded

execution, the metric evaluators are invoked. Examples of a metric evaluator include: performance, space, and network use analysis. Metric evaluators can only be invoked by placing code in the *reports* section of a contract (see Chapter 3). Metric evaluators are side-effect free.

1.4.2.4 Discussion

Through the use of domain-specific contract specification languages, static check, dynamic check, and metric evaluator plug-ins our VF supports a high degree of openness. Such openness allows for not only domain-specific constructs but also allows for static testing, dynamic testing, and metric evaluation researchers to contribute without having to create a specialized method/process for the integration of their work. They can simply create a corresponding plug-in using the appropriate SDK. Over time we hope to have a rich set of plug-ins to accompany our framework.

1.4.3 Compilation

Once the TRM has been specified and bound to a candidate IUT, the TRM is compiled into an intermediate language known as the Contract Intermediate Language (CIL) [14]. The CIL is a low-level language that is accepted by our VF runtime [23]. The purpose of the CIL is to allow for multiple domain-specific contract languages to be executed by a common runtime. Upon a successful compilation, all elements of the TRM have been bound to IUT artifacts and any required plug-ins have been located and initialized. The result of such a compilation is a single CIL file that contains all information required to execute the TRM against a candidate IUT. Details regarding the compilation process and the CIL can be found in Chapter 4.

1.4.4 Execution

Execution of a Testable Requirements Model (TRM) begins with a structural analysis of the candidate Implementation Under Test (IUT), and with execution of any static checks. Following execution of the static checks, the IUT is executed by the VF. The VF is able to track and record the execution paths generated by the IUT, as well as execute any dynamic checks, and gather metrics indicated by the TRM. The execution paths are used to determine if each scenario execution matches the grammar of responsibilities defined within the TRM. Next, metric evaluators are used to analyze and interpret any metric data that was gathered during execution of the IUT. All of the results generated from execution of the TRM against the candidate IUT are written to a Contract Evaluation Report (CER). Generation of the CER completes the process of executing a TRM against a candidate IUT.

1.4.5 Summary

We have implemented a framework that provides an open architecture for the specification of TRMs through the use of contracts. Such contracts are grounded at the system requirements level. Elements defined within a contract are bound to structural elements found within a binary candidate IUT. The majority of such bindings can be automatically inferred. The VF executes static checks, and then executes the IUT so that dynamic checks and the execution of scenarios can be validated. Once execution of the candidate IUT is complete metrics are analyzed using metric evaluators. The result of executing a TRM against a candidate IUT is a report indicating the outcome of static checks, dynamic checks, scenario execution, and metric evaluation. The report indicates

if the candidate IUT has been validated against the TRM, from both functional and non-functional viewpoints. Details regarding the execution process employed by our VF are presented in Chapter 4. We will now discuss the methodology used in validating our approach.

1.5 Validation of our Solution

We have used several methods to validate our research. Both high-level approaches to validate the entirety of the Validation Framework (VF), and low-level approaches for validating both the Another Contract Language (ACL) compiler and the Contract Intermediate Language (CIL) compiler were used. The following subsections present each of our validation approaches. Details regarding each of the validation approaches can be found at the end of both Chapters 3 and 4.

1.5.1 Another Contract Language Compiler Validation

As will be illustrated in Chapter 3, the ACL is a high-level general purpose TRM specification language. The ACL contains several new constructs that are not present in the literature we have reviewed on testable models (see Chapter 2). As such, we have validated the entirety of the ACL by creating small ACL code snippets to verify the behaviour of each ACL construct. In addition, for each compile-time error and warning message that the ACL compiler is able to generate, we have created a code snippet to verify that the correct compiler error or warning has been generated. Validation of the ACL compiler's code generation abilities was performed by using the resultant CIL instructions as input to the CIL compiler.

1.5.2 Contract Intermediate Language Compiler Validation

The CIL is a low-level stack-based language that provides a level of abstraction between the high-level contract languages, such as the ACL, and the VF runtime. The CIL compiler was validated using a similar methodology as the ACL compiler. For each compile-time error and warning message that the CIL compiler is able to generate, we have created a code snippet to verify that the correct compiler error or warning message has been generated. In addition, as the CIL compiler does not generate any output per se, but rather serves as input to the VF runtime, the code generation aspect of the CIL compiler was validated by checking the Contract Evaluation Report (CER) to ensure that the correct behaviour within the VF runtime occurred. In addition to the low-level validation performed on our two compilers, we have also created five case studies that not only validate the VF as a whole but also serve to validate the expressive power of the ACL.

1.5.3 Validation through Case Studies

During the course of our research and implementation activities we have designed and implemented five case studies. The case studies vary in complexity and are used to validate the entirety of our research. Each case study consists of a problem description, a corresponding TRM, and a candidate IUT that can be executed against the TRM to generate a CER. The first case study models a simple container and was designed as a learning tool for the ACL and our VF [19]. The second case study presents a more advanced container illustrating concepts such as inheritance, responsibility extension, and generics. It is this advanced container that is the basis for the example used in

Chapter 3. Next, a web-based case study was created to validate and demonstrate the use of ASP.NET web applications within our VF [21]. The web case study models a system where a user logs into a web page to access protected content. The remaining two case studies serve as larger scale examples modeling 'complete' systems.

The first models a grocery store and was selected because it requires little domain knowledge from the reader [20]. A grocery store can be viewed as a set of customers who each, enter the store, and select one or more items for purchase. The example includes a complete Testable Requirements Model (TRM) illustrating ACL concepts such as inheritance, and contract generics. The provided case study includes a complete listing and discussion of the TRM, a candidate IUT, and a set of bindings between the two [20].

The second larger scale example is one modeled after a physical university [22]. The university case study represents a university creating a set of courses each term. Once course creation is completed, students register in courses. Following registration the term begins. During a term, students must complete courses by doing assignments, midterms and final exams. After the term ends, each course reports final grades to the university, and the university decides if a given student is allowed to continue with his/her education. The university case study is the most complex of our five case studies and includes a complete TRM, candidate IUT, and a set of bindings [22].

Some details regarding each of the abovementioned case studies for the validation of our approach can be found in Chapter 3. The five case studies allowed us to validate

our vision of using a TRM decoupled from any implementation technology and a set of bindings to automatically generate and execute a set of executable checks that are run against a candidate IUT. The lower-level validation activities have allowed us to ensure that our VF is usable and operates as intended. We will now present the contributions of our research.

1.6 Contributions

In addition to providing the foundation for other research within our research group (see Chapter 5), our TRM and supporting VF contribute in the area of requirements engineering and validation by:

- Proposing a new set of requirements for a requirements model that supports 'operational validation.' This set is the first to include all of the following:
    - Capture of functional and non-functional requirements.
    - Testability of the requirements model.
    - Executability of checks generated from this testable model.
    - Semantics rooted in the notions of responsibilities and scenarios.
    - Abstraction of the testable model over several possible implementations.
    - Openness to support specific static checks, dynamic checks, and metric evaluators.
- Defining a TRM that satisfies these requirements (the ACL).
- Providing an open VF supporting the specification and execution of the TRM.

1.7 Organization of the Thesis

The remainder of this document will provide the necessary background for our research, followed by a detailed walkthrough of our Validation Framework (VF) using one of our five case studies; a comprehensive account of the VF's operation then follows. Chapter 2 includes background information on Model-Based Testing (MBT) as well as a discussion of state-of-the-art in our research domain. The chapter continues with a look at contract-based testing and how it has been implemented in Eiffel [64] and other languages. Chapter 3 uses our advanced container case study to introduce the ACL [8] and provide a walkthrough of how the VF executes the given case study. A brief discussion of the other four case studies is also provided. Chapter 4 dives deeper into the operation of the VF by looking at Phoenix, our automated binding tool, the Contract Intermediate Language (CIL), and generation and display of the Contract Evaluation Report (CER). Chapter 4 also includes concrete examples illustrating the creation of static checks, dynamic checks, and metric evaluators. Chapter 5 concludes the thesis by providing a summary of our approach, contributions, and areas for future work.

In addition to the information provided within this document, the reader is reminded that the VF is a concrete tool that can be downloaded and used. Additional examples, publications (i.e. [10, 17]), documentation, and the tool itself can be found at the following website: http://vf.davearnold.ca.

2 Related Work

The current chapter contains work that is related to our research. The chapter is grouped into three sections. The first section will present code-based testing tools. That is, the tests themselves are expressed through code. No modeling of the system is required. The second section will present Model-Based Testing (MBT). MBT includes both static and dynamic methods of testing a candidate Implementation Under Test (IUT) against a model. Finally, the third section will look at contract-based testing and corresponding Design-by-Contract (DbC) methodologies. It should be noted that due to the lack of literature covering the execution and instrumentation of contracts, we have broadened the literature review to include additional testing material. Such material includes tools and technologies that result in the generation of test cases. Readers who wish to skip the discussion of each tool and technology presented in this chapter, can jump directly to the related work summary table located in Section 2.4.

2.1 Code-Based Testing

Code-based testing tools allow for tests to be created by specifying additional code that performs tests. That is, the tests are expressed through implementation-specific code. Such test expression includes startup, data creation, the test driver, and teardown code. As such, the test code connects directly with the IUT and drives it through the testing process. The following subsections will briefly look at code-based tools currently used in industry [4].

2.1.1 JUnit

JUnit is a unit testing framework for Java [112]. Unit testing is a method used to validate individual units of source code [93]. A unit is defined as the smallest testable part of a software system [93]. In object-oriented programming a unit is represented by a method. In unit testing each test case is independent from others. Each test is performed using a code model to create mock objects and a test harness that tests the unit in isolation. Unit testing provides the ability to test that individual parts are correct. Testing can be employed early in the implementation process because only a single method is required. Unit testing forms the basis of test-driven development [31].

JUnit was developed by Kent Beck and Erich Gamma [112] and is one of the most successful unit testing frameworks [30]. It is built into the Eclipse [180] development environment and has been ported to other languages including: NUnit (C#) [156], PyUnit (Python) [164], FUnit (Fortran) [71], and CppUnit (C++) [55].

2.1.2 FindBugs, FxCop, and PREFast

FindBugs, from the University of Maryland, is a tool designed to look for bugs in Java programs [69, 89]. Unlike traditional code-based testing tools, where tests are written in code, FindBugs uses the concept of *bug patterns*. A bug pattern is a code idiom that often translates into an error. Bug patterns look for difficult language features, misunderstood API methods, misunderstood invariants when code is modified, and regular mistakes such as typos and incorrect operator usage [69]. FindBugs operates by using static analysis to inspect Java byte code for the occurrence of a bug pattern. Once a bug pattern is found it is reported to the user. FindBugs generates a high degree of

false positives, about 50% in the worst case [69]. FindBugs features a plug-in architecture so that developers may add new bug patterns to the pattern matching engine.

Similar tools such as FxCop [135] and PREFast for drivers [138], both by Microsoft, fall into the same category of code-based tools. FxCop ensures that a given .NET assembly is in conformance with the programming and design rules outlined in the Microsoft .NET Framework Design Guidelines [136]. PREFast is a static analyzer for driver code. These tools use a pattern represented either in source code or low-level byte code and search for its existence (or non-existence) within the IUT. It is these patterns that represent the code model.

2.1.3 Splint

Splint, short for secure programming lint, is a tool for statically checking C programs [66]. Checks include security vulnerabilities, coding mistakes, and Design-by-Contract (DbC) constraints. The specification of checks is performed by annotating the source code via C style comments. Then the Splint tool evaluates the checks against the source code using a lightweight static analysis technique [66].

Splint is part of the lint family of tools [110]. The term was derived from the bits of fiber and fluff found in wool from a sheep [110]. The lint tool originally flagged suspicious and non-portable constructs in C. Today, the term lint is used to indicate tools that flag suspicious usage in any programming language. Additional language-

specific lint tools include: Jlint (Java) [117], JsLint (JavaScript) [56], and Pylint (Python) [126].

Tools like JUnit, FindBugs, and Splint are only a few of the many types of code-based tools. Additional code-based tools include: Axivion Bauhaus Suite [161], ClockSharp [192], Coverity Prevent [54], Fortify [70], HP Code Advisor [85], Klockwork K7 [116], LDRA Testbed [121], Swat [50], and Telelogic Logiscope [177]. Additional tool comparisons can be found in [166] and [203].

2.1.4 Discussion

Regardless of the code-based tool used, each tool directly uses code for the expression of test cases. Such code does not represent a testable model, and is tightly coupled to implementation details. Our approach focuses on an implementation independent testable model that is bound to types and methods found within a candidate IUT, rather than expressing the tests directly in code. We will now examine model-based approaches that do create and use a testable model.

2.2 Model-Based Testing

Model-Based Testing (MBT) involves the derivation of test cases, in whole or in part, from a model that describes at least some of the aspects of the IUT [151, 198]. Typically, MBT is the automation of a black-box test design [33]. A MBT tool uses various test generation algorithms and strategies to generate tests from a behavioural model of the IUT. Such a model is usually a partial representation of the IUT's behaviour. The partial representation is caused by the fact that the model abstracts away some of the implementation details. Test cases derived from such a model are

functional tests on the same level of abstraction as the model. The test cases are grouped together to form an abstract test suite. The abstract test suite cannot be directly executed against the IUT because the test cases do not have the same level of abstraction as the code. Therefore, the abstract test suite must be specialized to create an executable test suite that can operate within the IUT. The specialization is performed by mapping each abstract test case to a concrete test case suitable for execution. In the case of online testing, the abstract test suite exists only as a concept, rather than an explicit artifact [123].

The use of models for the design and development of software systems has increased drastically over the last decade. Consider Bertolino's review of the state-of-the-art in software testing [34]. She remarks: "A great deal of research focuses nowadays on model-based testing. The leading idea is to use models defined in software construction to drive the testing process, in particular to automatically generate the test cases. The pragmatic approach that testing research takes is that of following what is the current trend in modeling: whichever be the notation used, say e.g. UML or Z, we try to adapt to it a testing technique as effectively as possible [...] The idea of model-based testing has been around for decades [...] but it is in the last few years that we have seen a groundswell of interest in applying it to real applications. Nonetheless, industrial adoption of model-based testing remains low and signals of the research-anticipated breakthroughs are weak."

2.2.1 The Unified Modeling Language

One of the most dominant notations for the specification of software models is the Unified Modeling Language (UML) [189]. Such dominance of the UML notation has helped to contribute to the resurgence of MBT, more specifically in the automated generation of test cases from abstract models. However, the UML is simply a notation, and while several UML-based proposals exist for validation, they act as bolt-on solutions to MBT [189]. In order for notations such as the UML and their supporting tools to support the notion of MBT, the viewpoint of validation must be brought into the forefront. For Bertolino [34], this means we must move away from the notion of MBT and towards the dream of test-based modeling. In the case of the UML, the notation itself would require modification to support MBT at its core, rather than a bolt-on solution. As stated by Bertolino, the use of models for the generation of test cases is still in its infancy. However tools and notations, UML compatible and otherwise, are beginning to emerge that support the generation of test cases from a model [123].

2.2.2 Model-Based Tools

Using the definition for a model-based test generator by Alan Hartman [80], we find that model-based tools must have the following two characteristics:

1. The model must define a specification of the IUT (a testable model), and
2. The tool must supply a set of test generation directives that guide test case generation.

The output of a MBT tool is usually a set of test cases that include a sequence of stimuli to the IUT and the expected responses as predicted by the model [34]. In this

section we are going to focus on such MBT tools that fulfill the above characteristics. Before doing so, we wish to distinguish from other classes of tools such as *test automation frameworks* and dedicated *modeling tools*.

A test automation framework accepts tests that have been manually created, automatically generated, or pre-recorded. The automation framework then executes the test sequences without human interaction. Examples of test automation frameworks include: WinRunner by Mercury [128], Rational Robot by IBM [91], and Tau Tester by Telelogic (now IBM) [179]. Our approach directly executes a testable model against a candidate IUT. No test cases are generated as the execution process is driven by the IUT itself. That is, our approach does not require the use of a test automation framework. As such, we will not discuss the operation of specific frameworks here.

Modeling tools are used to create models of an IUT. The tools provide support for model specification, analysis, and maintenance. Modeling tools serve as the creators of input to MBT tools, and do not generally have the ability to generate test cases. Examples of modeling tools include: IBM Rational Rose [92], Objecteering [158], Poseidon [74], Borland Together [39], AutoFocus [150], Simulink [185], The SCR tool [83], and Telelogic (now IBM) Tau [178]. As our approach operates directly on a binary candidate IUT, we do not require a model of the IUT. That is, our tool automatically infers a model of the IUT from the binary itself. No additional IUT specification is required. As such, we will not take our discussion of modeling tools any further.

2.2.3 Model-Based Testing and Our Approach

In order to develop our proposed approach to the problem described in Chapter 1, we have first looked at current and state-of-the-art Model-Based Testing (MBT) tools. We will also attempt to address some of the issues, outlined by Grieskamp in [76], that exist in current tools. Our approach uses a new notation, the ACL that is grounded at the requirements level and uses scenarios as the main specification mechanism. Scenarios allow stakeholders to take part in the model specification process. In order to avoid the state explosion problem, our approach does not use states to represent the underlying system, but rather performs pattern matching between an execution grammar specified in the model and actual execution traces generated by a candidate IUT. Details regarding our scenario matching process will be presented in Chapter 4. Validation is determined by the IUT's ability to execute the grammar of responsibilities specified in the model defining the scenario.

The following subsections will look at some of the current MBT tools [80, 123]. As we examine the various tools, background on any notation or theory used by the tool will also be presented. That is, the following subsections will serve two purposes: to present current MBT tools and to provide relevant background on current test notations and theory.

2.2.3.1 Lutess

Lutess [41] is a testing environment for synchronous reactive systems that is based on the synchronous dataflow language Lustre [79]. Lutess accepts three elements as input for the automatic generation of a testing harness: a test sequence generator, the

IUT, and an oracle. The test sequence generator is derived from an environment description written in Lustre. The description is composed of a set of constraints that describe the set of interesting test sequences, operational profiles, properties to be tested, and behavioural patterns. The environment description can be viewed as a synchronous program that observes the input/output stream of the IUT. The environment then determines if a given test sequence is realistic with respect to the IUT, and then the oracle determines correctness.

The oracle and the IUT can be provided in one of two ways. The first is as a synchronous program, and Lutess will handle the IUT as a black-box. The second way is to supply the IUT as a program written in Lustre [79]. In this case, Lutess automatically compiles and integrates the IUT into the test harness. We will now provide a brief overview of the Lustre language.

2.2.3.1.1 Lustre

Lustre is a high-level programming language for reactive systems consisting of two main concepts: *time-synchronization* and *dataflow orientation* [49, 79]. Lustre assumes that every reaction of the program to an external event is executed instantaneously. That is, Lustre assumes that the state of the system being tested does not change during the computation of a reaction. Such an assumption allows Lutess to use the notion that each internal event of the IUT takes place at a known point in time.

The dataflow approach used in Lustre creates a functional representation. Functional representations are open to automated analysis and transformation because

they are side-effect free. Parallel components are naturally expressed in Lustre through the use of independent dataflows. Synchronization is implicitly handled through the creation of data dependencies between different dataflows. Figure 3 contains an example of a counter specified in Lustre.

```
node COUNTER(val_init, val_incr : int; reset : bool)
returns (n : int)
let
   n = val_init -> if reset then val_init else pre(n) + val_incr;
tel;
```

Figure 3 - A Counter in Lustre [79]

The example above implements a simple counter as a node. A node recombines a set of dataflows into a new one. The *val_init* variable is used as the initialization of a new flow that is then incremented by *val_incr* in each call. Figure 3 shows two of the time operators provided by Lustre: *follows* and *pre*. The follows operator (->) is used to indicate that the element on the left side of the operator occurs before the element on the right. That is, the right element follows the left element. The second time operator, *pre*, is used to access the value of a variable during the previous flow. Figure 4 shows the use of the counter node. The first line generates a sequence of all even numbers. The second line cycles through numbers between zero and four.

```
even = COUNTER(0, 2, false);
mod5 = COUNTER(0, 1, pre(mod5) = 4);
```

Figure 4 - Use of the Counter Node

Once the environment description is specified in Lustre, Lutess constructs a test sequence generator. Formal construction steps are provided in [41]. In short, the test sequence generator is based on the provided environment description and a set of

probabilistic constraints to guide the generation. The environment description computes a predicate that indicates if the test sequence is relevant or not. The test sequence generator then inverts the predicate. That is, it computes the set of inputs for the IUT that satisfy the environment description. The provided oracle uses the input along with the output generated by the IUT to determine a pass or fail verdict for the test sequence.

Lutess allows for the construction of a test harness for fully automated test sequence generation and execution in the context of synchronous reactive systems. The harness is constructed from an IUT, a test environment specification, and an oracle. The IUT and the oracle can be provided as synchronous reactive programs. The test sequence generated is based on the test environment specification provided in Lustre. Lutess is not able to handle IUTs that have numerical inputs or outputs [123]. It is also not able to express liveness properties or to generate test suites based on coverage.

2.2.3.2 Lurette

Lurette [165] uses a similar approach to Lutess and is based on Lustre. Like Lutess, Lurette constructs the test harness from the IUT, a test sequence generator, and an oracle. Both tools use environment descriptions and oracles given as synchronous observers. A synchronous observer is a program that implements acceptors for sequences. That is a synchronous observer will output true as long as the sequence given to the observer represents a valid sequence of events in the environment.

Unlike Lutess, Lurette is able to validate systems that have numerical inputs and outputs. Lurette also requires that the IUT is provided as a C program that implements a predefined set of methods. The test sequence is generated on the fly during IUT execution. An initial input is provided by the test sequence generator and fed into the IUT. From then on the IUT and the test generator alternate between computing outputs and inputs. Like Lutess, Lurette is not able to deal with liveness properties and it only allows for the specification of test purposes in the form of safety properties [123]. We now look at one final Lustre-based tool: *GATeL*.

2.2.3.3 GATeL

GATeL [127, 181] takes a different approach from the two previously discussed tools. Lutess and Lurette start all test sequence generation from the initial state. Then the sequence of execution is generated on the fly. That is, each output set from the IUT is used to compute the next input set. The process is iterated until a negative test verdict is produced, or until the entire test sequence has been executed. GATeL starts with a set of constraints on the last state of the sequence. The set of constraints can contain invariants and any other type of constraint that can be expressed in Lustre. GATeL allows for the specification of state invariant properties and state path predicates to express the test purpose. To support such path predicates, GATeL constructs its test sequence backwards. Doing so generates the test sequence before IUT execution. The backwards search is implemented using a backtracking algorithm [127]. To find a matching sequence, GATeL uses constraint logic programming [60] in a search process going from the last to the first state. GATeL requires the IUT or a complete specification

of the IUT, the environment description, and a test objective to be supplied in Lustre. All three components are prohibited from using real variables or tuples [123].

The lustre-based approaches assume that each reaction to an external event occurs instantaneously. Such assumption implies that the state of the system does not change during the computation of a reaction. In object-oriented systems this assumption translates into the execution of a method where the method is side-effect free. Such a constraint is too restrictive for our approach. In addition, the tools presented in this section require the creation of a test case generator and IUT specific oracle. As we will see shortly, our approach allows for a single testable model to be applied to multiple candidate IUTs without the need for a specialized oracle or test harness. We will now move away from Lustre-based tools and look at a graphical constraint logic-based tool: *AutoFocus*.

2.2.3.4 AutoFocus

AutoFocus is a graphical tool targeting the modeling and development of distributed systems [90]. AutoFocus describes a system as a collection of components that communicate over typed channels. Each component can be decomposed into networks of communicating subcomponents. That is, an AutoFocus model is a hierarchical organized set of time-synchronous communicating Extended Finite State Machines (EFSMs) that use functional programs to represent guards and assignments. Models created in AutoFocus can be used for both code generation and testing.

Model-Based Testing (MBT) in AutoFocus requires a model of the IUT, a test case specification, and the actual IUT. The test case specification can take one of three forms: *functional*, *structural*, or *stochastic*. Functional test specifications are used for testing a particular feature or requirement. AutoFocus uses a non-deterministic state machine to represent sequences of interest. That is, execution of a state transition will trigger the feature being tested. In many cases there are several ways to test a given feature. In such cases, a non-deterministic state machine represents the set of possible test sequences in a natural way. Structural specifications take advantage of the hierarchical modeling supported by AutoFocus. They make it possible to generate test suites independently for different system components and to use these unit tests to generate integration tests [163]. Stochastic test specifications are used to avoid randomly generating test cases that are too similar.

Like GATeL, AutoFocus generates tests based on constraint logic programming. The AutoFocus model is translated into a constraint logic programming language and then is executed symbolically. For details regarding test case generation see [90]. In addition AutoFocus is able to generate test cases that conform to a given coverage criteria or to a given functional unit. AutoFocus uses EFSMs to represent the model, such modeling suffers from the state space explosion problem (more on this later) [76]. In addition, the tester is required to provide an EFSM representation of the IUT which we feel places too much of a burden on the user and is error prone if done manually. As previously stated our approach works directly with a binary IUT. The model is automatically derived from the IUT and is not required from the user. Our approach directly executes

the candidate IUT, as opposed to symbolically with AutoFocus, using a pattern matching approach instead of a state machine. Details regarding the execution and scenario matching process are provided in Chapter 4.

2.2.3.5 Conformance Kit

The Conformance Kit was developed in the early 1990s by KPN research to support the automatic testing of protocol implementations [119]. The kit uses Extended Finite State Machines (EFSMs) to represent system specifications. In addition to the typical EFSM constructs like variables and conditions on transitions, additional constructs like gates are provided to facilitate mapping to the IUT. The gate construct allows for splitting a specification into several EFSMs that communicate through gates.

The test suite generation process begins by using a converter to transform an EFSM into an equivalent Finite State Machine (FSM). The transformation is accomplished by enumerating the variables. A second step then minimizes the resulting FSM. Once the EFSM has been transformed into a minimized FSM the Conformance Kit offers several FSM techniques to derive test cases [119]. A *transition tour* is possible if the FSM is strongly connected. The disadvantage of a transition tour is that only the input/output behaviour is tested, end state correctness is not checked [123]. To address this disadvantage a tour including unique input/output sequences is created that also checks the end states. Such a tour is known as a *partition tour* because it creates a set of single sequences for each transition, rather than one finite test sequence covering all transitions. Each single sequence contains three elements:

1. A synchronizing sequence to transfer the FSM to its initial state.
2. A transferring sequence to move to the source state of the transition being tested.
3. A unique input/output sequence that verifies the correct destination state of the transition.

Finally, a *random sequence* can be computed to use a random number generator to produce events. The result of generating tests with the Conformance Kit is expressed in Tree and Tabular Combined Notation Version 2 (TTCN-2) [101]. As with AutoFocus, the Conformance Kit uses state machines to represent the testable model, and suffers the same state space explosion and expression problems as with AutoFocus [76]. The result of test generation is a set of TTCN-2 test cases. These test cases are not directly executable and require the creation of implementation-specific glue code for execution against a candidate IUT. Our approach removes this restriction by creating a testable model that is directly executed against a candidate IUT without the need for IUT specific glue code or test oracle. As TTCN is a common notation for the specification of test cases, we will provide a brief introduction to TTCN.

2.2.3.5.1 Tree and Tabular Combined Notation

TTCN was standardized in 1992 [100]. Since its standardization, TTCN has been widely used for describing protocol conformance test suites. The main characteristics of TTCN include [104]:

1. Its tabular notation allows users to describe easily and naturally in a tree form all possible scenarios of stimulus and various reactions between the tester and the IUT.
2. The verdict system is designed to facilitate conformance that the test result agrees with the test purpose.
3. It provides a mechanism to describe constraints on received messages so that conformance can be mechanically evaluated against the test purpose.

The first version of TTCN was not designed to handle concurrent behaviour within the tester and the IUT, modular test case construction or non-trivial data representation. As a result, TTCN was extended [101] to include a concurrency mechanism, modules and packages for reuse, and the integration of Abstract Syntax Notation One (ASN.1) [94] for the description and manipulation of data. ASN.1 is a precise formal notation used in telecommunications and networking for the description of data structures. ASN.1 provides a set of formal rules for describing the structure of objects that are independent of machine-specific encoding techniques.

The third, and current, version of TTCN was completed in October 2000 [104]. TTCN-3 was designed to include missing elements from TTCN-2 while still retaining its features. The main difference in TTCN-3 is that it can be expressed in two formats instead of just the Tabular Format for TTCN (TFT). TTCN-3 also includes a graphical presentation format, Graphical Format for TTCN (GFT). GFT uses a subset of Message

Sequence Charts (MSC) [103] with test-specific textual extensions. Textual extensions allow existing MSC tools to be used with little modification.

Figure 5 provides an example of TTCN-3 syntax [173]. The example defines a request to get the current weather conditions for a given city. A concrete weather request for Paris is created, and then the test case checks to see if a response was received. If a response was received then the test case passes, if not the test case fails.

With TTCN-3, TTCN has moved from a protocol-based testing notation to a general purpose testing notation. TTCN-3 supports protocol, integration, module, system, and API testing. TTCN-3 can be seen as a black-box method for the testing of interface-based components. One of the main drawbacks of TTCN-3 is that it is not backwards compatible with previous versions of TTCN [173]. With respect to our research, we remark that TTCN only supports static test interfaces and as such does not have the binding and IUT independence that is provided by our approach. In addition, TTCN does not address non-functional requirements. Additional information on all versions of TTCN can be found in [100, 101, 104, 105, 173] and a small case study outlining the use of TTCN-3 can be found in [61].

Returning to the Conformance Kit, it has been integrated into several tools and approaches [123], we will now look at one such tool.

```
module SimpleWeather {

type record weatherRequest {
      charstring location,
      charstring date
}

template weatherRequest ParisWeekendWeatherRequest := {
   location := "Paris",
   date := "15/06/2006"
}

type record weatherResponse {
   charstring location,
   charstring date,
   integer temperature,
   integer windVelocity,
   charstring conditions
}

template weatherResponse ParisResponse := {
   location := "Paris",
   date := "15/06/2006",
   temperature := (15..30),
   windVelocity := (0..20),
   conditions := "sunny"
}

type port weatherPort message {
   in weatherResponse;
   out weatherRequest;
}

type component MTCType {
   port weatherPort weatherOffice;
}

testcase testWeather() runs on MTCType {
   weatherOffice.send(ParisWeekendWeatherRequest);
   alt {
      [] weatherOffice.receive(ParisResponse) { setverdict(pass) }
      [] weatherOffice.receive { setverdict(fail) }
   }
}
control { execute(testWeather()) }
}
```

Figure 5 - TTCN-3 Example[2]

[2] Example modified from [173].

2.2.3.6 PHACT

In 1995, Philips extended the Conformance Kit creating a set of tools called the Philips Automated Conformance Tester (PHACT) [160]. PHACT extends the Conformance Kit with the ability to execute the computed TTCN-2 test cases against an IUT. To connect the abstract events defined within TTCN-2 to corresponding IUT events, a Protocol Implementation eXtra Information for Testing (PIXIT) [97] has to be created. PHACT uses three elements for test case execution: the *supervisor*, the *stimulator*, and the *observer*. The stimulator and the observer interact directly with the IUT. As such, for each candidate IUT a dedicated simulator and observer must be created. That is, PHACT does not have the binding support found in our approach. The supervisor interacts with both the stimulator and the observer to execute the TTCN-2 test suite and to give a pass/fail result based on the observed behaviour.

PHACT compiles and links several modules with the observer and simulator to test an IUT. The compilation and link operation results in an executable tester that can either be linked directly with the IUT or used as an external library. PHACT supports IUTs written in C or Java. PHACT is not publically available but its use has been documented by case studies [82, 149].

While PHACT is able to execute TTCN-2 test cases against an IUT, it requires IUT specific glue code in the form of a stimulator and observer. Our research eliminates the need for implementation dependent stimulators and observers by directly executing the IUT against an abstract testable model. With respect to PHACT such a connection is achieved through the manual creation of a PIXIT specification that maps TTCN-2

elements to events generated by the IUT. Through the notion of automated bindings, our research eliminates the need for the manual creation of such a mapping, allowing for an abstract testable model to be executed against several candidate IUTs. Details regarding the binding process will be presented in Chapter 4.

2.2.3.7 TorX

TorX is both an architecture for a flexible, open, testing tool for test case derivation and execution, and an implementation of such a tool [193]. TorX was developed in the late 1990s as a Dutch academic-industrial research project [194]. TorX can be used with any modeling language that can be expressed as a Labeled Transition System (LTS). It has been connected to CADP [67], Trojka (based on SPIN [88])[199], and to the LOTOS [96] simulator Smile [65]. For test purposes, TorX uses a special regular expression-like language and tool called jararaca [123]. The tool gives access to the LTS described in the test purpose. The main disadvantage with TorX is that it expects the user to provide the connection to the IUT. The connection is made through a program that implements the TorX adapter interface. In this interface abstract input and output actions are exchanged. It is the responsibility of the user to provide the glue code for the encoding, decoding, and actual IUT connection. For more information on the inner workings of the TorX tool see [194].

2.2.3.8 AGEDIS

Automated Generation and Execution of test suites for Distributed component-based Software (AGEDIS) [3] was a project running from October 2000 until the end of 2003. The project was developed by a consortium consisting of seven industrial and

academic research groups headed by IBM Research. The goal of the AGEDIS project was to develop a methodology and tools for the automation of software testing, with emphasis on distributed component-based software systems.

AGEDIS is based on an open architecture focusing on interfaces. The main AGEDIS interfaces are as follows [81]:

- Behavioural modeling language
- Test generation directives
- Test execution directives
- Model execution interface
- Abstract test suite
- Test suite trace

The first three interfaces constitute the main user interface, and the last three are used internally. AGEDIS Modeling Language (AML) is a UML 1.4 profile that serves as the behavioural modeling language. Class diagrams together with associations describe the structure of the IUT. The behaviour of each class is defined with a state machine using Verimag's IF [42] as the action language. Stereotypes are used to describe the interface between the model and the IUT. For more information on AML see [3]. Test purposes are provided as test generation directives in the form of system level state machines or MSCs. AGEDIS also provides five predefined test generation strategies [81]:

- Random test generation
- State coverage
- Transition coverage
- Interface coverage
- Interface coverage with parameters

Abstract specification parts like classes, objects, and methods have to be mapped to the IUT. Such mapping is accomplished by using an XML schema that instantiates the test execution directives interface. The model execution interface encodes all the behavioural models of the IUT into an IF representation for abstract test suite generation.

The test suite is then created using an XML schema to represent a set of steps that consist of method calls, observations, and directions for the calculation of verdicts. The test cases can then be parameterized to run with different values, and other test cases can be invoked from within a test case. AGEDIS is restricted to static systems. That is, objects cannot be created or destroyed during test case execution. In contrast, as will be shown shortly our approach supports test cases for dynamic systems. AGEDIS supports IUTs written in C, C++, and Java. AGEDIS itself is implemented in Java. For more information on AGEDIS see [3, 81].

2.2.3.9 AsmL

The Abstract State Machine Language (AsmL) is developed by the Foundations of Software Engineering (FSE) group at Microsoft Research [139]. AsmL is an executable

specification language based on the theory of Abstract State Machines (ASMs) invented by Yuri Gurevich [78]. AsmL is a .NET language and is aimed at specifying systems in an object-oriented manner. AsmL has been embedded into Microsoft Word using a plug-in, and uses XML for literate specifications.

AsmL has a built-in conformance test facility that is based on two steps. First the ASM specification is transformed into a FSM, and then well known FSM-based algorithms [78] are used to generate a test suite. The process of generating a FSM out of an ASM is a difficult task that requires considerable expertise from the tester for defining both a suitable equivalence relation and a relevance condition that prunes the state space into something manageable. It is also problematic that the ASM to FSM transformation may yield a FSM that is non-deterministic. The AsmL test generator cannot handle non-deterministic FSMs. Microsoft Research is working to address the non-deterministic FSM generation issue and state space explosion problems in their current state-of-the-art tool: *Spec Explorer.* For more information on AsmL see [27, 77, 78, 139].

2.2.3.10 Spec Explorer

We have presented several model-based tools. Each tool accepts a specification, an IUT, and a test purpose. The tools use the specification to create an internal model, usually based on a Finite State Machine (FSM) or FSM-like representation. The model is then traversed according to the test purpose leading to the generation of test cases. The test cases can be specified using different notations, such as TTCN or an XML schema. Our approach differs from the existing model-based tools, in that we attempt

to have a model that is not separate from the code. That is, our testable model is to be tightly bound to the IUT as each element within the model must be directly mapped to an implementation artifact. Details regarding such a binding process will be presented later. The result of such a binding is that the generated test cases are directly executable because a direct binding exists between the IUT and the testable model. The previously discussed model-based tools do not generate directly executable tests. This is not to say that the generated tests are not executable, but glue code must be provided for such execution. Our approach allows for the direct execution of test cases, and white-box testing such as Design by Contract (DbC) [130] to be used within the TRM. A background on contract-based testing will be presented in Section 2.3. We will now examine the state-of-the-art with respect to model-based testing: *Spec Explorer.*

Spec Explorer [46] is a state-of-the-art MBT tool developed by the FSE group at Microsoft Research. Spec Explorer is based on the previously discussed AsmL tool [139]. It has been used internally within Microsoft since 2004 for testing operating system components, and is currently available for download [145]. Spec Explorer is a tool for testing reactive, object-oriented systems. The inputs and outputs can be viewed as parameterized action labels that represent the invocations of methods with dynamically created object instances and other complex data structures.

A model in Spec Explorer is specified using Spec# [146]. Spec# is a textual programming language that includes and extends C# [98]. In addition to the functionality provided by C#, Spec# adds [47]:

- Pre- and post-conditions.
- High-level data types.
- Logical quantifiers like *forall* and *exists*.

A Spec# model can be viewed as a program. It can be compiled and executed just like a C# program. A Spec# model can call .NET framework code, and the model can be explored using Spec Explorer. Before we look at how Spec Explorer works, we will present an example[3] of a Spec# model. Figure 6 defines a model of a distributed, reactive chat system using Spec#. The system supports an arbitrary number of clients. Each client may send text messages that will be delivered by the system to all of the other clients that have entered the chat session. A client always receives messages from a sender in the order sent. However, if there are multiple senders, the messages may be interleaved.

The model in Figure 6 contains a class that represents the abstract state and operations of a client. Each client instance contains two variables: *entered* and *unreceivedMsgs*. The *entered* variable indicates if the client has entered the chat session or not. The *unreceivedMsgs* variable stores separate queues for messages that have been sent by other clients but not yet received by the current client. The reader should note that Spec# represents a model and not an IUT. No IUT would be expected to maintain queues of messages that it has not yet received. The idea of Spec# is that it

[3] Chat example adapted from the example in [44].

should be easier to model a system rather than implement it. Figure 6 then defines four actions.

```
class Client {
   bool entered;
   Map<Client,Seq<string>> unreceivedMsgs;
   [Action] Client() {
      this.unreceivedMsgs = Map;
      foreach (Client c in enumof(Client), c != this) {
         c.unreceivedMsgs[this] = Seq{};
         this.unreceivedMsgs[c] = Seq{};
      }
      entered = false;
   }
   [Action] void Enter()
      requires !entered; {
      entered = true;
   }
   [Action] void Send(string message)
      requires entered; {
      foreach (Client c in enumof(Client), c != this, c.entered)
         c.unreceivedMsgs[this] += Seq{message};
   }
   [Action(Kind=ActionAttributeKind.Observable)]
   void Receive(Client sender, string message)
      requires sender != this &&
         unreceivedMsgs[sender].Length > 0 &&
         unreceivedMsgs[sender].Head == message; {
      unreceivedMsgs[sender] = unreceivedMsgs[sender].Tail;
   }
}
```

Figure 6 - Chat System in Spec# [46]

The first action is the constructor. The constructor creates an instance of a new client. Once the client has been created it will contain empty message queues between the new client and all previously created client instances. Each queue can be viewed as a virtual one-way channel between each pair of client instances. The system will contain *n(n-1)* queues if there are *n* clients.

The second action, *Enter*, advances the client state so that it has entered the chat session. The *Enter* action uses the *requires* keyword to indicate a precondition that ensures the client cannot already be in the chat session.

The *Send* action appends a new message to the queues of unreceived messages in all other clients that have entered the session. Again, the *requires* keyword is used to ensure that the client has entered the chat session before sending a message.

The final action, *Receive*, extracts a message from a sender and places it into the queue for that sender. Spec# supports two types of actions: *controllable* and *observable*. A controllable action is one that can be invoked by a user to provide system input [46]. An example of a controllable action is the *Send* action in Figure 6. Observable actions are actions that are not controlled by the user, and respond to an output message from the system. An example of an observable action is the *Receive* action in Figure 6.

Using a model specified in Spec#, Spec Explorer extracts a representative behaviour of the system according to user-defined parameters for scenario control [46]. Spec Explorer accomplishes this using a state exploration algorithm that works as follows: In a given model the current state determines the next possible invocations. An invocation is denoted by an action/parameter combination that is *enabled* by the preconditions in the current state. Spec Explorer then computes the successor states for each invocation. The process is repeated until there are no more states or invocations to

explore. For more information on the state exploration algorithm used by Spec Explorer see [28, 46, 47, 48, 145].

Parameters used for the invocations are provided by state dependent parameter generators. Enabledness is determined by the precondition of an action. Spec Explorer also uses heuristics to prune the generated state space [46]. Figure 7 provides an example of a scenario extracted from the chat model in Figure 6. State filters are used to restrict the number of clients and to avoid the case where the same message is sent twice by a client.

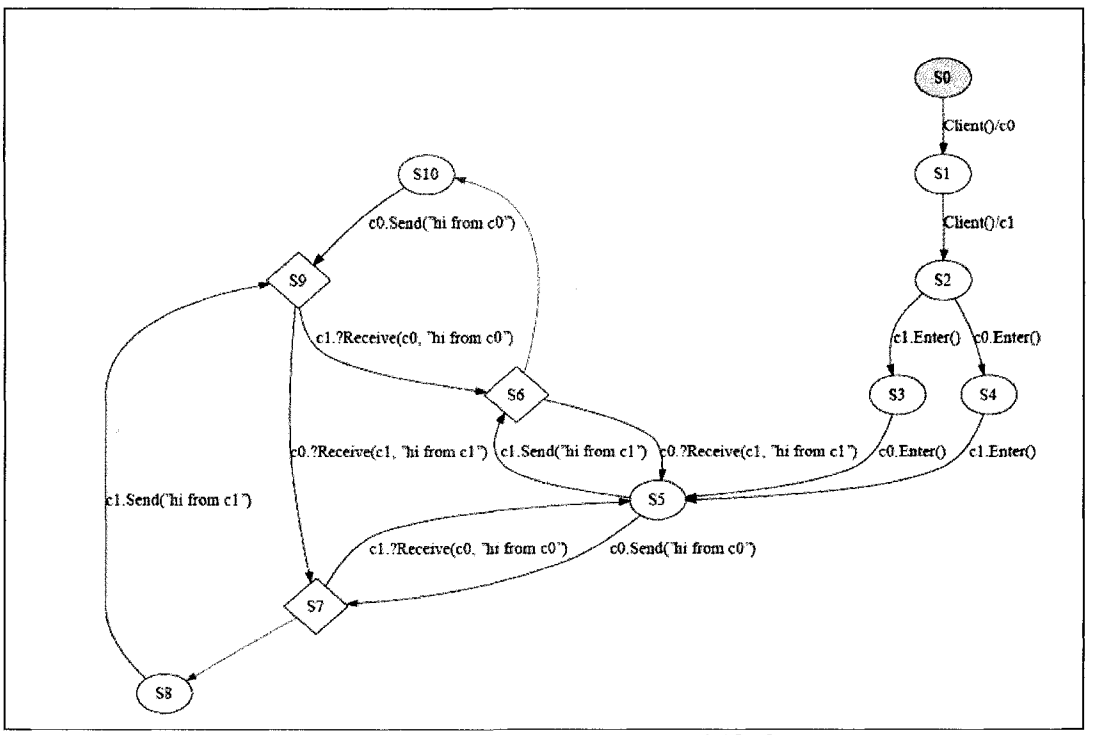

Figure 7 - A Spec Explorer Scenario Model [46]

Each node in Figure 7 represents a distinct state of the system. Each arc is a transition that can be used to change state. Each state can either be *passive* or *active*. Passive states are represented by diamonds and indicate that a client may wait for an

action from the system or transition into an active state after a state-dependent timeout occurs. Ovals represent an active state where a client may give the system new work to do.

Spec Explorer indicates that an IUT conforms to the model if the following conditions are met [46]:

- The IUT must be able to perform all transitions outgoing from the active state.
- The IUT must produce no transitions other than those outgoing from a passive state.
- Every test must terminate in an expected accepting state.

Spec Explorer implements conformance testing in one of two modes: *offline* or *online* [46]. Offline testing reduces the model to a test suite that can be compiled to create a stand-alone test driver and oracle. In online, or on-the-fly, testing model exploration and conformance testing are merged into a single algorithm. If the IUT is a non-distributed .NET program, then Spec Explorer will automatically generate the testing harness. In other cases the tester is required to write a wrapper around a distributed IUT. For more information on Spec# and Spec Explorer see [28, 46, 47, 48, 145, 146]. We will now move past the state-of-the-art and look at areas of improvement within model-based testing.

2.2.4 Beyond State-Of-The-Art

Testing is one of the most expensive aspects of software development [38]. Due to the reduction of time and skilled personnel, software is often not tested as thoroughly

as it should be [204]. Current testing practices are not only labourious and expensive but often unsystematic, lacking an engineering methodology and discipline, and adequate tool support [76]. Model-Based Testing (MBT) is a promising approach to address these problems. However, even within Microsoft only about 5-10% of their product teams are using or have tried using MBT [76]. Grieskamp [76] from Microsoft Research indicates this low percentage can still be considered a success compared to the use of formal quality assurance approaches, such as verification [76]. He argues that the slow adoption of MBT is the result of the extensive time required to learn modeling notations together with a lack of state-of-the-art authoring environments, missing scenario-based modeling, poor documentation, and no integration with test management tools. He states that the lack of scenario-based modeling makes it hard to involve stakeholders in the testing process, rather than just the test organization.

Regardless of the specific model-based tool, all of the tools presented in Section 2.2 strive to express a model using an executable specification language. Such specification languages use *guarded-update* rules on a global data state representing the IUT. The rules describe transitions between data states and are labeled with *actions*. Each action corresponds to a method invocation in a test harness, or in modern tools directly in an IUT. The rules can be parameterized by the tester using techniques like equivalence partitioning [76]. Grieskamp further argues that the challenge with the development of MBT tools and techniques lies in the ability to easily express the testable model at a level of abstraction that is implementation independent, yet is executable. The following subsections will examine areas of improvement within MBT.

2.2.4.1 *Modeling*

Grieskamp states that modeling notations have become less important than the environments that support them [76]. Testers who are already using a modern development environment such as Eclipse or Visual Studio are used to having advanced features like automatic indentation, incremental compilation, context-sensitive completion, and refactoring. These users are unlikely to adopt a new technology in an environment that does not provide the same functionality as the one they are already used to. The lesson learned by Microsoft is that if one comes up with a new notation, the tools that support such a notation should provide modern authoring support [76]. Creating such authoring support is a non-trivial task, usually an order of magnitude more difficult than writing a compiler.

As shown in Section 2.2, we have looked at MBT tools that are based on formal languages. The examples we have presented of such syntax have been simple and intuitive. However studies have shown that such formal notations have a learning curve that is too steep for adoption [204]. Testers struggle with concepts like universal and existential quantification and set comprehensions [204]. These issues lead to the requirement of a testable language that is executable yet abstract enough to be understood by stakeholders. As an example, consider the Spec# language [146]: Spec# aims to reduce the gap between the specification language and the implementation language. While Spec# can be easily picked up by someone with C# programming experience, it is difficult for a stakeholder to understand, as it resides at a low-level of abstraction.

Looking deeper into modeling techniques, Grieskamp notes that the main obstacle is not the modeling language itself but rather the modeling style [76]. He argues that most MBT approaches are not attractive in the requirements phase because they are state-based and not scenario-based. Incorporation of stakeholders requires a scenario-based modeling technique [76]. In addition to any tool, notation, or technique, Stobie argues that, supporting education and documentation is also required for a successful adoption to occur [174].

2.2.4.2 Technology

Model-Based Testing (MBT) is known to generate a large amount of tests even for small models. The large amount of tests turns out to be quite a problem in practice, rather than an advantage [76] and is known as the *state explosion problem*. A large number of states increases the time required to run the test suite. The required time is a significant cost factor. Grieskamp remarks that the solution to the state explosion problem is not stochastic on-the-fly (online) testing [76].

To reduce the number of states, and thus the number of test cases, the notion of *test selection* is used to select a set of representative tests from the model. Test selection is usually implemented through graph traversal techniques that are applied to a FSM representing the model. Other techniques include parameter generation, pair wise combination, equivalence partitioning, etc. Users of MBT tools within Microsoft complain that they do not have enough *fine-grained* control over the test selection process [76, 204].

Some of the tools we have examined in this section define the notion of a *test purpose*. The test purpose is used to select some desired behaviour from the model. For a summary of test purpose techniques and notations see [68, 195]. Grieskamp believes that instead of having additional notation for the description of test purposes, models should be used to express the test purpose and he views the test selection problem with test purposes as a model composition problem [76]. In a scenario-based approach the test purposes would be specified within the scenario.

In addition to MBT, our approach seeks to take advantage of contract-based testing. We will now provide relevant background on contract-based testing and how our approach defines and uses contracts to compliment our use of scenarios.

2.3 Contract-Based Testing

As the name suggests, contract-based testing resembles a contract between two entities, such as a person and a business. When it comes to contract-based testing, a contract is specified by a user and the candidate implementation must respect and adhere to the contract. The use of contracts for the specification of behavioural compositions was first proposed by Helm, Holland and Gangopadhyay in 1990 [84]. They define behavioural compositions as groups of interdependent objects cooperating to accomplish tasks. Contracts were then used as a construct for the explicit specification of behavioural compositions [84]. The purpose of contracts was to help formalize the collaboration and behavioural relationships between objects. As an example of such a behavioural relationship consider the Model, View, Controller (MVC) paradigm [45]. The controller and view objects interact to ensure that the view always

reflects the state of the model. The MVC is similar to the *observer* design pattern [72]. The initial contract they developed is shown in Figure 8. As illustrated, their contracts are defined in a high-level language that allows for an abstract description of the required behaviour. Their notation is formal in nature, and consists of specifying the required instance variables and methods for a given object. Their contract language only supports two actions, the sending of a message, denoted by $P \rightarrow M$, and the setting of an instance variable, denoted by Δv [84]. Their definition of contracts also includes two operations: *refinement* and *inclusion*. The motivation for the two operations was the ability to express complex behaviour in terms of simpler behaviour. Refinement allows for the specialization of contractual obligations and invariants [46]. Inclusion allows contracts to be composed from simpler sub-contracts [84]. Our proposed framework also supports refinement and inclusion; details will be presented in Chapter 3.

```
contract SubjectView
    Subject supports [
        value : Value
        SetValue(val:Value) ↦ Δvalue {value = val}; Notify()
        GetValue() : Value ↦ return value
        Notify() ↦ (|| v : v ∈ Views : v→Update() )

        AttachView(v:View) ↦ {v ∈ Views}
        DetachView(v:View) ↦ {v ∉ Views}
        ]
    Views : Set(View) where each View supports [
        Update() ↦ Draw()
        Draw() ↦ Subject→GetValue() {View reflects Subject.value}
        SetSubject(s:Subject) ↦ {Subject = s}
        ]
    invariant
        Subject.SetValue(val) ↦ (∀v : v ∈ Views : v reflects Subject.value)
    instantiation
        (|| v : v ∈ Views : (Subject→AttachView(v) || v→SetSubject(Subject)) )
end contract
```

Figure 8 - SubjectView Contract from Helm, Holland, and Gangopadhyay [84]

With the proliferation of component-based software and the promise of reuse through components, technologies such as the Common Object Request Broker Architecture (CORBA) [187] and the Distributed Component Object Model (DCOM) [140] have emerged to help encourage the use of component software. Contracts are used to specify how such components are to be integrated and used within a larger system. Four levels of contracts have been devised for the correct integration of components [35]. The first level, basic, is required to make the system work. The second level, behavioural, improves the level of confidence in a sequential context. The third level, synchronization, improves confidence in a distributed or concurrency context. Finally, the fourth level, quality of service, is used for non-functional requirements. The following subsections will explore each of the four levels.

2.3.1 Basic Contracts

Basic contracts are used to define what must be done in order to make the system work [35]. Here the contract specifies the interface that is to be used in order to communicate with the component or object. In CORBA such an interface is specified by using Interface Definition Language (IDL) [187]. In DCOM and object-oriented systems the interface is specified through an actual class interface. Languages such as Java [75] or C# [98] provide an interface primitive type. Languages such as C++ [99] provide the notion of an abstract class to represent an interface. The actual interface provides details regarding the names and parameters of messages that are required in order to interact with the component or object.

Basic contracts reside at the lowest level, and such contracts are not negotiable. That is, a specified message or parameter cannot be omitted or changed. As an example, consider an interface to an object. Consumers of that object are not permitted to make changes to the interface provided by the object.

Basic contracts operate at the structural level and do not include behavioural elements. We will now move up one level and examine behavioural contracts.

2.3.2 Behavioural Contracts

An interface only specifies the available methods and their parameters. Interfaces do not specify the effect on the invocation of a method. A methodology known as *Design-by-Contract* (DbC) [131, 148] was created for the specification of behavioural contracts and was first implemented in the Eiffel language [64]. DbC is composed of three major elements: *preconditions*, *post-conditions*, and *class invariants*. The following subsections will provide a brief overview of the three elements.

2.3.2.1 Preconditions

A precondition [131] is located at the beginning of a method. Preconditions specify requirements that must exist in order for a method to run correctly. If one or more requirements specified in a precondition fail, then an exception, in the C++ sense [99], is raised. A precondition can be viewed as a set of Boolean conditions that must be true in order for a method to execute correctly. For example, consider a *pop* method defined on a stack. A suitable precondition would check that the stack is not empty. If the *pop* method is called on an empty stack, the precondition fails and an exception is raised.

When a precondition fails, the caller of the method containing the precondition is responsible, not the method itself. That is, the client (invoker) has violated its contract with the supplier (method).

2.3.2.2 Post-conditions

A post-condition [131] is located at the end of a method. The post-condition ensures the executed method has completed all of its required tasks. If one or more of these tasks has not been completed, the post-condition fails and an exception is raised. Like the precondition, a post-condition consists of a set of Boolean conditions. All of the conditions must evaluate to true for the post-condition to be valid. The Boolean conditions found in a given post-condition have two special features: *previous value* and *return value*.

The first special feature is that a post-condition can reference the value of a given parameter or attribute before the method is executed. The feature can be used to determine if the value of a given parameter or attribute has changed during execution of the method. Using our stack example, consider the operation of the *pop* method. The *pop* method takes the top element off the stack and returns it to the caller. A valid post-condition would ensure that the new size of the stack is one less than the size of the stack before the method executes.

The second special feature is that post-conditions can reference the return value of a method. This feature can be used to determine if the correct value is being returned

to the caller of the method. Returning to the stack example, we can define a post-condition to make sure that we are returning the correct element.

When a post-condition fails in a given method, the fault lies in the method itself. The method (supplier) did not fulfill its part of the contract. The caller (client) did, as the precondition must have been valid. If the precondition had been invalid, an exception would have been raised before the method could execute.

2.3.2.3 Class Invariants

Preconditions and post-conditions are bound to individual methods. A class invariant [131] allows for the expression of constraints on the attributes of a class. Such constraints must be satisfied by every method contained within the class. That is, class invariants are used to capture integrity constraints and semantic properties that define a class [148].

With class invariants residing at the class level, it is important to know when the class invariant constraints are checked. Constraints are first checked when the creation method (constructor) for an instance of a class has finished executing. If no creation method exists for a given class, a default one with an empty body is implicitly created. In this case, when a new instance of a class is created, the constraints are immediately checked. Following the creation of an instance, the invariant constraints are checked before a method is executed, and after a method has finished executing. It is important to note that a method may invalidate the constraints during its execution, but the constraints are revalidated upon completion of the method. When an instance of a

class is destroyed, a destruction method (destructor) is called. In this case, the constraints are only checked at the beginning of the destruction method. Not having the constraints checked at the end of the destruction method allows the method to clean up any resources used, without having to worry about validating the constraints. Returning to the stack example, a corresponding class invariant would state that the value of the size attribute is always greater than or equal to zero.

Design-by-Contract (DbC) is an object-oriented design methodology composed of three elements: *preconditions*, *post-conditions*, and *class invariants*. Preconditions must be true before a method is allowed to execute and capture rules that the client (invoker) must adhere to before using a method. Post-conditions are tested at the end of method execution to determine if the method executed correctly and ensure that the client gets the result that was advertised by the server (supplier). Class invariants are used to express constraints on global properties of a class or structure. Class invariants are tested when a new instance of a class or structure is created, as well as before and after the execution of every method. Class invariants are also tested before the destruction of an instance.

DbC offers several benefits to the software process. These benefits include improved designs, improved reliability, improved documentation, and easier debugging [148]. Before continuing to the third level of contracts, we will take a look at some of the tools and technologies that implement behavioural contracts.

2.3.3 Behavioural Contract Implementation

Design-by-Contract (DbC) was first implemented in the Eiffel language [64], the use of DbC has spread to several other programming languages and notations. The following subsections will present Eiffel along with some other DbC implementations.

2.3.3.1 Eiffel

The Eiffel programming language is named after the engineer Gustave Eiffel, the creator of the famous Eiffel tower in 1887 [64]. As the Eiffel tower was completed on time and within budget, the Eiffel language was designed to do the same with software projects. Eiffel was developed by Interactive Software Engineering (ISE) in 1985. Eiffel became a mainstream language in 1988 with the release of Bertrand Meyer's seminal work on contracts [131]. Eiffel has gone through various revisions and upgrades. The current version, 6.3, contains an advanced Integrated Development Environment (IDE) and supports .NET code generation [64].

Eiffel is a fully object-oriented language, with a very verbose and English like syntax. Eiffel allows for the expression of analysis, design, and implementation all in the same language [64]. In Eiffel, all code must be placed within a class. A class is composed of features. A feature can be either an attribute or a routine. An attribute is a data item. A routine can either be a method that does or does not return a value. In Eiffel, everything is expressed in terms of behavioural contracts. Figure 9 illustrates a simple *counter* class written in Eiffel.

```
Indexing
description: "Counters that you can increment by one,
            decrement, and reset"
class interface
   COUNTER
feature -- Access
   item: INTEGER -- Counter's value.
feature -- Element change
   increment is
      -- Increase counter by one.
   ensure
      count_increased: item = old item + 1
   decrement is
      -- Decrease counter by one.
   require
      count_not_zero: item > 0
   ensure
      count_decreased: item = old item - 1
   reset is
      -- Reset counter to zero.
   ensure
      counter_is_zero: item = 0
invariant
   positive_count: item >= 0
end
```

Figure 9 - A Counter Class Written in Eiffel

As illustrated in Figure 9, a precondition is expressed by using the *require* keyword, post-conditions through the use of the *ensure* keyword. Invariants are defined using the *invariant* keyword. All aspects of Eiffel are written in one language, there is no separation between the implementation and the contracts. In Eiffel the contracts make up the implementation.

2.3.3.1.1 AutoTest

AutoTest is a fully automatic testing tool for Eiffel systems [25]. AutoTest uses the contracts found in Eiffel code for the automatic generation of test cases. For each method defined within the IUT random data is generated to simulate a method call. The resultant test cases are executed against the IUT. AutoTest tests each method

individually and does not consider the inter-method relationships (i.e. scenarios). In addition, the test driver uses random parameter data, which may not adequately test a given method. Additional details regarding how our approach differs from AutoTest and the Eiffel methodology will be presented shortly. Next we will present another tool for implementing Design-by-Contract (DbC).

2.3.3.2 Jcontract

Parasoft's Jcontract is a DbC tool [159]. Jcontract compiles Java source code that contains contracts in the form of comments. The result is Java byte code that contains checks to verify the specified contracts at runtime.

Jcontract consists of three modules. The first module is the actual compiler. The compiler generates *.class* files with extra code to check the contracts specified in the source at runtime. The second module is the runtime that checks and monitors the contracts. The third and final module is the monitor that shows the progress of the application and reports contract violations [159]. Rather than provide a tool that integrates into existing tools, Parasoft created a complete toolkit.

As the contracts are specified in comment blocks, if the developer does not wish to use the contract checking mechanism they may simply use the regular Java compiler. As with the previously discussed Spec#, the syntax for the contract is based on the implementation language. If used at a higher level of abstraction, the contracts would have to undergo a significant transformation in order to be applied to multiple implementation technologies. And as the contracts are placed in comments, they may

also interfere with actual source code comments. We will now examine another DbC tool: *iContract*.

2.3.3.3 iContract

Like Jcontract, iContract uses source comments to specify preconditions, post-conditions, and class invariants in Java programs [120]. However, unlike Jcontract, iContract uses Object Constraint Language (OCL) [188] expressions to define the contracts. Details regarding the OCL will be presented shortly. As iContract also specifies DbC elements in comments, it has the same drawbacks as those mentioned in the previous section. The motivation for placing contracts in source comments is that the source can be compiled by a compiler that does not support contracts. Conversely, our approach works after the IUT has been compiled and as such our contracts will be specified separately from the implementation.

2.3.3.4 Code Contracts

Code Contracts, a project by Microsoft Research, provide a language-agnostic way to express coding assumptions in .NET programs [134]. The contracts take the form of preconditions, post-conditions, and class invariants that are placed directly into the source code of any .NET compatible language. The contracts are evaluated at runtime and can also be used to enable static contract verification, and the generation of documentation [134]. The Code Contracts project is currently being integrated into the next version (4.0) of the .NET runtime and will be fully supported in Visual Studio 2010 [133].

Code Contracts provide another way to specify DbC constraints directly in source code. As such, Code Contracts are tightly coupled to the implementation. In addition they do not support scenarios or non-functional requirements. We will now examine some contract specification languages.

2.3.3.5 The Object Constraint Language

The OCL was first developed in 1995 during a business-modeling project within IBM [175]. The OCL was designed to be both formal and simple. The OCL is currently in its second version and has been integrated into the Object Management Group's Unified Modeling Language (UML) specification [189]. That is, the OCL is a formal language that acts as a plug-in to the UML. The OCL can be used to specify constraints and invariants on UML artifacts.

The OCL supports four types of constraints. The first three are the previously discussed Design-by-Contract (DbC) elements. The fourth is a *guard*. A guard is a constraint that must be true before a state transition fires. Figure 10 illustrates an example of an OCL invariant. The invariant defines that the pay rate of an *Employee* depends on his/her academic status. The example specifies that an *Employee* with *diploma* status has the pay rate *one*, a *doctor* has a pay rate of *two*, and finally a *professor* has a pay rate of *three*.

```
context Employee
inv payInv: ((self.status.name = 'diploma') implies
        (self.payRate = 'one')) and
   ((self.status.name = 'doctor') implies
        (self.payRate = 'two')) and
   ((self.status.name = 'professor') implies
        (self.payRate = 'three'))
```

Figure 10 - An OCL Invariant Example

Integration of the OCL with other languages and tools is not straightforward [12]. The OCL contains an ambiguous grammar that can only be parsed if structural access to the underlying system is available. The grammatical issues make it impossible to build a complete syntax tree from a standard parser generator. The OCL also defines an *allInstances* operation on each type. The *allInstances* operation is defined to return a collection of all active instances within the given type [188]. Implementation of such an operation is problematic, especially under languages that provide automatic garbage collection, such as C# or Java. In addition, the OCL does not support the modeling of interaction between UML artifacts. That is, each OCL constraint is centered on the UML artifact to which it is attached. Our previous work has developed a method to implement the *allInstances* operation for .NET languages [24]. There are several tools and compilers [26, 62, 115], including one written by this author [11, 12], for the compilation and evaluation of OCL constraints.

2.3.3.6 Alloy

Alloy is a formal constraint specification language [107, 108]. Alloy can be used as an alternative to the OCL. Alloy is a simple, precise, and tractable notation for object modeling. Alloy allows for automatic analysis through its analysis tool, Alcoa [147]. Alcoa can perform simulations and consistency checking of constraints specified in Alloy.

Alloy has several differences over the OCL. The core differences are as follows. Scalar types are treated as singleton sets. This treatment allows for sets and scalars to be viewed as the same with respect to navigation [108]. Types are implicit and are associated with various domains. A domain is defined as a set that is not declared as a subset of any other set [108]. Classes correspond to sets that are subsets of the domain, and subclasses correspond to subsets [108].

Syntactically, an Alloy specification consists of one or more paragraphs. A paragraph can be one of two types. The first is a domain paragraph; that defines all domains used in the specification. The second type of paragraph is a state paragraph. State paragraphs declare additional sets whose elements come from the domain paragraphs. The declarations may include multiplicity constraints, mutability constraints and partitions [108].

Figure 11 illustrates an example of an Alloy specification. The figure defines two domains: *Person* and *Name*. The sets *Man* and *Woman* partition the *Person* set. This means a man cannot become a woman. The *parents* relation defines that a person's parents do not change. The *siblings* relation is defined as being many-to-many, while the *wife* relation states: one man to at most one woman. *Name* relates a person to a unique name. For more information on Alloy please see [107, 108, 147].

```
model Family {
   domain{Person, Name}
   state {
      partition Man, Woman: static Person
      Married: Person
      parents: Person -> static Person
      siblings: Person -> Person
      wife(~husband): Man? -> Woman?
      name: Person -> Name!
   }
   def siblings {
      // Two people have the same siblings if an only if
      // they have the same parents
      all a, b | a in b.siblings <-> (a.parents = b.parents)
   }
   inv Basics {
      // A person has a wife if and only if he
      // is a man and is married
      all p | some p.wife <-> p in Man & Married
      // No persons wife is also a sibling
      no p | p.wife in p.siblings
      // A person has at most one father and at most one mother
      all p | (sole p.parents & Man) && (sole p.parents & Woman)
      // No person is an ancestor of him/herself
      No p | p in p.+parents
   }
}
```

Figure 11 - Example Contract Specification using Alloy

We have briefly discussed some of the tools and technologies that implement behavioural contracts. In addition to the tools and technologies presented in the previous sections, additional notations include B [1], the Java Modeling Language (JML) [122], VDM [111], and Z [102]. The additional notations do not add any additional semantics, but rather are notations for the specification of behavioural contracts. Our approach uses a contract that represents a testable model. Our contract is specified separately from the implementation and uses a general-purpose high-level contract specification language known as Another Contract Language (ACL) [8]. The ACL will be presented in Chapter 3. The ACL is able to express both the required structural

elements presented in the basic contract level, and the behavioural contracts specified in the second level. That is, our framework completely supports the use of Design-by-Contract (DbC) elements within a contract. Details regarding the specification and execution of such elements will be presented in Chapter 4. All of the previously discussed contract methodologies do not extend pass level two. That is, they specify structural and DbC constraints but nothing more. We will now present the third contract level: *Synchronization Contracts*.

2.3.4 Synchronization Contracts

Behavioural contracts treat the body of a method as an atomic action. That is, preconditions are used to verify the parameters and object state before the method executes and post-conditions are used to verify correct method execution. However, the actual execution behaviour of the method is not specified or verified. Synchronization contracts are used to specify the global behaviour of objects in terms of synchronizations between method calls [35]. The aim of such a contract is to describe the dependencies between services provided by a component, such as a sequence or parallelism.

The body of a method can be seen as a sequence of events that occur in a given order to accomplish the goal of the method. These events could include the invocation of other methods or assignment of instance variables. A simple synchronization contract would specify the order in which such events would occur. Just specifying the order of the execution of statements in a single method body is not sufficient. Instead the synchronization contract specifies constraints across the global behaviour of the

system. Such contracts can indicate methods that can execute in parallel, and methods that are a prerequisite for other methods.

Our framework supports the specification and verification of synchronization contracts through the use of scenarios. Our scenarios specify a grammar of responsibilities (corresponding to methods) that represent the global behaviour of the system. Using some of the operators defined by Ryser and Glinz in SCENT [168], we are able to capture concurrency in our scenarios. Using the synchronization contracts along with basic and behavioural contracts, we are able to express, qualify, and contractually define all the behavioural properties of a software system. However, we do not have a way of specifying quality of service constraints within our contracts. That is, up to this point, we have looked at a contract model for the specification of functional requirements, but not non-functional requirements. Recall that the requirements for our testable model include both functional and non-functional requirements. So we need one final level of contracts: *Quality of Service Contracts*.

2.3.5 Quality of Service Contracts

Quality of service contracts deal with the non-functional aspects of a system [35]. These non-functional aspects can include performance, availability, usability, and security. Non-functional aspects of a system can be acquired from a process known as goal modeling [125, 162, 205]. Goal modeling analysis begins with softgoals that represent non-functional requirements that the system stakeholders agree upon [205]. Such analysis can be performed using the Goal-oriented Requirement Language (GRL) [152, 205].

Our framework provides support for the specification, capture, and analysis of quality of service contracts. Dedicated *metric* contract sections are used to capture and store quality of service for the IUT, and then a *reports* contract section analyzes the results, as will be illustrated in the next chapter. Details of how quality of service aspects are handled by our framework will be presented in Chapter 4.

2.3.6 Summary

In order to support contract-based validation, a contract must exist that is able to capture the entirety of the testable model. We have presented four contract levels that compose the required contract. The first level, basic contracts, provides what must be done to make the system work. Such a contract translates into the structural composition of the IUT. At the first level, the contract consists of an interface indicating method names and parameter types. The second level, behavioural contracts, is the most commonly used level. A behavioural contract is specified using Design-by-Contract (DbC) constructs. Such constructs include preconditions, post-conditions, and invariants. We have presented notations and tools that support the specification and execution of behavioural contracts, most notably Eiffel. Eiffel is an environment designed for the specification and evaluation of behavioural contracts as a means of implementing a software system. Going to level three we find synchronization contracts that are used to specify the global interaction between methods, and the sequence of events, or scenario, that defines the behaviour of a method. Finally, at level four we add quality of service contracts to represent non-functional requirements.

The aim of our framework has been the creation of a contract that is complete enough to represent a testable model that can be executed against an IUT to determine validation. In order for such a contract to be created, our framework supports each of the previously defined contract levels in their entirety. It should be noted, that our contract specification language, Another Contract Language (ACL), will contain a single notation for the specification of all contract levels. That is, the contract writer will not have to write a contract at level one, level two, and so on, but rather a single contract covering all levels. The syntax and semantics of such a contract will be presented as an example in the next chapter.

2.4 A Summary Overview of Existing Work

Table 1 contains a summary of each tool and technology presented in this chapter. The table consists of five columns. The first column indicates the general technology category. The second column provides some concrete tools that implement the given technology. Next, the category of technology is denoted. Valid types include code-based approaches, model-based approaches and contract-based approaches. The fourth column provides a point-form summary of the given tool or technology with respect to how it is similar or different from our approach. For a more detailed summary, see the appropriate section in this chapter or one of the references found in the final column of the table.

Technology	Tools	Category	Summary	References
Unit Testing	JUnit, CppUnit	Code	- Tests are expressed directly using source code, no abstraction. - Tests each method in a vacuum. - Tightly coupled to the implementation technology. - Requires knowledge of low-level implementation details for the creation of tests. - Permits testing before development of the candidate IUT is complete.	[30, 31, 55, 71, 93, 112, 156, 164]
Static Pattern Matching	FindBugs, FXCop, PREFast	Code	- Static analysis tools that rely on implementation level patterns to determine bugs. - Patterns are specified at either the code- or byte-level. - Can result in the generation of false positives. - Patterns can be reused, but are implementation technology-specific.	[69,89, 135, 136, 138]
Static Checking	Splint, Lint, Jlint	Code	- Code analysis to determine bugs, security, or portability constraints. - Tightly coupled to the implementation technology. - Can result in the generation of false positives. - Uses source code comments to drive the static checker. - Requires source code of the candidate IUT.	[56, 66, 110, 117, 126]
The Unified Modeling Language	IBM Rational Rose, Telelogic Tau	Model	- Dominant notation for the specification of models. - MBT is not directly supported, requires additional tools/technology. - Implementation independent. - If test cases are generated, they are not directly executable.	[34, 92, 178, 189]
Lustre	Lutess	Model	- Lustre is a programming language for reactive systems centralized around time-synchronization and dataflow orientation. - Lutess allows for the test sequence to be executed against an IUT. - Requires an oracle to determine correctness. - Cannot handle liveness or numerical inputs or outputs.	[41, 49, 79]
Lustre	Lurette	Model	- Allows for the inclusion of IUTs that have numerical inputs and outputs. - Only operates on IUTs written using C. - Requires the IUT to implement a set of predefined methods. - Requires an oracle to determine correctness. - Cannot handle liveness.	[165]
Lustre	GATeL	Model	- Uses constraint logic programming for the specification of state invariants and path predicates. - Starts from the last sequence and works backwards. - Finds a test sequence that satisfies both the constraint logic programming constraints and the initial test purpose. - Requires an oracle to determine correctness. - Cannot handle liveness.	[60, 127, 154, 181]
Extended Finite State Machines	AutoFocus	Model	- Uses an EFSM model of the IUT, the IUT, and a test case specification. - The test case specification can be functional, structural or stochastic. - Features are tested by arriving at their state. - Requires the developer to provide an EFSM of the IUT. - Tests are generated by traversal of the EFSM, and constraint logic programming constraints. - Execution is performed symbolically. - Does not scale due to state explosion problem.	[90, 163]

Tree and Tabular Combined Notation 3	Conformance Kit	Model	- Designed for testing protocol implementations. - Textual (TFT) and Graphical (GFT) notations. - Only supports static test interfaces and is tightly coupled to the IUT. - Includes ASN.1 notation for expression of data structures. - No support for non-functional requirements.	[61, 94, 100, 101, 103, 104, 173]
Tree and Tabular Combined Notation 2	PHACT	Model	- Able to execute TTCN-2 test cases against an IUT. - Requires the creation of an implementation-specific supervisor, stimulator and observer (glue code). - Recompiles the IUT and glue code together to produce a single executable tester. - Supports C and Java. - Not publically available.	[82, 97, 149, 160]
Labeled Transition System	TorX	Model	- Can be used with any specification language that is expressed as a labeled transition system. - User is required to provide IUT connection. - Each IUT much have a dedicated connection (glue code).	[193, 194]
N/A	AGEDIS	Model	- Represents a methodology for the automation of software testing. - Allows for the generation and execution of test cases. - Supports C, C++, and Java. - Developers implement interfaces to specify test generation algorithms. - Does not support objects, static systems only. - Only tests functional requirements.	[3, 81]
Abstract State Machines	AsmL	Model	- Transforms an ASM representation into a FSM and then uses FSM-based algorithms to generate a test suite. - Suffers from the state space explosion problem. - Can do some pruning to reduce the state space. - Cannot execute the generated test suite. - Cannot handle non-deterministic FSMs.	[27, 77, 78, 139]
Spec#	Spec Explorer	Model	- Takes a Spec# specification and executes the generated tests against an IUT. - No glue code required. - Supports design-by-contract elements. - Uses a FSM internally for model representation. - Suffers from the state explosion problem. - Does not support scenario modeling. - Only tests functional requirements. - Current state-of-the-art in model-based testing.	[28, 46, 47, 48, 145, 146]
Eiffel	AutoTest	Contract	- The contracts and implementation are specified as one. - Automated testing is supported only at the method level. - During regular execution of the IUT the Design-by-Contract (DbC) elements are tested. - Does not support scenario modeling. - Only tests functional requirements.	[25, 64, 131, 148]
Jcontract	Jcontract	Contract	- Java specific DbC tool. - Contracts are specified as comments in Java source files. - Specialized compiler compiles the Java source and contracts together to create an instrumented binary. - Execution of the IUT executes the contracts. - Does not support scenario modeling. - Only tests functional requirements.	[159]
Code Contracts	Visual Studio 2010	Contract	- .NET specific DbC tool. - Contracts are specified directly in any .NET language. - The next version of the .NET runtime will have directly support. - Does not support scenario modeling. - Only tests functional requirements.	[133, 134]

The Object Constraint Language	iContract, The C# / OCL Compiler, The Octopus Tool, The Dresden OCL Toolkit, etc.	Contract	- Contacts are placed on UML artifacts. - Specialized tools are required to transform the OCL so that it can be executed. - The language is not easily adapted to implementation level. - Does not support scenario modeling. - Only tests functional requirements.	[11, 12, 24, 26, 62, 115, 120, 188]
Alloy	Alcoa	Contract	- An alternative to the OCL. - Contracts are specified all together in a single location. - To abstract for direct IUT integration. - Alcoa is a constraint analyzer, does not support direct execution. - Does not support scenario modeling. - Only tests functional requirements.	[107, 108, 147]
Scenario Modeling (SCENT)	None in existence	Contract	- Allows for the specification of method interactions. - Captures scenarios modeling functional requirements. - Includes concurrency and synchronization operators. - No known implementation of the technology.	[168]
Goal Requirements Language	Only editors and domain-specific execution environments	Contract	- Allows for the specification of non-functional requirements. - Uses a graphical goal-oriented approach. - Decoupled from the IUT. - No direct execution is possible, except in specific domains.	[152, 205]

Table 1 - Overview of Existing Work

This chapter has provided relevant background pertaining to our framework. The chapter started with a look into the current state of code-based testing, and supporting tools. We continued with model-based testing, concluding with the presentation of Spec#, a research project at Microsoft aimed at creating a more cost effective way to develop and maintain high-quality software. Our review continued with a look at contract-based testing and how contracts are handled in Eiffel. Moving deeper into the four contract levels, we mentioned SCENT, a methodology for expressing scenario relationships, and finally with the GRL that allows for the capture of non-functional requirements. We now turn to Chapter 3 to begin a detailed example modeling a container.

3 Example Walkthrough

In order to provide an overview of our approach, we will present an example of a simple container that we will validate. Our example begins with the elicitation of a Testable Requirements Model (TRM) expressed using our Another Contract Language (ACL). Next, a suitable Implementation Under Test (IUT) will be presented as a concrete realization of the TRM. A set of bindings will be automatically generated to connect the implementation independent TRM to the concrete candidate IUT. With the bindings in place, our Validation Framework (VF) will execute the TRM against the IUT. The result of such execution is a Contract Evaluation Report (CER). The report displays the results of executing a TRM against a candidate IUT.

Section 3.1 presents the initial incarnation of our container example, one that requires all elements within the container to be unique. Section 3.2 introduces a revised container example allowing for multiple occurrences of the same item. In addition, each section will use a unique IUT in order to illustrate how multiple IUTs can be applied to a single TRM. Finally, Section 3.3 outlines and provides references to four supplementary validated case studies we have created.

3.1 The Container Example Version 1: No Multiple Occurrences

Our example begins with the creation of a Testable Requirements Model (TRM) for the container. The TRM is expressed in a language known as the ACL. The container we are modeling is strongly-typed. That is, the container stores a single type of item. Each item will encapsulate an integer value. The container supports standard operations such as add and remove. In the first incarnation of our example, we will not allow

duplicate items to be stored within the container. Following the creation of a TRM for the container, an IUT will be presented that can be bound and executed against the TRM.

3.1.1 The Testable Requirements Model

As previously mentioned, scenario-driven approaches to Model-Based Testing (MBT) do exist. Few, however, consider inter-scenario relationships [168] in the process of generating test cases. Among those who do, Nebut et al. [153] propose algorithms for test case generation from use-case like scenarios enhanced with contracts, that is, with pre- and post-conditions [130]. They then argue that contrary to diagrammatic approaches to the sequential ordering of scenarios, such as Briand and Labiche's use of UML activity diagrams [43], contracts are scalable (especially with respect to production and readability). These authors also demonstrate that coverage strategies akin to those of state-based approaches [36] are readily available for their scenario contracts. In other words, the automated generation of test cases from scenario-based models (including those using contracts) is entirely feasible.[4]

It is crucial to understand that, as with most other MBT approaches, the test cases generated from the algorithms put forth by Nebut et al. are totally disconnected from any implementation. To the best of our knowledge, with the notable exception of Grieskamp's state-based work at Microsoft [76], no one has succeeded in building a MBT framework that generates fully instrumented test cases.

[4] Though difficult problems, such as the combinatorial explosion of the number of scenarios paths remain.

Our TRM proceeds from the scenario contracts proposed by Nebut et al. [153]. Whereas they simply augment scenarios with pre- and post-conditions, we further enhance scenarios and responsibilities with static checks, dynamic checks, and metric evaluators in order to define a system's contracts. Static checks correspond to structural queries on an IUT, whereas dynamic checks and metric evaluators query an execution of an IUT. Details regarding the execution of each type of check and evaluator will be provided in Chapter 4.

The premise of our approach is that the test cases generated from models (such as a TRM) are decoupled from any particular implementation. Thus a single set of contracts, or TRM, can be tested against several candidate implementations. This is in sharp contrast with the code-centric approaches to testing where code is the starting point for obtaining tests. The following sections will present three contracts and one interaction to illustrate some key semantic elements of our TRM.

3.1.1.1 The Item Contract

The first element of our TRM is a simple contract that will be used to represent each item stored within the container. The *Item* contract listing is shown in Figure 12. The *Item* contract begins with an import statement. Import statements allow for the inclusion of external checks and metric evaluators into the current contract listing. Additional information on checks and evaluators will be provided shortly. A namespace declaration follows. Each contract and interaction must reside within a namespace. In the container example, all elements of our TRM will be located within the *DaveArnold.Examples* namespace. As with object-oriented programming languages,

namespaces allow for the logical grouping of elements. In the case of Figure 12, a single contract named *Item* is declared within the namespace.

```
Import Core;

Namespace DaveArnold.Examples
{
   Contract Item
   {
      Observability Scalar Integer Value();

      Responsibility new()
      {
         Post(Value() >= 0);
      }

      Responsibility finalize()
      {
         Pre(Value() >= 0);
      }
   }
}
```

Figure 12 - Item Contract

Contracts are one of two first order entities that can be used for the specification of a TRM. A contract provides a way to package a set of responsibilities and scenarios into a logical grouping. Practically, a contract is connected to a type within the IUT. That is, a contract contains elements that are to be applied to a single IUT type. Such application is performed using the notion of bindings and will be discussed later in this chapter. An interaction provides a way to specify the interaction between several contracts. That is, an interaction allows for the specification of inter-scenario relationships. The container TRM contains an interaction that will be presented shortly.

The *Item* contract consists of three sections. The first section named *Value* is an observability. Observabilities are query-methods that are used to provide state information about the IUT during execution. That is, they are read-only methods that

acquire and return a value stored within the IUT. Each observability defines an observation requirement imposed on the IUT for contract execution. The *Value* observability is used to return a scalar (non-list) integer value that will represent the actual value stored by this element of the container. The next section, *new*, is a responsibility. A responsibility represents a task or functional requirement that the IUT must implement. The *new* responsibility is a special responsibility that specifies a set of checks that must hold immediately after the creation of a new contract instance. The *new* responsibility is analogous to a constructor found in most object-oriented programming languages. The body of the *new* responsibility defines a single post-condition, using the *Post* keyword, to ensure that the integer value returned by the *Value* observability is greater-than or equal to zero. That is, each element stored within the container must represent a positive value, a requirement that is purely illustrative. The final section, *finalize*, is also a special responsibility that specifies a set of checks that must hold immediately before the contract instance is destroyed. The *finalize* responsibility is analogous to a destructor. Here, the body of the *finalize* responsibility defines a single precondition, using the *Pre* keyword, to ensure that the integer value returned by the observability is still greater-than or equal to zero. The *finalize* responsibility completes our *Item* contract. We will now define the first of two contracts that will define the TRM for the container itself.

3.1.1.2 The ContainerBase Contract

As the use of containers is common in the development of software systems, we will first create a reusable abstract container contact. Then in Section 3.1.1.3 this abstract

container will be refined into a concrete container for use within our example. Using such a base container also illustrates the notion of contract generics and inheritance supported by the ACL. The *ContainerBase* contract is shown in Figure 13.

```
Import Core;

Namespace DaveArnold.Examples
{
   abstract Contract ContainerBase<Type T>
   {
      Scalar Integer size;

      Observability Boolean     IsFull();
      Observability Boolean     IsEmpty();
      Observability T           ItemAt(Integer index);
      Observability Integer     Size();
      abstract Observability Boolean  HasItem(T aItem);

      Responsibility new()
      {
         size = 0;
         Post(IsEmpty() == true);
      }

      Responsibility finalize()
      {
         Pre(IsEmpty() == true);
      }

      Invariant SizeCheck
      {
         Check(context.size >= 0);
         Check(context.size == Size());
      }

      Responsibility GenericAddition(T aItem)
      {
         Pre(aItem not= null);
         Pre(IsFull() == false);
         Execute();
         size = size + 1;
         Belief InCollection("The item should now be in the container")
         {
            Post(HasItem(aItem));
         }
      }
```

```
        Responsibility Add(T aItem) extends GenericAddition(aItem)
        {
            Execute();
        }

        Responsibility Insert(Integer index, T aItem)
            extends GenericAddition(aItem)
        {
            Pre(index >= 0);
            Execute();
            Post(ItemAt(index) == aItem);
        }

        Responsibility T Remove()
        {
            Pre(IsEmpty() == false);
            Execute();
            size = size - 1;
            Post(value not= null);
            Post(HasItem(value) == false);
        }

        Responsibility RemoveElement(T aItem)
        {
            Pre(IsEmpty() == false);
            Pre(HasItem(aItem) == true);
            Execute();
            size = size - 1;
            Post(HasItem(aItem) == false);
        }

        Scenario AddAndRemove
        {
            once Scalar T x;
            Trigger(Add(x) | Insert(dontcare, x)),
            Terminate((x == Remove()) | (RemoveElement(x)));
        }
    }
}
```

Figure 13 - ContainerBase Contract

After the standard import and namespace declaration, the *ContainerBase* contract is declared. The *abstract* modifier indicates that our contract is abstract and cannot be directly bound to a type within an IUT. Abstract contracts must be refined into a derived contract. The *ContainerBase* contract is also a generic contract. Generic contracts are specified by using a list of generic parameters following the name of the contract, enclosed by a pair of matching angle brackets. Contract generics allow for the

creation of general purpose template contracts that can be reused across multiple domains. The *ContainerBase* contract contains a single generic parameter named *T* that will represent a type within the IUT. That is, *T* will represent the element type stored within the container.

The body of the *ContainerBase* contract begins with the declaration of a single contract variable. Contract variables are analogous to instance variables found in object-oriented programming languages. Each contract instance has its own set of contract variables. It should be noted that contract variables are independent from any variables found within the IUT. A contract variable provides a way for a contract instance to store and exchange information between responsibilities, scenarios and metrics. Our only contract variable is named *size*. The *size* contract variable will be used to store the number of items that should be in the container. That is, the number of items that should be stored in the container as per the TRM, rather than the number of items actually stored by the IUT. The contract variable declaration indicates that the *size* contract variable is a scalar integer value. In the Another Contract Language (ACL), variables can be defined to be either a scalar or list value. A scalar value represents a single value, and a list value represents a collection of values. Variables declared as a list automatically have a set of twelve pre-defined list operations that can be applied to the variable. These operations include the insertion, search, and removal of items. Details regarding the twelve pre-defined list operations can be found in the ACL specification document [8]. In short, the ACL contains built-in list types. The list or scalar modifier is optional, the scalar modifier is assumed when no modifier is specified.

Immediately following the list or scalar modifier, the variable type must be specified. In the case of the *size* variable, the *Integer* built-in type is used. Variables can be declared using one of the eight built-in types or by using an exported type that is defined within the IUT. Examples of both will be given in this chapter.

Following the contract variable declaration, five observability methods are defined. Recall that an observability method represents an observation requirement placed on the IUT. That is, each non-abstract observability method will be bound to an IUT query-method that will provide the required IUT state information. Our first observability method is named *IsFull* and will be bound to an IUT method that will return true if the container is full, and false otherwise. The *IsEmpty* observability will provide similar information indicating if the container contains any items or not. The *ItemAt* observability is used to get the item being stored at the given index. The observability accepts a single scalar integer value representing the index, and returns the actual element stored by the IUT. As the Testable Requirements Model (TRM) is not tied to any particular IUT, the type of the return value is represented by an identifier: *T*. The *T* identifier is the previously discussed generic parameter and acts as a placeholder for the real element type used within the IUT. Next, the *Size* observability is defined. The *Size* observability will return the number items actually stored by the container. That is, it will be bound to a method that returns the number of items currently stored by the IUT. Finally, the *HasItem* observability will be used to determine if a given element resides within the container. The observability will return true if the element, represented by the variable *aItem* of type *T*, is in the container, false otherwise. Unlike the previously

defined observabilities, *HasItem* is marked abstract. Abstract observabilities are not bound to an IUT counterpart, but rather must be defined by any contract that refines the *ContainerBase* contract. Such refinement will be shown shortly.

After the observability definitions, the *new* responsibility is defined. Recall that the body of the *new* responsibility is executed immediately following the creation of a new contract instance. The *new* responsibility in the *ContainerBase* contract begins by setting the *size* contract variable to zero, as we have yet to add any items to the container. A post-condition is used to ensure that the container does not start with any items. The body of the post-condition uses the *IsEmpty* observability to determine if the container is empty.

Analogous to the *new* responsibility, the *finalize* responsibility is defined. Recall that the body of the *finalize* responsibility is executed immediately before destruction of the current contract instance. Contract instances are destroyed when the IUT instance to which the contract instance is bound is executing its destructor.[5] The body of the *finalize* responsibility contains a single precondition to ensure that the container is empty prior to destruction. That is, our TRM does not allow for the destruction of a container that still contains items.

Following the *new* and *finalize* responsibilities, an invariant named *SizeCheck* is defined. Invariants provide a way to specify a set of checks that are to be executed

[5] The *finalize* responsibility is called upon entry to the corresponding destructor.

before and after the execution of all bound responsibilities[6] defined within the containing contract (including any responsibilities inherited from base contracts). The execution of invariants follow design-by-contract [130] rules, in that, invariants are executed upon entry and exit of every bound responsibility with two exceptions. The invariant is not executed upon entry to the *new* responsibility, and upon exit of the *finalize* responsibility. The exceptions allow for the state of both the IUT and contract to be invalidated during instance construction and destruction. Each invariant defined within a given contract must have a unique name that is specified following the *Invariant* keyword. This name is used for reporting the result of invariant execution and has no other meaning. The body of the *SizeCheck* invariant defines two checks, via the *Check* keyword. The first ensures that the *size* contract variable has a positive value. The second check compares the *size* contract variable with the actual number of items stored by the IUT. That is, the check ensures that the number of items that the container is supposed to be storing, as per the TRM, matches the number of items actually being stored by the IUT. Note that the *size* contract variable is accessed using the *context* keyword. The *context* keyword is analogous to the *this* keyword found in C++ or Java. It is used to disambiguate between local contract scope and contract instance scope, or to get a reference to the current contract instance. In the case of the *SizeCheck* invariant, the *context* keyword is not required, but we have included it to illustrate its use.

[6] A bound responsibility is one that is directly mapped to a method within the IUT. Invariants are not executed on unbound responsibilities as they represent a grammar of responsibilities that will flatten to a sequence of bound responsibilities that have invariants. That is, the invariant will be executed anyway, so for performance reasons the invariant is not executed on unbound responsibilities.

Next, the *GenericAddition* responsibility is defined. The *GenericAddition* responsibility will be used as a template for all responsibilities that add a new item to the container. The item to add is represented by the single parameter passed to the responsibility. The parameter is of type *T* and is named *aItem*. When the contents of the *GenericAddition* responsibility are executed by the VF runtime, the *aItem* parameter will be assigned the value of the actual item being added to the container. The body of the *GenericAddition* responsibility is divided into two parts. The parts are separated by an execute statement, denoted by *Execute()*. Statements appearing above the execute statement are evaluated immediately before the IUT method bound to the responsibility is executed. Statements appearing after the execute statement are evaluated immediately after the IUT method bound to the responsibility has finished executing. It should be noted that the *new* and *finalize* responsibilities do not use the execute statement. Such usage will result in a compile-time error. This is because the *new* responsibility executed after the creation of a new contract instance, and the *finalize* responsibility is executed before the destruction of a contract instance. The body of the *GenericAddition* responsibility begins with two preconditions. Preconditions must be placed before the execute statement. The first precondition checks to make sure that the item being added to the container is a real item, and is not null. Our Testable Requirements Model (TRM) indicates that the container is not able to store empty (null) items. The second precondition checks to make sure that the container is not already full. That is, the *IsFull* observability returns false. Next, the execute statement is used to inform the VF runtime that the bound IUT method should execute.

Once the IUT method has finished executing, the remainder of the responsibility's body is executed. First the *size* contract variable is incremented to reflect the addition of a new item to the container. Finally, a belief is specified indicating that the new item should now be in the container. Beliefs can be placed in most contract sections and represent a user-friendly name that applies to a group of statements (the body of the belief). If all statements contained within the body of a belief succeed, then the belief is said to succeed, otherwise the belief fails. Our belief is named *InCollection* and has a description indicating that the new item should now be part of the collection. The body of the belief is composed of a single post-condition that uses the *HasItem* observability to ensure that the item has actually been added to the container. If the *HasItem* observability returns true then the post-condition succeeds, as does the containing belief. If the observability returns false, then the post-condition fails, along with the containing belief. As the *GenericAddition* responsibility represents the generic add item behaviour we will now look at two responsibilities that extend from *GenericAddition*: *Add* and *Insert*.

The *Add* responsibility performs the task of adding a new item to the container. Our TRM does not enforce a specific addition algorithm. We will use the behaviour provided in the body of the *GenericAddition* responsibility for the *Add* responsibility. Such inclusion is performed through the notion of responsibility extension. Each bound responsibility may extend any number of other responsibilities defined either within the same contract or within a parent contract. Responsibility level extension, allows for a set of checks to be used across multiple responsibilities. The contents of the

GenericAddition responsibility will be added to the body of the *Add* responsibility as follows. Code placed before the execute statement is inserted at the beginning of the body of the *Add* responsibility. Code placed after the execute statement will be inserted immediately after the execute statement found in the *Add* responsibility. If the *Add* responsibility does not contain an execute statement, a compile-time error will be generated. As the *GenericAddition* responsibility contains all behaviour required for the addition of a new item to the container, no other statements are required in the body of the *Add* responsibility.

The *Insert* responsibility represents the task of adding a new item to the container at a specific location. Again we will extend from the *GenericAddition* responsibility to provide the basic addition behaviour. As we are inserting at a specific location, we will add two checks to the body of the *Insert* responsibility. The first is a precondition to ensure that the index where the item is to be inserted is greater-than or equal to zero. The second, a post-condition to ensure that after the actual insertion, the item has been inserted at the requested location.

The *Remove* responsibility removes an item from the container, and returns it to the caller. The actual item to be removed is determined by the bound IUT method. That is, the TRM does not enforce a specific item removal algorithm. The return type is specified after the *Responsibility* keyword and before the name of the responsibility. In our case the return type is *T*. The responsibility will be bound to an IUT method that will perform the task of removing an item from the container. The body of the *Remove*

responsibility begins with a precondition to ensure the container is not empty. That is, we are not trying to remove an item from an empty container. Next the execute statement is used to instruct the IUT to actually remove an item. Once the IUT has finished removing the item, the *size* contract variable is decremented to reflect the removal. Two post-conditions follow, the first ensures that a valid item was removed from the container, by making sure that the item is not null. Recall that we did not allow empty (null) items to be added to the container. The actual item that was removed from the container can be accessed using the *value* keyword. The *value* keyword represents the return value from the IUT method that is bound to the *Remove* responsibility. The second post-condition ensures that the item is no longer stored in the container.

Next, the *RemoveElement* responsibility represents the task of removing a specific item from the container. The item to remove is provided as the lone parameter to the responsibility. The responsibility begins with two preconditions, the first ensures that the container is not empty, and the second checks to see if the item to remove is currently being stored within the container. That is, we are not trying to remove an item that is not in the container. Next, the execute statement is used to remove the given item by executing the IUT method bound to the responsibility. Once the IUT method has finished executing, the *size* contract variable is decremented to reflect removal of the item. Finally, a single post-condition is specified to ensure that the requested item was actually removed from the container.

Responsibilities represent an individual task or functional unit within the Testable Requirements Model (TRM). We can model the interaction and ordering of responsibility execution through the use of a scenario. A scenario is defined by a triggering event, an optional execution grammar, and a termination event. The triggering event is usually the execution of a bound responsibility, but it can be other things such as the observation of an event. The triggering event is used to denote the beginning of the scenario, where the termination event is used to denote the end. After the triggering event occurs within the TRM, a new instance of the given scenario is created. The scenario instance exists until the specified execution grammar has executed followed by the termination event. Each time the triggering event occurs a new scenario instance is created. That is, a scenario can have any number of active scenario instances. Scenarios are identified by their names, and do not accept parameters or return values. The *AddAndRemove* scenario will model the lifespan of each element stored within the container. The scenario body begins with the declaration of a scenario variable named *x*. The variable is scalar and of type *T*. Notice the use of the *once* modifier. The *once* modifier indicates that the scenario variable can only be assigned to once. An additional assignment will result in a runtime error. The scenario begins with a triggering event, denoted by the *Trigger* keyword. The triggering event is either the execution of the previously discussed *Add* responsibility or the *Insert* responsibility, but not both. The element to insert for both responsibilities is the *x* scenario variable. As the scenario variable has yet to be assigned, following execution of the *Add* or *Insert* responsibility, the scenario variable will automatically be assigned the

item that was added to the container. That is, a new scenario instance will be created for each unique item added to our container. Notice the use of the *dontcare* keyword to substitute the index parameter passed to the *Insert* responsibility. The *dontcare* keyword states that the value of the index parameter is not important from the viewpoint of the scenario. The *AddAndRemove* scenario does not contain an execution grammar, so after creation the scenario instance simply waits for the termination event to occur. The termination event, denoted by the *Terminate* keyword, occurs when the element that was added to the container is removed. There are two ways to remove an element from the container. The first is if the *Remove* responsibility returns the item that we added, and the second is if the *RemoveElement* responsibility is executed with our element, *x*, as the parameter. Any scenarios that are still in progress (i.e., the termination event has yet to occur), when the IUT completes its execution, fail. The *AddAndRemove* responsibility completes the definition of the *ContainerBase* abstract contract. We will now define a concrete contract that refines *ContainerBase*.

3.1.1.3 The Container Contract

The *Container* contract, shown in Figure 14 specializes the *ContainerBase* contract defined in the previous section. In addition to refining the functional aspects found in the *ContainerBase* contract, we will also add some non-functional aspects to the *Container* contract. The *Container* contract begins with the *MainContract* keyword as opposed to the *Contract* keyword that was used to declare the *Item* and *ContainerBase* contracts. Both contract types have the same semantics, with the exception of how each contract is bound to an IUT type. Contracts defined using the *Contract* keyword

are only bound if used within another contract that is being bound. Contracts defined using the *MainContract* keyword must be bound to a type within the IUT. Each TRM must contain at least one contract defined using the *MainContract* keyword. For additional details on the *Contract* and *MainContract* keywords see the Another Contract Language (ACL) specification document [8]. The *Container* contract also specializes the *ContainerBase* contract via the *extends* keyword. That is, the ACL supports contract inheritance. Any number of base contracts can be specified following the *extends* keyword, as multiple inheritance is supported.

```
Import Core;

Namespace DaveArnold.Examples
{
   MainContract Container extends ContainerBase<tItem>
   {
      List Integer container_times;
      Scalar Timer item_timer;
      Scalar Integer number_of_items;

      refine Observability Boolean HasItem(tItem item)
      {
         tItem x;
         Boolean result = false;
         loop(0 to Size())
         {
            x = ItemAt(counter);
            result = result || x == item;
         }
         value = result;
      }

      Parameters
      {
         Scalar Boolean CheckMembers = true;
      }
```

```
Structure
{
    choice(Parameters.CheckMembers) == true
    {
        Belief CheckMember(
            "There should be a member of our container")
        {
                HasMemberOfType(tItem);
        }
    }
}

refine Responsibility new()
{
    number_of_items = 0;
    container_times.Init();
}

refine Responsibility finalize()
{
    fire(ContainerDone);
}

refine Responsibility Add(tItem item)
{
    Pre(HasItem(item) == false);
    Execute();
    item_timer.Start(item);
    number_of_items = number_of_items + 1;
}

refine Responsibility Insert(Integer index, tItem item)
{
    Pre(HasItem(item) == false);
    Execute();
    item_timer.Start(item);
    number_of_items = number_of_items + 1;
}

refine Responsibility tItem Remove()
{
    Execute();
    item_timer.Stop(value);
    container_times.Add(item_timer.Value(value));
}

refine Responsibility RemoveElement(tItem item)
{
    Execute();
    item_timer.Stop(item);
    container_times.Add(item_timer.Value(item));
}
```

```
Responsibility RemoveScn()
{
    Remove() | RemoveElement(dontcare);
}

stub Responsibility AddElement(tItem item)
{
    Pre(item not= null);
    [Default] Add(item);
}

Scenario ContainerLifetime
{
    Trigger(new()),
    atomic
    {
        (Add(dontcare) | Insert(dontcare, dontcare))*,
        (RemoveScn())*;
    },
    observe(ContainerDone),
    Terminate(finalize());
}

Metric List Integer ContainerTimes()
{
    context.container_times;
}

Metric Scalar Integer NumberOfItems()
{
    context.number_of_items;
}

Reports
{
    Report(
    "The average time in the container is {0} milliseconds",
    AvgMetric(ContainerTimes()));

    ReportAll(
    "The average time in all containers is {0} milliseconds",
    AvgMetric(ContainerTimes()));

    Report("The number of items added to the container is {0}",
    NumberOfItems());

    ReportAll(
    "The number of items added to all containers is {0}",
    NumberOfItems());
}
```

```
        Exports
        {
            Type tItem conforms Item
            {
                not context;
                not derived context;
            }
        }
    }
}
```

Figure 14 - Container Contract

As the *ContainerBase* contract is a generic contract, a value for the single generic parameter is required. The generic parameter value is specified by the *tItem* symbol. The *tItem* symbol acts as a placeholder for the IUT type that represents the element type stored within the container. The symbol itself will be explicitly defined at the end of the *Container* contract.

The body of the *Container* contract begins with the declaration of three contract variables. The first, named *container_times*, is a list of integers that will be used to store the amount of time each element spends in the container. We will use this information for reporting non-functional metrics about our container. Next, a scalar timer named *item_timer* is declared. The ACL supports timers as a built-in type. In fact, a single timer can be used to time multiple items concurrently, as will be illustrated shortly. Finally, a scalar integer named *number_of_items* is declared. The *number_of_items* variable will be used to store the total number of items that are stored by the container during execution.

The *Container* contract will automatically inherit the five observabilities defined in the *ContainerBase* contract. Recall that the *HasItem* observability method was defined

using the *abstract* modifier. As such, we must refine it here. We have two choices for refinement. The first is to bind the observability to a query-method within the IUT. The second is to provide a body for the observability that calculates the requested value. The second choice is useful when the IUT does not provide the requested query-method. Here, we will provide a body to refine the *HasItem* observability. The body of the observability method begins with the declaration of a variable named *x*. The variable is of the *tItem* type, which is the type of item stored by the container. In addition, a variable of type Boolean is declared that will represent the result of executing the observability method. Initially, we have not found the requested item so we will initialize the *result* variable to false. Next, a loop is defined to iterate once for each item currently stored within the container. That is, from zero to the number of items stored within the container, provided by the *Size* observability inherited from the *ContainerBase* contract. The body of the loop contains two statements. The first gets the item stored in the container at the index specified by the loop counter, using the *ItemAt* observability. The resultant item is assigned to the *x* variable. The second statement, compares the *x* variable to the item that we are looking for. If they match, the *result* variable is set to true. Therefore, at the end of the loop, the *result* variable will have a value of true if the requested item is stored within the container and false otherwise. To return the value stored by the *result* variable to the caller, an assignment to the *value* keyword is performed. That is, the *value* keyword represents the return value of the observability.

After the observability definition, the *Container* contract contains a parameters section. The parameters section can contain any number of parameter definitions. The common use of parameters is to allow for the selection of various checks or metric evaluators. Parameters are analogous to constants found in programming languages, except that their values can be specified in one of three ways. The first is to specify the value directly, as was done in Figure 14. The second way is to use the binding tool to specify or override the value of a parameter. Finally, the value can be specified at runtime. The third way is useful if different instances of the same contract require different parameter values. The *Container* contract contains a single scalar Boolean parameter named *CheckMembers* whose value is set to true. We will use the parameter to indicate if we should perform static checks or not. Its use will be demonstrated in the next section.

Each contract can contain at most one structure section. The contents of a structure section are executed before the IUT is allowed to run. The structure section is used to specify any static checks that apply to the contract. Static checks are used to perform a check on the IUT that can be accomplished without execution. Examples of static checks include: tests on inheritance depth, and the correct structural use of design patterns [72]. A static check can be viewed as an operation. Each check has a return type and may accept a fixed number of parameters. All static checks are guaranteed to be side-effect free. The body of the structure section begins with a choice statement. Choice statements are analogous to *if* statements found in programming languages. The statement begins with the *choice* keyword followed by a condition enclosed in

matching brackets. The value stored in the *CheckMembers* parameter will be used as the condition. After the closing bracket, an operator is supplied. The operator indicates how the condition is to be compared to the value that follows. In our case, the body of the choice statement will be executed if the *CheckMembers* parameter is set to true. The body of the choice statement contains a belief. The belief is named *CheckMember* and has a description indicating that the container should have, as a data member, an underlying list that is capable of storing items of the type *tItem*. Such a check, located in the body of the belief, is accomplished by using the *HasMemberOfType* static check provided in the *Core* namespace that was imported at the beginning of Figure 14. The *HasMemberOfType* static check, accepts a type, and will succeed if the IUT type bound to the *Container* contract contains a variable, scalar or list, that is of the type *tItem*. As *tItem* is a placeholder, it will actually check if the IUT type contains a variable that is of the type bound to the *tItem* symbol. If such a type is found, the static check will succeed and result in the containing belief succeeding. If not, then the static check will fail, and result in the containing belief failing as well. More information regarding the execution of static checks can be found in Chapter 4.

Following the structure section, the *new* responsibility is refined. Refinement is accomplished by use of the *refine* modifier and is required when the base contract already defines a contract section with the same signature. When a responsibility is refined, the contents of any base responsibilities are evaluated along with the refined responsibility body. That is, the refined responsibility body provides additional behaviour for the responsibility. The refined *new* responsibility adds two additional

statements. The first statement sets the *number_of_items* contract variable to zero as our container does not store any items upon creation. The second statement, initializes the *container_items* integer list using the built-in *Init* method. The *Init* method allocates required memory and initializes the list. All lists must be initialized prior to use.

The *finalize* responsibility is also refined to include a fire statement. The fire statement is used to generate the *ContainerDone* observable event. Observable events can be used as steps in scenarios or relations to indicate the completion of a task or activity (as will be illustrated later). The events themselves are represented by identifiers and do not have to be previously declared. Once an event has been fired, through the use of a fire statement, the event is broadcasted to all active contract instances. We will see shortly how such an event is observed.

Refinement of the *Add* responsibility follows, a precondition is added to ensure that the item that we are trying to add to the container is not already stored in the container. That is, this Testable Requirements Model (TRM) does not support duplicate items in the container. Following the execute statement; a timer is started keyed on the item that was added to the container, using the *Start* built-in method defined on the timer type. We will use the timer to keep track of how long the item is stored within the container. For a complete list of built-in timer methods, see the Another Contract Language (ACL) specification document [8]. Finally, the *number_of_items* contract variable is incremented as we have just added a new element to the container. The

Insert responsibility is also refined to include the same precondition, timer start call, and contract variable increment.

As we have started a timer upon addition or insertion of an element to our container. We need to stop the timer when the corresponding element is removed. Such a task is accomplished by refining the *Remove* and *RemoveElement* responsibilities as follows. After the execute statement the *Stop* timer method will be invoked on the *item_timer* contract variable to stop the timer. Next, the amount of time[7] that the item spent in the container will be added to the *container_items* list. That is, the *container_items* list will contain an element for each item that was added and then removed from the container, indicating how long each item has been stored in the container.

The responsibilities discussed thus far will each be bound to a method found within the IUT. However, in some cases the behaviour of a responsibility can be specified in terms of other responsibilities. That is, instead of binding a responsibility to the IUT, the responsibility is defined in terms of an execution grammar. As the TRM represents a high level of abstraction, the specified execution grammar is not tied to the IUT, but rather is specified in terms of other responsibilities. As an example consider the *RemoveScn* responsibility. The *RemoveScn* responsibility represents the task of removing an element from the container. However, we do not care which of the two bound removal responsibilities are used, *Remove* or *RemoveElement*. Thus, the

[7] Time is represented in milliseconds.

execution grammar contained within the body specifies that either the *Remove* responsibility or the *RemoveElement* responsibility will execute, but not both. We will now discuss the responsibility body in detail. The *RemoveScn* responsibility begins with an execution grammar in place of the execute statement. Responsibilities that contain an execute statement will be bound to a method within the IUT, where responsibilities that do not, will be defined in terms of their execution grammar. In the case of our example, the execution grammar specifies that either the *Remove* or *RemoveElement* responsibility can be executed via the *or* operator (/), but not both. The *dontcare* keyword is used to substitute the parameter to the *RemoveElement* responsibility. Once the execution grammar is satisfied, execution will return to the calling scenario or relation.

Our TRM supports one additional type of responsibility, a stub responsibility. Stub responsibilities are analogous to stubs found in Use Case Maps (UCMs) [6]. They represent a placeholder for one or more responsibilities. The responsibilities to be executed can depend on a condition or parameter. Our example contains a simple stub responsibility to illustrate their use. A stub responsibility is denoted by the placement of the *stub* modifier before the *Responsibility* keyword as is the case for the *AddElement* stub responsibility. The body of the stub responsibility contains two statements. The first statement is a simple precondition to ensure that the item to be added to the container is a valid item (i.e., not null). The second statement is the stub selection statement. Each responsibility must contain at least one stub selection statement. The stub selection statement consists of a condition enclosed in square brackets, followed

by a responsibility that is to be used in place of the stub if the previous condition is satisfied. In our example, we will use the *Default* keyword to indicate that our single selection statement is the default choice. While our example stub responsibility only has a single choice, there is no limit to the number of choices a stub responsibility may contain. Stub responsibilities are not bound to an IUT counterpart, but rather select other responsibilities that could be bound to the IUT. Stub responsibilities are invoked when they are encountered in an execution grammar. The two previous responsibility types are invoked as a result of scenario processing. When a scenario instance encounters an unbound responsibility, the scenario invokes the responsibility. In the case of a non-stub compound responsibility, the responsibility acts as a compositional mechanism that can be named and referenced in scenarios. In the case of a stub responsibility, the responsibility acts as a placeholder allowing for several alternatives to be selected.

The *Container* contract includes a slightly more complex scenario than the one discussed in the *ContainerBase* contract: *ContainerLifetime*. As the name suggests, the *ContainerLifetime* scenario models the lifetime of our container. The body of the scenario begins with a triggering event, the execution of the *new* responsibility. Such a triggering event is common in cases where a given scenario models the complete lifetime of a contract instance. Note the *follows* operator (,) placed after the triggering event. The *follows* operator provides temporal ordering between elements of an execution grammar. In *A*, *B* the element *A* must execute before *B* (*A* follows *B*). Once a scenario instance has been triggered the next execution grammar element is an atomic

block. An atomic block, denoted by the *atomic* keyword, is a block of scenario grammar that must be executed as an atomic transaction. That is, no other responsibilities are allowed to execute except the ones specified by the execution grammar. The body of the atomic block contains two elements. The first element specifies that zero or more items are to be added to the container. The grammar specifies that either the *Add* or *Insert* responsibility can be used to add items to the container. In addition, the *dontcare* keyword is used to indicate that the actual item and insertion index are not of importance from the perspective of the scenario. The zero or more (*) operator indicates that the preceding execution grammar element may execute zero or more times. The second element of the atomic block is the *RemoveScn* compound responsibility. *RemoveScn* will be used to remove zero or more items from the container. The removal of items from the container completes the atomic block, and execution will move on to the next element of execution grammar, an observe element. The observe element is used to indicate that the next action is the observation of an event. Recall that events are fired using a fire statement. The observe element is denoted using the *observe* keyword followed by the event identifier enclosed in a pair of round brackets. Once the given event occurs, scenario execution will advance to the termination event. In the *ContainerLifetime* scenario, the termination event is the execution of the *finalize* responsibility. That is, the scenario terminates when the current contract instance is destroyed. For more information on elements that can be used as part of execution grammars see the ACL specification document [8]. The constructs we have discussed thus far all deal with functional aspects of a system using

a scenario-based approach. In order to craft a complete TRM we also need to model the non-functional aspects. We will now examine the non-functional constructs contained within the *Container* contract.

We begin with the definition of two metrics. Metrics provide a way to acquire metric information gathered during the execution of the containing contract. The first metric, *ContainerTimes*, returns a list of integers representing the amount of time that each element spent within the container. The body of the metric contains a single statement that will return our *container_items* list to the caller. If more than a single line is needed to compute the required metric, an assignment must be made to the *value* keyword, as was done in the *HasItem* observability method. The second metric, *NumberOfItems*, returns the total number of items that were stored in the container. The value is obtained by returning the *number_of_items* contract variable. All gathered metric values need to be analyzed and interpreted before they are presented to the user. Such analysis is performed in the reports section.

Each contract may have at most one reports section. The reports section is denoted by the *Reports* keyword. The purpose of the reports section is to use the values provided by the metric sections and evaluate them using metric evaluators. Metric evaluators are provided by domain experts to analyze the gathered metric data. Examples of metric evaluators include: performance, space, and network use analysis. Metric evaluators can only be invoked within a reports section. The reports section is invoked after the execution of the bound IUT has completed. The body of the reports

section allows for additional information to be generated that will be displayed in the Contract Evaluation Report (CER). The CER is a complete report illustrating the result of executing the IUT against the Testable Requirements Model (TRM). Details of the CER will be presented shortly. There are two ways to write additional information to the CER. The first is a report statement, denoted by the *Report* keyword. A report statement is similar to a C++ *printf* or C# *WriteLine* statement, in that it accepts a format string, and then a list of values to place into the string. Each report statement is executed once for each contract instance. In the case of the *Container* contract, the first report statement will use the *AvgMetric* metric evaluator to calculate the average value from the given list. The result is placed into the formatting string at the location denoted by *{0}*. The resultant string is then written to the CER. The next statement is a report all statement, denoted by the *ReportAll* keyword. A report all statement operates the same way as the report statement, except that it generates a single result for all contract instances. That is, in the *Container* contract it will calculate and report the average time across all container instances. The second report and report all statements report the total number of items that were added to the container, per instance, and in total respectively.

The final section in the *Container* contract is the exports section, denoted by the *Exports* keyword. Each contract may have at most one exports section. The exports section is used to define and assign binding rules for non-built-in types used within the containing contract. We have used the *tItem* type to represent the type of item being stored within the container. The exports section contains a single entry to define the

tItem symbol. The entry begins with the *Type* keyword to indicate that the *tItem* symbol is representing a type. Next the symbol name, *tItem*, is specified. The *Type* and *tItem* symbol are the only required elements. However, we can use the *conforms* keyword to indicate that the type that is bound to the *tItem* symbol will automatically have the previously defined *Item* contract applied. That is, the IUT type that is bound to the *tItem* type will automatically be bound to the *Item* contract. Next, within the body of our type definition, zero or more binding rules can be specified. Binding rules provide a way to link bindings together or to set restrictions on the IUT types that can be bound. The first binding rule, *not context*, indicates that the *tItem* type cannot be bound to the same type that is bound to our contract. That is, our testable model does not support a container of containers. The second binding rule, *non derived context*, indicates that the *tItem* type cannot be bound to any type that is derived from our container. The exports section completes the *Container* contract.

3.1.1.4 The ContainerInter Interaction

The contracts that have been discussed thus far are based around the notion of a type. Each contract is bound to a type located within the IUT. As such our contract semantics impose a limitation that everything is based around the notion of a type. That is, there is no way to specify the interaction between contracts, and more importantly between scenarios. Such relationships can be captured through the use of an interaction. The interaction for our container is shown in Figure 15.

```
Import Core;

Namespace DaveArnold.Examples
{
   Interaction ContainerInter
   {
      Relation MultipleContainers
      {
         Contract Container c;
         c.AddAndRemove[6] || c.ContainerLifetime[2];
      }
   }
}
```

Figure 15 - ContainerInter Interaction

Like contracts, interactions must reside within a namespace declaration. As with all of our contracts our interaction will be located in the *DaveArnold.Examples* namespace. The interaction itself is defined using the *Interaction* keyword followed by the name of the interaction: *ContainerInter*. Unlike contracts, interactions only contain a single section known as a relation. Relations are used to specify an ordering of execution. The ordering is imposed on scenarios and responsibilities defined within any number of contracts. Relations themselves are denoted by the *Relation* keyword, and are followed by the name of the relation. The relation name is used for reporting, and has no other use.

The *ContainerInter* interaction contains a single relation named *MultipleContainers*. The relation defines how our two scenarios interact. The body of the relation begins with the declaration of a contract variable. A contract variable represents an instance of a contract. The variable declaration begins with the *Contract* keyword followed by the name of the contract, and then the variable name. In the *MultipleContainers* relation we are going to create an instance of the *Container* contract named *c*. The next statement is the relation grammar. Relation grammars can be complex like the

execution grammars found within scenarios, but for the purposes of this example we will keep it simple. The *independence* (//) operator indicates that six *AddAndRemove* scenario instances can execute independently from the two *ContainerLifetime* scenario instances. The number of instances is a result of how the IUT is designed. We will present such an IUT in the next section. Normally, the number of scenario instances would come from the system requirements.

In summary, we have presented an implementation independent Testable Requirements Model (TRM) for a container. The container is composed of a set of items that, individually, must conform to the Item contract. The container itself is composed of an abstract base contract and a concrete specialization containing responsibilities for the addition and removal of items. Specific addition and removal must follow the execution grammar defined in our two scenarios. Metrics regarding the number of items and the amount of time items are stored within the container are captured and analyzed. In addition, an interaction was created to define a relation between our two scenarios. For more information on contracts, interactions, responsibilities, scenarios, and relations see the ACL specification document [8] or one of the other case studies that will be discussed in Section 3.3.

3.1.2 The Implementation Under Test

Our Validation Framework (VF) provides an open architecture for the specification and execution of the TRM presented in the previous section. As will be detailed in Chapter 4, the VF is integrated into Microsoft's Visual Studio [137] for ease of use and incorporation into existing development processes. The VF requires three elements as

input. The first element is the set of one or more contracts and interactions, that is, the TRM. More precisely, we have defined a high-level general purpose contract language known as the Another Contract Language (ACL). As we have seen in Section 3.1.1, the ACL includes constructs for the capture of static checks, dynamic checks, responsibilities, scenarios, metric capturers, and metric evaluators. Additional domain-specific constructs can also be added to the ACL, via VF plug-ins [8] as will be discussed in Chapter 4.

The second input element is one or more candidate IUTs against which the Testable Requirements Model (TRM) will be executed. The VF accepts implementations in binary form. That is, source code for the IUT is not required. Through the use of Microsoft's Phoenix Research Development Kit [144], the VF is able to open .NET managed executables, as well as C++ binaries that contain debugging information. The debugging information is required to resolve symbol names during the binding process.

Bindings represent the third and final input element to our framework. Each IUT has a unique set of bindings. Before a TRM can be verified by the VF, some of its elements must be bound to implementation artifacts located within the IUT. Such bindings are essential for the automated execution of tests. The binding process begins by using Phoenix to obtain a structural representation of the IUT. Then, the binding tool allows for the specification of a mapping from each observability and responsibility defined within a contract to an actual method (or group of methods) within the IUT. Most importantly, bindings allow contracts to be independent of implementation details, as

specific IUT method names, and parameter types/orders used within the IUT do not have to correspond to a similarly-named contract artifact. In addition, such bindings allow several candidate IUTs to be verified against a single TRM. We will now present such a candidate IUT for our container, and then examine corresponding bindings between the IUT and TRM. By candidate, we mean one that meets all requirements defined by the TRM. Looking at the previous Testable Requirements Model (TRM), we can determine a set of requirements for our IUT. The *Item* contract from Figure 12 contains an observability named *Value* that returns a scalar integer. Recall that observabilities define observable requirements for the IUT. That is, they are bound to read-only methods that provide the requested state information. Therefore, any candidate IUT will have to supply a method providing the required state information for each observability defined within the TRM. The *Item* contract also contains a *new* and *finalize* responsibility. As these responsibilities are executed during construction and destruction of an IUT type, no additional constraints on the IUT are mandated with respect to the *Item* contract.

3.1.2.1 The ContainerItem Class

Our IUT will be written using C# and will consist of three classes. The class shown in Figure 16 will represent an element stored within the container. The *ContainerItem* class is straightforward, it contains a single instance variable named *item* to represent the actual value stored by this container item. Upon construction of a new *ContainerItem* instance, the *item* instance variable is assigned a random number between 0 and 100. On destruction of the instance the *item* instance variable is set to

zero. Finally, the *Value* accessor provides read-only access to the actual value stored by the container item. The *Value* accessor also satisfies the observability requirement imposed by the *Item* contract.

```
using System;

namespace DaveArnold.VF.Examples.Container
{
    public class ContainerItem
    {
        private static Random rx = new Random();
        private int item = 0;

        public ContainerItem()
        {
            item = rx.Next(100);
        }

        ~ContainerItem()
        {
            item = 0;
        }

        public int Value { get { return item; } }
    }
}
```

Figure 16 - ContainerItem Class

3.1.2.2 The Container Class

Continuing with the *Container* contract of Figure 14, which includes the contents of the ContainerBase contract of Figure 13, each of the four non-defined observability methods contribute an observability requirement for the candidate Implementation Under Test (IUT). The *Add*, *Insert*, *Remove*, and *RemoveElement* responsibilities are defined as bound responsibilities. Thus, they each represent a task that the candidate IUT must realize. It should be noted that the IUT does not have to provide a one-to-one mapping of responsibilities to methods. For example, the *Add* responsibility could be implemented across several methods within the IUT. That is, the addition of an item to

the container could be accomplished by the invocation of a series of methods within the IUT. The *RemoveScn* and *AddElement* responsibilities represent unbound, and stub responsibilities respectively. As such, they do not require a counterpart within the candidate IUT. Scenarios are defined using an execution grammar, and are not directly connected to the IUT. The only other requirement for our candidate IUT is found in the exports section. The exports section indicates that the candidate IUT must contain an element type, and the element type cannot be the same type used to implement the container (or any derivation thereof). Such a type must also conform to the requirements specified in the *Item* contract. The *Container* class shown in Figure 17 satisfies the requirements obtained from our TRM.

```
using System;
using System.Collections.Generic;

namespace DaveArnold.VF.Examples.Container
{
    public class Container
    {
        private List<ContainerItem> items;

        public Container()
        {
            items = new List<ContainerItem>();
        }

        ~Container()
        {
            items = null;
        }

        public bool IsFull { get { return false; } }

        public bool IsEmpty { get { return items.Count == 0; } }

        public int Size { get { return items.Count; } }

        public ContainerItem ItemAt(int index)
        {
            return items[index];
        }
```

```
        public void Add(ContainerItem item)
        {
            items.Add(item);
        }

        public void Insert(int index, ContainerItem item)
        {
            items.Insert(index, item);
        }

        public ContainerItem Remove()
        {
            ContainerItem result = items[0];
            items.RemoveAt(0);
            return result;
        }

        public void RemoveElement(ContainerItem item)
        {
            items.Remove(item);
        }
    }
}
```

Figure 17 - Container Class

The *Container* class encapsulates a strongly typed generic list of *ContainerItem* objects. The generic *List* type is provided by the .NET Framework Class Library [142]. The *Container* class begins with the declaration of the underlying list of container items. The constructor creates a new instance of the underlying list, and the destructor sets the *items* list to *null*. Next the *IsFull* accessor is used to indicate if the container is full or not. As the generic *List* type is unbounded, *IsFull* will always return false. The *IsEmpty* accessor indicates if the container is empty or not, emptiness is determined by checking the number of items contained within the underlying list. The final accessor size, returns the number of items stored in the underlying list.

A series of five methods follow the accessors and complete the container. The *ItemAt* method returns the *ContainerItem* stored at the given index. *Add* and *Insert* allow for new elements to be added and inserted into the underlying list respectively.

The *Remove* method removes the first element from the list and returns it to the caller. Finally, the *RemoveElement* method removes the given element from the underlying list, completing the *Container* class. We will now present the final class in our Implementation Under Test (IUT): *Program*.

3.1.2.3 The Program Class

The *Program* class shown in Figure 18 contains the *Main* method for the IUT. The *Main* method can be seen as the test driver for the candidate IUT. That is, the *Program* class is not bound to an element within the Testable Requirements Model (TRM), but controls its execution. As such the TRM does not impose any structural requirements on the *Program* class. However, if the IUT is to be declared validated, then the behaviour generated by the *Main* method must obey all scenarios and relations defined within the TRM. The *Main* method provided in Figure 18 obeys the TRM presented in Section 3.1.1.

The body of the *Main* method begins with a *for* loop that executes twice. The body of the *for* loop creates a new instance of the *Container* class and three items. The first two items are added, via the *Add* method, and the third item is inserted into the container via the *Insert* method. Next, the first two items are removed from the container. Removal of the third item follows using the *Remove* method. Finally, the container instance is assigned to *null*. The *Program* class completes the candidate IUT.

```
using System;

namespace DaveArnold.VF.Examples.Container
{
    class Program
    {
        static void Main(string[] args)
        {
            for (int i = 0; i < 2; i++)
            {
                Container c = new Container();
                ContainerItem ix = new ContainerItem();
                ContainerItem iy = new ContainerItem();
                ContainerItem iz = new ContainerItem();
                c.Add(ix);
                c.Add(iy);
                c.Insert(0, iz);
                c.RemoveElement(ix);
                c.RemoveElement(iy);
                c.Remove();
                c = null;
            }
        }
    }
}
```

Figure 18 - Program Class

3.1.3 Bindings

As previously discussed, the remaining element required for the execution of a Testable Requirements Model (TRM) against a candidate IUT is a set of bindings. Such binding information links contracts, observabilities, bound responsibilities, and exported symbols to corresponding IUT elements. Bindings can be specified manually or using the Automated Binding Engine (ABE). The ABE does not use a specific binding algorithm. Rather, it is open in nature, and uses extension modules that are developed using a software development kit [9]. We have implemented two ABE extension modules as part of the VF, details of which will be presented in Chapter 4. For now, it is enough to know that the ABE uses the selected extension module to infer a correct binding between each TRM element and a corresponding IUT counterpart. Table 2 contains the

complete binding table connecting our container TRM presented in Section 3.1.1 and our candidate IUT presented in Section 3.1.2. The TRM type column in Table 2 uses a *C* to indicate a contract, *O* for observability, *R* for responsibility, and *E* for exported symbol. Likewise the IUT Type column uses a *C* to indicate a class, and *M* to indicate a method.

TRM Element Name	TRM Type	IUT Bindpoint	IUT Type
Item	C	Examples::ContainerItem	C
Integer Item.Value()	O	int Examples::ContainerItem.Value()	M
Void Item.new()	R	int Examples::ContainerItem.ctor()	M
Void Item.finalize()	R	int Examples::ContainerItem.dtor()	M
Container	C	Examples::Container	C
Boolean Container.IsFull()	O	bool Examples::Container.IsFull()	M
Boolean Container.IsEmpty()	O	bool Examples::Container.IsEmpty()	M
tItem Container.ItemAt(Integer index)	O	Examples::ContainerItem Examples::Container.ItemAt(int)	M
Integer Container.Size()	O	int Examples::Container.Size()	M
Void Container.new()	R	void Examples::Container.ctor()	M
Void Container.finalize()	R	void Examples::Container.dtor()	M
Void Container.Add(tItem item)	R	void Examples::Container.Add(Examples::ContainerItem)	M
Void Container.Insert(Integer index, tItem item)	R	void Examples::Container.Insert(int, Examples::ContainerItem)	M
tItem Remove()	R	Examples::ContainerItem Examples::Container.Remove()	M
Void RemoveElement(tItem item)	R	void Examples::Container.RemoveElement(Examples::ContainerItem)	M
tItem	E	Examples::ContainerItem	C

Table 2 - Binding Table for the Container Testable Requirements Model and C# IUT

Once a set of bindings between a TRM and each candidate IUT are complete, the binding tables are saved. Saving the binding table removes the need for re-specification each time the VF executes a candidate IUT against a TRM. The binding data only needs to be re-specified if either the IUT or TRM are modified. If such a modification does occur, the binding tool only requires re-computation of the modified portions.

3.1.4 Execution

With the binding information specified, our Validation Framework (VF) is able to execute a candidate Implementation Under Test (IUT) against the Testable Requirements Model (TRM). The execution process begins with the evaluation of any static checks. Such a check can be performed on the IUT without execution. The execution of a static check begins by using Phoenix [144] to gather structural and behavioural elements of the IUT (details of which will be provided in Section 4.2.2). Once the structural and behavioural information is gathered, the actual static checks are executed. Each static check is able to examine the structural and/or behavioural elements within the IUT to produce a result. All static checks produce a pass or fail result. That is, the success or failure of a check is not a rhetorical exercise open to debate. Instead, the check is executed against the candidate IUT, to generate a deterministic result. Such a pass or fail result is displayed in the Contract Evaluation Report (CER). The report is displayed following execution of the candidate IUT against the TRM.

Once all of the static checks have been executed, the VF executes the IUT. By 'execute' we mean that the VF launches the IUT in a new process, and attaches itself to the process. Put another way, the VF acts as a specialized debugger and/or profiler keeping track of instance creation, method invocation, and instance destruction. In addition, the VF will pause execution of the IUT as needed in order to execute various checks as methods bound to responsibilities are invoked. At the same time, the VF also maintains a separate execution environment for the TRM. The environment contains

any contract, interaction, relation, or scenario instances that have been created as a result of IUT execution. Precise details regarding how a TRM is executed against a candidate IUT are provided in Chapter 4.

Once the IUT finishes execution, the VF notifies any scenario or relation instances that have yet to terminate that the IUT has finished, and thus the scenario or relation instance fails. Next, the results of executing each scenario and relation instance are written to the CER. Such information includes the execution grammar specified in the TRM, the actual IUT execution trace, and any unexpected responsibilities or observable events.

The final step in execution of an IUT against a TRM is the evaluation of any reports sections. Evaluation consists of the VF invoking any specified metric evaluators to analyze and report on metric information gathered during execution of the IUT. The gathering of metric data is performed by the VF in conjunction with any metrics gathered within the TRM itself, as was the case in our container. The results of metric analysis are formatted as per the string provided to the report and report all statements and written to the CER. The evaluation of all reports sections completes the process of executing a candidate IUT against a TRM.

3.1.5 The Contract Evaluation Report

The Contract Evaluation Report (CER) displays results covering all aspects of a candidate IUT executed against the Testable Requirements Model (TRM). The results of each contract, contract instance, observability, invariant, responsibility, scenario, metric

analysis, interaction, interaction instance, and relation are shown. Each distinct candidate IUT executed by our VF will have a separate CER. Multiple CERs can be compared to find differences between candidate IUTs.

The CER itself is presented using a tree that displays each element of the TRM in either green or red. Elements in green represent areas where execution of the candidate IUT followed the specified TRM. Elements in red represent areas where execution of the candidate IUT deviated from the TRM. Upon selection, each node within the CER tree displays a report view showcasing precise execution details based on the currently selected tree item. Figure 19 shows both the tree and report views generated by the execution of our container TRM against our candidate IUT using the bindings provided in Table 2. The report view displayed in Figure 19 is the scenario instance view. The scenario instance view provides the actual triggering and termination events for a given scenario, as well as the exact execution trace that was observed during execution of the selected scenario instance. In addition, details regarding the execution of any preconditions, checks, beliefs, post-conditions, and dynamic checks executed as part of the scenario instance are provided. The CER has a total of 22 different report views that can be displayed depending on the selection made in the report tree. Additional information about the information written to the CER and its corresponding views will be provided in Chapter 4.

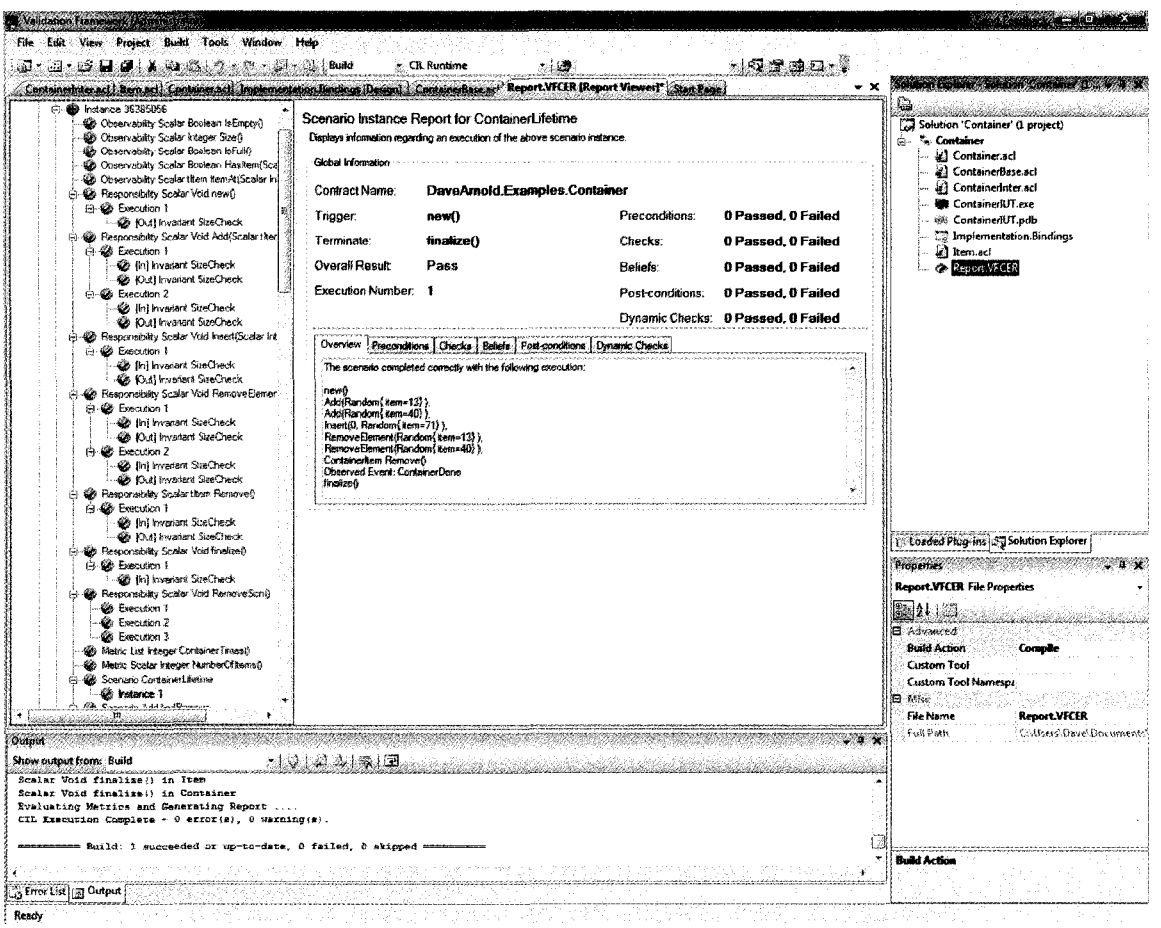

Figure 19 - Container Contract Evaluation Report - Scenario View - C# IUT

Our example has created an implementation independent TRM representing a container. We then created a concrete candidate IUT using requirements imposed by our TRM. The TRM and candidate IUT were connected through the notion of bindings. Bindings allow an implementation independent TRM to execute against one or more candidate IUTs without the need for glue code as presented in Chapter 1. Our VF performs the actual execution, by monitoring execution of the IUT, and validating the checks and execution grammars specified in the TRM. The result of such execution is the Contract Evaluation Report (CER). The CER displays the results of executing a candidate IUT against the TRM. Generation of the CER completes our example. We will now examine how our TRM can be modified to support multiple copies of the same

element within the container. Such a modification will lead to key semantic aspects of the Another Contract Language (ACL).

3.2 The Container Example Version 1.1: Multiple Occurrences

In order to support multiple occurrences of the same item within the container, our Testable Requirements Model (TRM) can no longer use the value of an item to determine inclusion in the container or to key timers. Instead, we must use references (i.e., pointers) to track items, and as such compare items by reference instead of value. As our TRM is implementation independent, it does not directly support the notion of pointers. That is, our TRM handles value and reference types as a single type.

However, in the case of multiple occurrences we require that items be compared by reference instead of by value. To support such comparisons, we will use additional mechanisms of the ACL in our TRM as follows:

1. The *&=* operator will compare two variables by reference for equality, instead of by value (i.e., the == operator). It should be noted, that the IUT is not required to support or use reference types when the *&=* operator is used. Rather, the *&=* operator serves as an instruction to our VF and does not impose any additional constraints on the IUT.
2. The *not&=* operator will compare two variables by reference for inequality, instead of by value (i.e., the *not=* operator).

3. Any exported symbol within a contract that is bound to a type within the IUT can invoke the *IdentityOf()* built-in method to acquire a reference to the item. That is, a pointer to the item.

Using these features of the Another Contract Language (ACL), we will now modify the container example to support multiple occurrences of the same item. This second version of our example will conclude our presentation of the ACL.

3.2.1 The Testable Requirements Model

All modifications to the Testable Requirements Model (TRM) of the container are confined to the *Container* contract. The *Item* and *ContainerBase* contracts are unchanged from Section 3.1 and will not be repeated here. In addition the *ContainerInter* interaction is also unchanged and will not be repeated. We will focus our attention on the *Container* contract.

3.2.1.1 The Container Contract

The modified *Container* contract is shown in Figure 20. The modified lines are highlighted. The first modification is located in the body of the *HasItem* observability. The elements within the container are now compared using the &= operator instead of the == operator. The change allows duplicate elements to reside within the container, as each element within the container will have a unique reference. Next, where the timers are started and stopped the built-in *IdentityOf()* method is used to use each item's reference value as the timer key, rather than the item itself. Doing so allows each copy of the same item to have a unique timer key.

```
Import Core;

Namespace DaveArnold.Examples
{
   MainContract Container extends ContainerBase<tItem>
   {
      List Integer container_times;
      Scalar Timer item_timer;
      Scalar Integer number_of_items;

      refine Observability Boolean HasItem(tItem item)
      {
         tItem x;
         Boolean result = false;
         loop(0 to Size())
         {
            x = ItemAt(counter);
            result = result || x &= item;
         }
         value = result;
      }

      Parameters
      {
         Scalar Boolean CheckMembers = true;
      }

      Structure
      {
         choice(Parameters.CheckMembers) == true
         {
            Belief CheckMember(
               "There should be a member of our container")
            {
                  HasMemberOfType(tItem);
            }
         }
      }

      refine Responsibility new()
      {
         number_of_items = 0;
         container_times.Init();
      }

      refine Responsibility finalize()
      {
         fire(ContainerDone);
      }
```

```
refine Responsibility Add(tItem item)
{
    Pre(HasItem(item) == false);
    Execute();
    item_timer.Start(item.IdentityOf());
    number_of_items = number_of_items + 1;
}

refine Responsibility Insert(Integer index, tItem item)
{
    Pre(HasItem(item) == false);
    Execute();
    item_timer.Start(item.IdentityOf());
    number_of_items = number_of_items + 1;
}

refine Responsibility tItem Remove()
{
    Execute();
    item_timer.Stop(value.IdentityOf());
    container_times.Add(item_timer.Value(value.IdentityOf()));
}

refine Responsibility RemoveElement(tItem item)
{
    Execute();
    item_timer.Stop(item.IdentityOf());
    container_times.Add(item_timer.Value(item.IdentityOf()));
}

Responsibility RemoveScn()
{
    Remove() | RemoveElement(dontcare);
}

stub Responsibility AddElement(tItem item)
{
    Pre(item not= null);
    [Default] Add(item);
}

Scenario ContainerLifetime
{
    Trigger(new()),
    atomic
    {
        (Add(dontcare) | Insert(dontcare, dontcare))*,
        (RemoveScn())*;
    },
    observe(ContainerDone),
    Terminate(finalize());
}
```

```
        Metric List Integer ContainerTimes()
        {
            context.container_times;
        }

        Metric Scalar Integer NumberOfItems()
        {
            context.number_of_items;
        }

        Reports
        {
            Report(
            "The average time in the container is {0} milliseconds",
            AvgMetric(ContainerTimes()));

            ReportAll(
            "The average time in all containers is {0} milliseconds",
            AvgMetric(ContainerTimes()));

            Report("The number of items added to the container is {0}",
            NumberOfItems());

            ReportAll(
            "The number of items added to all containers is {0}",
            NumberOfItems());
        }

        Exports
        {
            Type tItem conforms Item
            {
                not context;
                not derived context;
            }
        }
    }
}
```

Figure 20 - Modified Container Contract

The modifications to the *Container* contract are all that is required for our Testable Requirements Model (TRM) to support multiple occurrences of an element within the container. We will now create an IUT that stores multiple occurrences of the same item.

3.2.2 The Implementation Under Test

Rather than modify the C# Implementation Under Test (IUT) from Section 3.1.2, we will create a new IUT in C++ to illustrate how our VF and corresponding binding tool can

execute a single TRM against multiple IUTs designed using different implementation technologies. The following sections will define the new IUT.

3.2.2.1 The Item Class

The *Item* class is defined in Figure 21 and implemented in Figure 22. The *Item* class will represent an item stored within the container. As before each item will store a single integer value. The value itself can be accessed using the *Value* method. The value of the container item is specified via the constructor.

```
ref class Item
{
private:
   int m_iValue;
public:
   Item(int value);
   ~Item(void);

   int Value();
};
```

Figure 21 - Item C++ Header

```
#include "Item.h"

Item::Item(int value)
{
   m_iValue = value;
}
Item::~Item()
{
   m_iValue = 0;
}
int Item::Value()
{
   return m_iValue;
}
```

Figure 22 - Item C++ Class

3.2.2.2 The Container Class

The *Container* class is defined in Figure 23. Unlike our previous IUT, our container will maintain an array of items, and an integer value representing the number of items

stored within the container. Also notice the polymorphic *Remove* methods, as we will see shortly, the binding tool can still connect the IUT with our TRM. The implementation of the *Container* class is shown in Figure 24.

```
#include "item.h"

ref class Container
{
private:
   array<Item^>^ m_pItems;
   int m_iSize;
public:
   Container(void);
   ~Container(void);

   bool IsFull(void);
   bool IsEmpty(void);
   int  Size(void);
   Item^ ItemAt(int index);
   void Add(Item^ item);
   void Insert(int index, Item^ item);
   Item^ Remove();
   void Remove(Item^ item);
private:
   Item^ RemoveAt(int index);
};
```

Figure 23 - Container C++ Header

```
#include "Item.h"
#include "Container.h"

Container::Container()
: m_iSize(0)
{
   m_pItems = gcnew array<Item^>(10);
}

Container::~Container() { m_pItems = nullptr; }
bool Container::IsFull() { return m_iSize == 10; }
bool Container::IsEmpty() { return m_iSize == 0; }
int  Container::Size() { return m_iSize; }

Item^ Container::ItemAt(int index)
{
   return m_pItems[index];
}
void Container::Add(Item^ item)
{
   m_pItems[m_iSize++] = item;
}
void Container::Insert(int index, Item^ item)
{
   for(int i =m_iSize;i>index;i--)
      m_pItems[i] = m_pItems[i-1];
   m_pItems[index] = item;
   m_iSize++;
}
Item^ Container::Remove()
{
   return RemoveAt(0);
}
void Container::Remove(Item^ item)
{
   for(int i=0;i<m_iSize;i++)
   {
      if(ItemAt(i) == item)
      {
         RemoveAt(i);
         return;
      }
   }
}
Item^ Container::RemoveAt(int index)
{
   Item^ result = m_pItems[index];
   for(int i=index;i<m_iSize;i++)
      m_pItems[i] = m_pItems[i+1];
   m_pItems[--m_iSize] = nullptr;
   return result;
}
```

Figure 24 - Container C++ Class

3.2.2.3 *The main Method*

The only element remaining for our new IUT is the program entry point, the *main* method. As with our C# IUT, the *main* method, shown in Figure 25, contains a *for* loop that iterates twice. Each iteration of the loop creates a new container and a single element. Three copies of the element are added to the container, followed by the removal of the three elements. The same functionality as the previous IUT was maintained so that the scenario and relation portions of the TRM do not require modification.

```
#include "container.h"

int main(array<System::String ^> ^args)
{
    for(int i =0;i<2;i++)
    {
        Container ^c = gcnew Container();
        Item ^item = gcnew Item(0);
        c->Add(item);
        c->Add(item);
        c->Insert(0, item);
        c->Remove(item);
        c->Remove(item);
        c->Remove();
        c = nullptr;
    }
    return 0;
}
```

Figure 25 - Container C++ main Method

3.2.3 Bindings

As a new Implementation Under Test (IUT) has been created, a new set of bindings must also be created to connect our C++ candidate IUT to the Testable Requirements Model (TRM). Even with the differences in implementation languages, strategies, and the user of pointers, the Automated Binding Engine (ABE) that is part of our VF is able to

infer all 16 required bindings. Table 3 contains the resultant binding table. Information regarding how the actual bindings are inferred will be presented in Chapter 4.

Once a set of bindings connecting our C++ IUT to the container TRM have been specified; the VF is able to execute the C++ IUT against the TRM to produce a Contract Evaluation Report (CER). The execution process is the same as previously discussed in Section 3.1.4 and will not be repeated. Figure 26 displays the relation instance view of the CER. Notice the additional IUT, and bindings in the Solution Explorer window

TRM Element Name	TRM Type	IUT Bindpoint	IUT Type
Item	C	Item	C
Integer Item.Value()	O	int Item.Value()	M
Void Item.new()	R	int Item.Item(int)	M
Void Item.finalize()	R	int Item.~Item()	M
Container	C	Container	C
Boolean Container.IsFull()	O	bool Container.IsFull()	M
Boolean Container.IsEmpty()	O	bool Container.IsEmpty()	M
tItem Container.ItemAt(Integer index)	O	Item* Container.ItemAt(int)	M
Integer Container.Size()	O	int Container.Size()	M
Void Container.new()	R	void Container.Container()	M
Void Container.finalize()	R	void Container.~Container()	M
Void Container.Add(tItem item)	R	void Container.Add(Item*)	M
Void Container.Insert(Integer index, tItem item)	R	void Container.Insert(int, Item*)	M
tItem Remove()	R	Item* Container.Remove()	M
Void RemoveElement(tItem item)	R	void Container.Remove(Item*)	M
tItem	E	Item	C

Table 3 - Binding Table for the Container Testable Requirements Model and C++ IUT

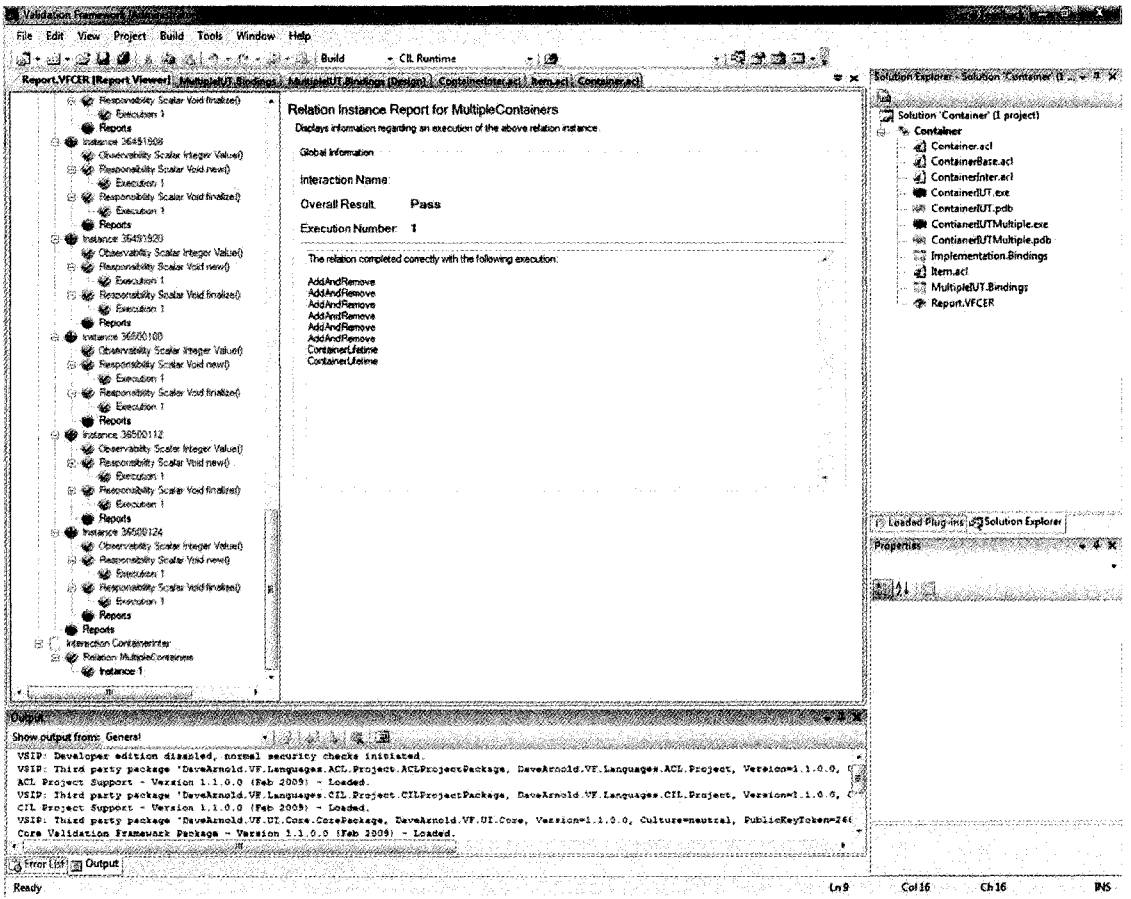

Figure 26 - Contract Evaluation Report - Relation View - C++ IUT

The previous two sections have illustrated how an implementation independent TRM modeling a container can be executed against two different IUTs through the notion of bindings. Bindings bridge the gap between the TRM and a concrete implementation. Our VF is able to take the TRM, candidate IUT, and bindings as input and execute the candidate IUT against the TRM to produce a CER. The example presented here is one of five case studies that we have created to validate our approach and the VF itself. The following section will provide a brief summary of the additional four case studies and links to their documentation and source code.

3.3 Additional Case Studies

In addition to the container presented in the previous two sections, four additional case studies have been created to illustrate and validate different aspects of our Validation Framework (VF). Each case study is presented in a technical report along with accompanying source code that provides a Testable Requirements Model (TRM), corresponding Implementation Under Test (IUT), and a set of bindings. All examples are fully operational within the VF. That is, each example specifies a set of checks via a TRM that have been executed against their corresponding IUTs using a set of bindings to produce a Contract Evaluation Report (CER). Such execution provides validation for not only our approach but also for the tool itself.

3.3.1 A Simple Container

Our introductory example, takes the form of a simple container [19]. Unlike the container presented here, the simple container example makes use of a single contract for the container. Concepts such as inheritance, responsibility extension, and generics are not used. The main strengths of the example lie in the documentation. The document provides a step-by-step guide for creating and executing the example within the VF. It serves as an introduction to the tool itself and is recommended for anyone who is new to the VF and the notion of TRMs. The VF installation and usage guide [23] also provides step-by-step instructions for the installation and general functionality of the VF.

3.3.2 Web-Based IUTs

In addition to conventional IUTs that execute alongside of the VF, the VF also contains a built-in web server for the validation of web applications. Web applications that are created using ASP.NET technology can be executed against a TRM. The web case study creates a TRM for a user logging in and out of a protected area of a website [21]. The case study introduces the concept of a web-based IUT and illustrates how the VF executes a web application using its built-in web server. Additional information regarding how web-based IUTs are executed against a TRM will be presented in the next chapter. It should be noted that a single TRM can be used to validate both a conventional and web-based IUT as long as each IUT has a valid set of bindings.

3.3.3 The Grocery Store

The grocery store case study provides a large scale example [20]. Concepts such as inheritance and generics are used in a multi-contract TRM. The case study models a physical grocery store and was selected because it requires little domain knowledge from the reader. A grocery store can be viewed as a set of customers who each, enter the store, and select one or more items for purchase. Each of these items is placed into their shopping cart. Once the customer has finished selecting the items that he/she wishes to purchase, they proceed to the checkout, where they select an open cash, and then join the queue (possibly empty). Finally the customer purchases their goods and leaves the store. The provided documentation and source code [20] include a complete listing and discussion of the TRM, a basic IUT, and bindings between the two.

3.3.4 The University

The university case study is modeled after a physical university [22]. Concepts such as inheritance, generics, and scenario interaction are covered in depth using multiple contracts and interactions. The premise of the case study is the operation of a university as follows. The university creates a set of courses each term. Once course creation is completed, students register in courses. Following course registration, the term begins. During a term, students must complete their courses by doing assignments, midterms and final exams. After the term ends, each course reports final grades to the university, and the university decides if a given student is allowed to continue with his/her education. The university case study is the most complex of our five case studies. The case study comes complete with documentation, a TRM specified in ACL, a basic IUT, and set of corresponding bindings [22].

Through our five case studies we are able to validate our approach of using an implementation independent TRM and a set of bindings to automatically generate and execute a set of executable tests that run against a candidate IUT. Our VF tool allows for the specification, and execution of such a TRM. We will now present the VF, and detail its operation.

4 The Validation Framework

The current chapter will provide details regarding the operation of the Validation Framework (VF) that we have built. That is, all of the process and corresponding execution discussed in this chapter has been fully implemented and validated using the five case studies mentioned in Chapter 3. The VF itself, example case studies, and additional documentation can be downloaded from the following location: http://vf.davearnold.ca. The chapter begins with an overview of the execution process employed by the VF. A discussion of the implementation technology used in construction of the VF follows. Discussion of the Another Contract Language (ACL) follows, the ACL being used for the specification of Testable Requirements Models (TRMs). We will then examine the different types of Implementation Under Tests (IUTs) that are supported by the VF, and their resultant bindings. The binding discussion will include how the Automated Binding Engine (ABE) works and how user provided binding extensions offer support for domain-specific binding algorithms. We will continue with the Contract Intermediate Language (CIL) that provides an intermediate representation of our TRM once binding has been completed. With each of the required input elements provided specific details regarding execution of the VF will be provided, including, execution of static checks, dynamic checks, and metric evaluators. The chapter continues with a discussion of the Contract Evaluation Report (CER), which provides details resulting from execution of the VF. Discussion of additional validation techniques used to validate individual modules of the VF concludes the chapter.

4.1 Overview

The VF is a standalone tool allowing for the specification of a TRM, and execution of that TRM against a candidate IUT to produce a CER. Figure 27 provides an overview of the execution process used by the VF. Each bubble shown in the figure will be discussed in a dedicated section in this chapter.

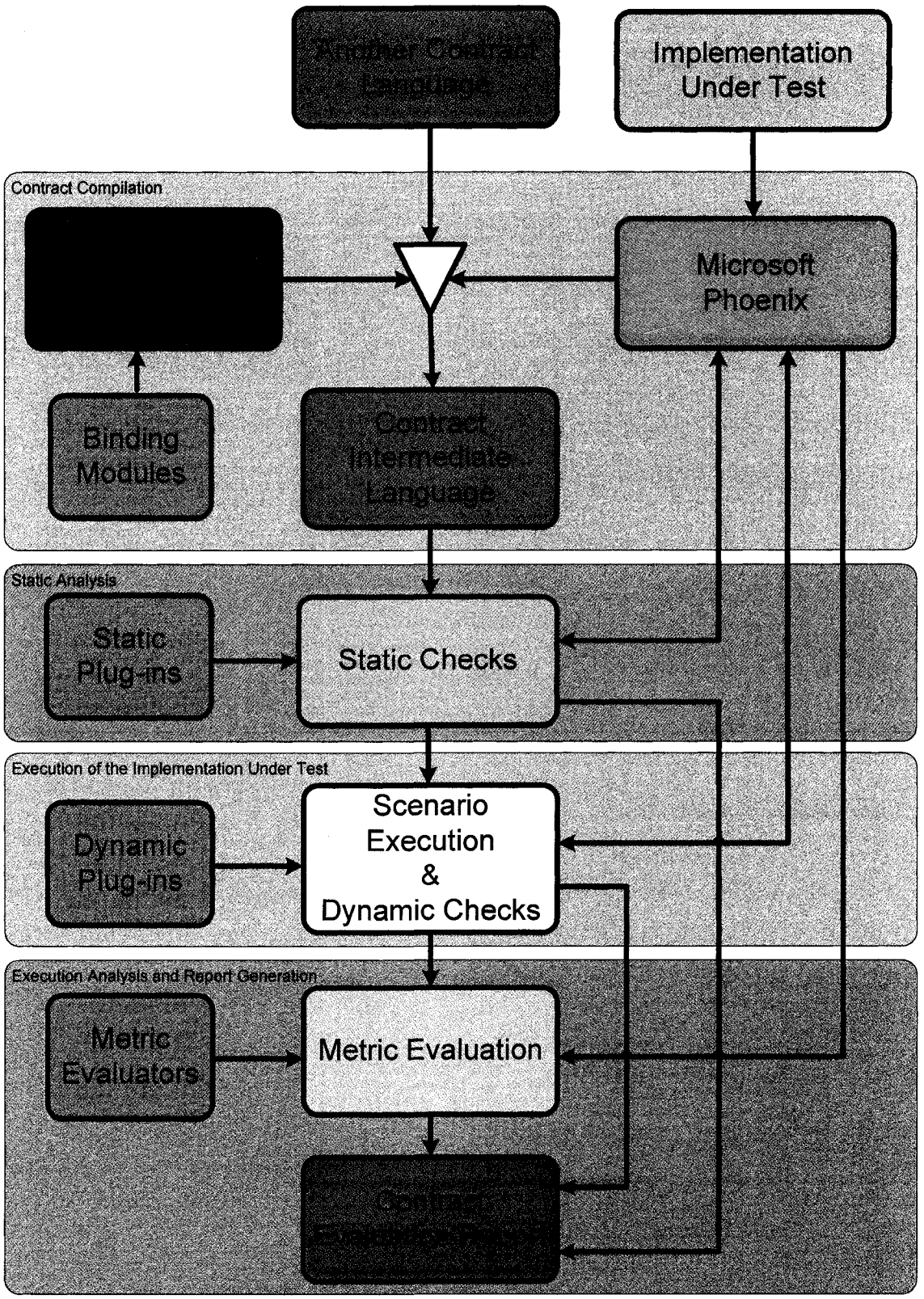

Figure 27 - Validation Framework Process Overview

The VF requires three elements as input. Two must be explicitly provided, and the third can be either automatically inferred or specified manually. Each input element is contained within a contract project. Contract projects allow for the grouping and containment of input elements, and the resultant Contract Evaluation Report (CER). The first input element is the Testable Requirements Model (TRM). As we have already seen in Chapter 3, the TRM is specified using our Another Contract Language (ACL) [8]. The ACL is a high-level general-purpose contract specification language that provides syntax and semantics for the capture of a TRM. The second input element is a candidate IUT. The VF executes the TRM against the candidate IUT to produce a CER. A contract project can contain any number of distinct candidate IUTs, however only one is executed at a time. If a contract project does not contain an IUT, the VF is still able to perform syntactical and semantic analysis on the TRM but no other execution is possible. The third input element is composed of a set of bindings, which are required for each candidate IUT that is part of the contract project. As previously discussed, bindings provide a mapping between an implementation independent TRM and a concrete IUT. Bindings can be specified in one of two ways: manually or automatically. Details regarding both methods of binding specification will be provided later in this chapter. Once each of the three input elements are provided; the VF compiles the TRM with the binding information to produce a single Contract Intermediate Language (CIL) listing representing the entirety of the TRM ready to be executed against the IUT. Details regarding the CIL will be presented shortly.

The runtime portion of the VF executes the CIL against a candidate IUT. The execution process begins with static analysis. That is, checks that do not require execution of the IUT are executed against the IUT. Static checks are provided by third-party modules known as plug-ins. The result of executing each static check is displayed in the CER. Once execution of any static checks has been completed, the IUT is launched in a separate process. The VF attaches itself to the newly created process, and the IUT is allowed to execute. As execution of the IUT progresses, the VF monitors its execution path and compares it to the execution grammars specified within the TRM in the form of scenarios and relations. Furthermore, as methods within the IUT are invoked, the VF executes any pre- and post-conditions as well as invariants defined within the TRM. In addition to the built-in checks provided by the VF, the user can also provide any number of domain-specific dynamic checks that are executed alongside the IUT. The result of executing each check, both built-in and user supplied, is written to the CER.

Once the IUT has completed execution, domain-specific metric evaluators are used to analyze and report on metric information gathered during execution of the IUT. The results of such metric evaluation are written to the CER. Once all metric analysis has been completed, the CER is displayed and includes all of the information written to it during execution of the TRM against the candidate IUT. Generation of the CER completes the VF's execution process. We will now outline the implementation technology used to implement the VF, before looking at each aspect of the VF in detail.

4.2 Implementation Technology

Implementation of the Validation Framework (VF) was performed using Microsoft's .NET Framework [141]. The .NET framework is a managed programming model for building applications on Windows clients, servers, and mobile or embedded devices. Microsoft has standardized the .NET virtual machine and the C# programming language in ECMA standards 335 and 334 respectively [63, 98]. The .NET framework also has an open source implementation known as the Mono Project [186]. Mono implements both ECMA standards allowing for the development and execution of .NET applications on Linux [184], Solaris [176], Mac OS X [7], Windows [143], and Unix [191]. The Mono .NET runtime is sponsored by Novell [155].

Due to the VF's direct integration with the .NET virtual machine, the use of Microsoft's Phoenix Research Development Kit [144], and Visual Studio [137] integration, the VF has been implemented and validated using the runtime provided by Microsoft[8]. However, it should be noted that elements of the framework, such as the ACL compiler and core runtime, could be ported to the Mono runtime with little effort. Developers can also use the Mono runtime for implementation of binding extension modules, static checks, dynamic checks, and metric evaluators.

Implementation of the VF itself is contained within 1,355 classes totaling over 260,000 lines of mostly C# code. Approximately five percent of the VF is implemented

[8] Before the release of the Visual Studio 2008 SDK and the Phoenix RDK, an early test version of our framework was based upon our own editor, portable executable file reader, disassembler, and instrumentation facilities [13]. We have chosen to use Visual Studio and Phoenix as they provide state-of-the-art implementations in these areas. However our own editor and backend support allows us to avoid being tightly coupled to either Visual Studio or Phoenix.

using C++. The C++ code is required in areas where managed code cannot be used, such as advanced memory management and low-level execution of the IUT.

4.2.1 Java and the Eclipse Platform

Many elements of the VF could be implemented using Java [175] and integrated into the Eclipse platform through the use of a plug-in [180]. However, due to some of the low-level activities performed by the VF, direct modification of the Java Virtual Machine (VM) would be required. That is, we would have to supply a specialized Java VM with the VF. Given this reason, as well as the support provided by Phoenix, and our existing C#/.NET experience, the .NET platform was selected for implementation. The following sections will present Phoenix and the add-in mechanism provided by Visual Studio 2008. Both technologies have been used during implementation of the VF.

4.2.2 Microsoft Phoenix

Phoenix is being developed by Microsoft Research and is the code name for the software optimization and analysis framework that will be the basis for all future Microsoft compiler technologies [144]. Phoenix is a framework for building compilers and tools for program analysis, optimization, and testing. Phoenix is an open, extensible environment designed to meet the needs of two audiences: researchers and developers [144]. Phoenix provides researchers with a modular well-defined infrastructure for the creation and development of tools and compiler elements. Production developers can use the Phoenix back-end for advanced code generation and optimization. Figure 28 provides a conceptual overview of the Phoenix platform.

As shown by the orange rectangle in Figure 28, the core of Phoenix is the Intermediate Representation (IR). The IR is a strongly typed linear representation. It is used to represent the instruction stream of a method as a series of dataflow operations. The IR represents each method at multiple levels of abstraction, from a high-level machine-independent representation, to a low-level machine dependent representation. The IR explicitly represents all the control flow and dataflow of an instruction stream.

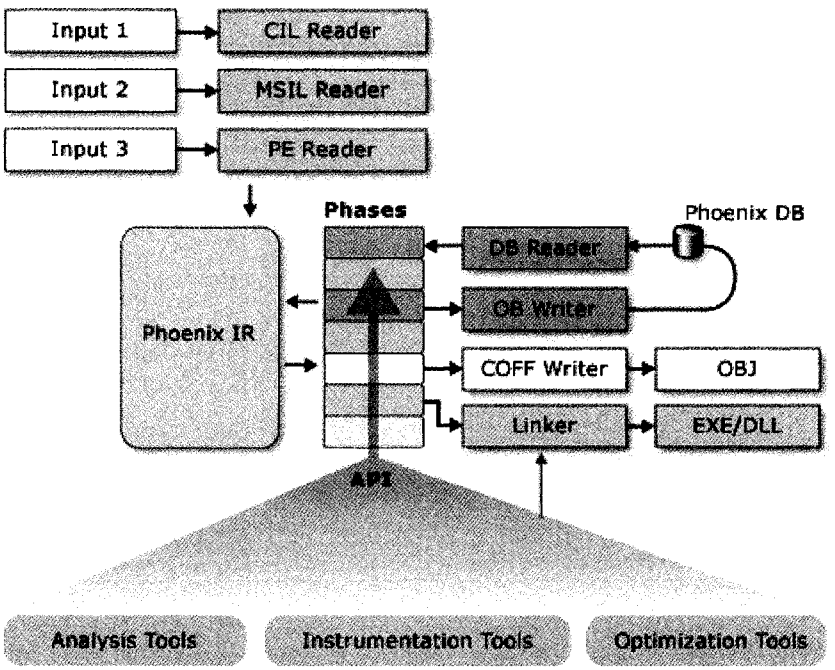

Figure 28 - Phoenix Conceptual Overview [144]

The VF uses the Portable Executable (PE) and Microsoft Intermediate Language (MSIL) readers provided by Phoenix to open and load binary candidate IUTs. The readers then translate the IUT into Phoenix IR. The VF uses the IR to perform binding activities and execute static checks. The following subsections will illustrate how Phoenix is used to aid with bindings and with the execution of static checks.

4.2.2.1 *Bindings*

The IR representation includes a symbol table to represent structural components of the IUT. Phoenix associates symbols with entries such as variables, labels, types/classes, method names, addresses, metadata entries, and module import/exports [144]. Together, these symbols provide a complete structural representation of the IUT, including the relationships among program entities. The symbols gathered by Phoenix are used by our Automated Binding Engine (ABE) to infer bindings. In the case where a binding cannot be inferred automatically, the ABE uses the symbols provided by Phoenix to prune the list of possible bindings presented to the user for selection. Details regarding how the ABE infers bindings between the Testable Requirements Model (TRM) and candidate IUT will be provided later in this chapter.

4.2.2.2 *Static Checks*

Recall that static checks are evaluated by examining the structural composition of the candidate IUT. That is, the IUT is not executed during the evaluation of a static check. Static checks operate by executing a side-effect free query on the structure of the IUT. Such execution translates into executing a query on the Intermediate Representation (IR) representation of the IUT provided by Phoenix. The IR provides type information, control flow, and dataflow representation of the IUT at both the method and global levels.

Creators of static checks have access to the entire IR tree for the types and methods that the static check is applied to. The static check creator is able to directly query the

IR in order to perform their requested check. An example of such a static check is provided in Section 4.7.1.

4.2.3 Side-Effects

As previously stated static checks, dynamic checks, and metric evaluators are side-effect free. In addition, as stated in Chapter 3, IUT methods that are bound to observabilities within Testable Requirement Models (TRMs) are also required to be side-effect free. The Phoenix c2 compiler is able to process and analyze an IR representation to determine the side-effects of executing a set of instructions [144]. The information generated by the c2 compiler is used by our framework to ensure that all checks and observability methods are indeed side-effect free.

Details regarding how the c2 compiler operates and how the VF analyzes the information provided by the compiler are very low-level and highly technical. However, because the ability to determine if a given IUT method or user supplied check is in fact side-effect free is essential to the operation of the VF; the following subsection will provide a technical example. Readers who are not interested in the technical details are encouraged to skip the subsection and continue with how the VF is integrated into Microsoft Visual Studio.

4.2.3.1 Using Phoenix to Capture Side-Effects

The IR can be used to describe a method using several levels of abstraction, from machine-independent high-level to machine dependent low-level. Phoenix provides four levels of IR abstraction: high-level IR (HIR), mid-level IR (MIR), low-level IR (LIR), and encoded IR (EIR) [144]. Phoenix uses compilation phases to transform IR from one level

of abstraction to another. The process of moving from high-level IR to lower-level IR is known as *lowering* and the inverse is known as *raising* [144].

When performing binding operations and executing static checks the HIR is already sufficiently low-level. As an example consider the C++ program in Figure 29. The program declares two integer variables and a pointer to one of them. The program then adds the first variable to the value referenced by the pointer and prints the result to the console.

The resultant HIR produced by the c2 compiler is shown in Figure 30. The specific elements of the HIR are not of importance here. Notice that no specific computer architecture or runtime dependencies are represented in HIR. The HIR does describe the flow of operations and the use of a temporary variable for dereferencing the pointer. In general the HIR will provide enough information for binding and static check execution. The HIR will not however provide enough information to determine the existence of side-effects.

```
#include <stdio.h>

int main(int argc, char** argv)
{
   int x = 5;
   int y = 3;
   int* p = &y;

   x = x + *p;

   printf("%d\n", x);
}
```

Figure 29 - Example C++ Program for Phoenix

```
$L1: (refs=0)
   {-4}, {-7}          = START _main(T)
_main: (refs=1)
   _argc, _argv        = ENTERFUNC
   _x                  = ASSIGN 5
   _y                  = ASSIGN 3
   _p                  = ASSIGN &_y
   t275                = ASSIGN _p
   t276                = ADD _x, [t275]*
   _x                  = ASSIGN t276
   {-9}                = CALL* &_printf, &$SG3674, _x, {-9}, $L5(EH)
                         RET 0, $L3(T)
$L5: (refs=1)
                         UNWIND
$L3: (refs=1)
                         EXITFUNC
$L2: (refs=0)
                         END {-4}
```

Figure 30 - HIR Generated by Phoenix

Phoenix produces LIR by using a sequence of lowering processes that include machine instruction selection, code generation, and register allocation. Figure 31 illustrates the LIR representation of the C++ program from Figure 29 and the HIR representation in Figure 30. The HIR represents a machine-independent view of the C++ program, where the LIR version represents a machine-dependent view. To move between the two views, Phoenix adds machine-dependent opcodes and attributes to instructions. The LIR listing is generated using x86 as the target architecture. Therefore, the LIR uses x86 registers such as EAX and ESI to represent the operands used in the ADD instruction, and the CCFLAGS register to represent the implicit side-effects of the instruction. The information provided by the LIR listing gives our VF the ability to determine if a given method within the IUT is side-effect free, and to ensure that no plug-in generates a side-effect within the IUT.

In order to aid with the development of static checks and the specification of bindings, the VF includes an Intermediate Representation (IR) viewer. The IR viewer is

able to display all four levels of IR generated by Phoenix. The IR viewer is opened within the VF when a candidate IUT is double-clicked in the Solution Explorer. Figure 32 provides a screen shot of the IR viewer provided by the VF.

```
$L1: (refs=0)
   {-4}, {-7}          = START _main(T)
_main: (refs=1)
   _argc, _argv        = ENTERFUNC
   [ESP], {ESP}        = push EBP
   EBP                 = mov ESP
   ESP, EFLAGS         = sub ESP, 12(0x0000000c)
$L9: (refs=0) Offset: 6(0x0006)
                         ENTERBODY
   _x[EBP]             = mov 5
   _y[EBP]             = mov 3
   tv279-(EAX)         = lea &_y[EBP]
   _p[EBP]             = mov tv279-(EAX)
   tv275-(EAX)         = mov _p[EBP]
   tv276-(ECX)         = mov _x[EBP]
   tv276-(ECX), EFLAGS = add tv276-(ECX), [tv275-(EAX)]*
   _x[EBP]             = mov tv276-(ECX)
   [ESP], {ESP}        = push _x[EBP]
   [ESP], {ESP}        = push &$SG3674
   {-9}, {EAX ECX EDX ESP EFLAGS MM0-MM7 XMM0-XMM7 FP0-FP7 FPUStatus} =
                         call* &_printf, $out[ESP], $out[ESP]+32, {-9},
                         {EAX ECX EDX ESP EFLAGS MM0-MM7 XMM0-XMM7
                         FP0-FP7 FPUStatus}, $L5(EH)
   ESP, EFLAGS         = add ESP, 8
   tv283-(EAX)         = mov 0
$L3: (refs=0) Offset: 58(0x003a)
                         EXITBODY
   ESP                 = mov EBP
   EBP, {ESP}          = pop [ESP]
   {ESP}               = ret {ESP}, $L10(T)
$L5: (refs=1) Offset: 62(0x003e)
                         UNWIND
$L10: (refs=1) Offset: 62(0x003e)
                         EXITFUNC tv283-(EAX)
$L2: (refs=0) Offset: 62(0x003e)
                         END {-4}
```

Figure 31 - LIR Generated by Phoenix

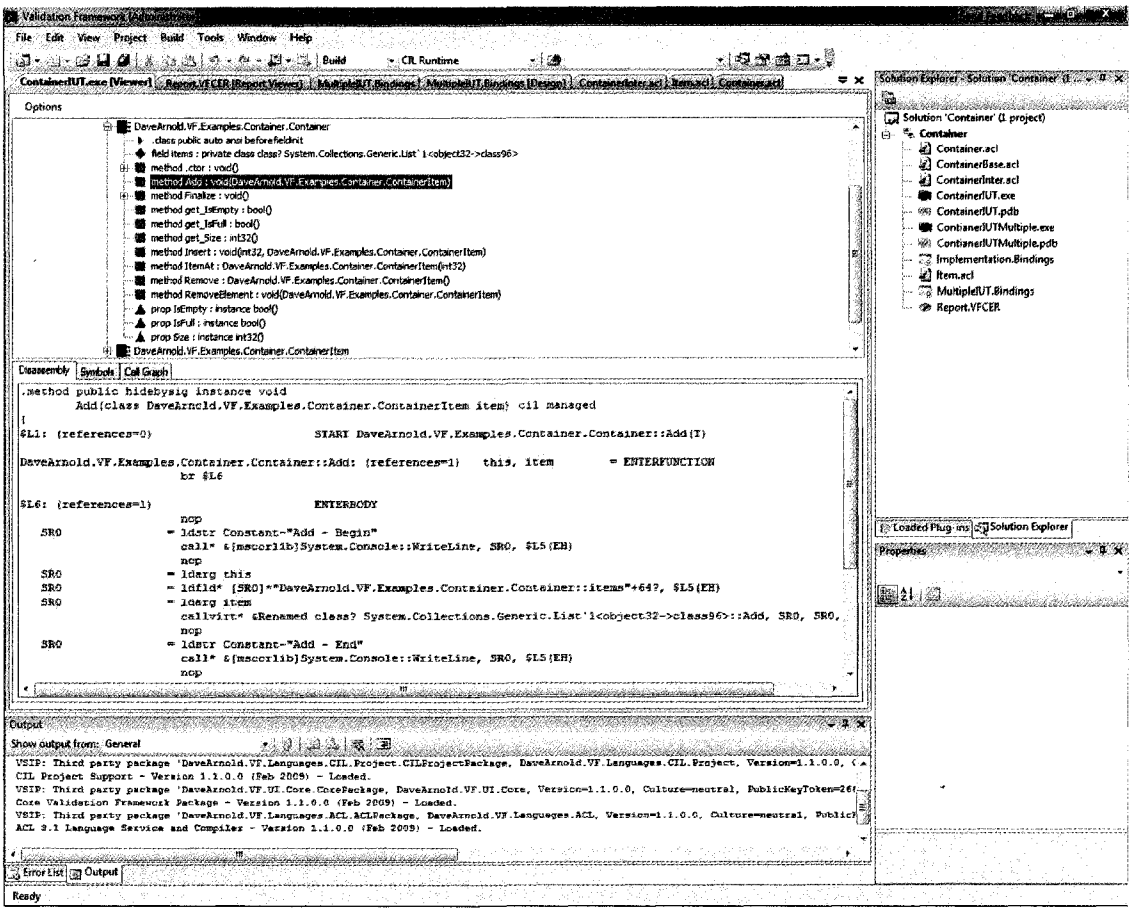

Figure 32 - Validation Framework IR Viewer

4.2.4 Microsoft Visual Studio Integration

In order to better apply the VF to the development process it has been fully integrated with Microsoft Visual Studio [137]. Visual Studio is a complete set of development tools for building ASP.NET web applications, XML web services, desktop applications, and mobile applications [136]. Visual Basic, C++, and C# all use the common Integrated Development Environment (IDE) provided by Visual Studio. The 2008 release of Visual Studio includes a Software Development Kit (SDK) for extending the IDE. Such extension allows for the creation of new languages, tools, and views that operate alongside existing Visual Studio artifacts. Using the Visual Studio SDK to implement the VF, we have taken advantage of syntax highlighting and context sensitive

completion for both the Another Contract Language (ACL) and Contract Intermediate Language (CIL). In addition, the Visual Studio SDK allows the VF to have a similar look and feel as the standard Visual Studio IDE. This look includes, project support, output and error list windows as shown above in Figure 32. Our VF IDE also includes integration of the Automated Binding Engine (ABE), and the Contract Evaluation Report (CER) viewer. The Validation Framework Installation and Usage guide includes a listing and examples of all views provided as part of the VF [23]. We will now begin our discussion of each section of the VF, starting with the ACL a language we have created for the specification of Testable Requirements Models (TRMs).

4.3 Another Contract Language

As we have shown in Chapter 3, the Another Contract Language (ACL) is a high-level general purpose contract specification language. The ACL is used for the specification of a TRM and serves as one of the three input elements that the VF requires. Each TRM is composed of one or more ACL contracts. Through the examples in Chapter 3, we have shown that the ACL is non-diagrammatic and proceeds from the scenario contracts proposed by Nebut et al. [153]. Their scenario contracts were chosen not only for their testability but also for their handling of inter-scenario relationships. Whereas they augment scenarios with pre- and post-conditions, we further enhance scenarios and responsibilities with static checks, dynamic checks, and metrics in order to define the contracts of a system.

The ACL itself is a strongly-typed language that is decoupled from any specific implementation technology. Such decoupling allows a TRM specified in ACL to be

applied to several candidate IUTs without any modification. As illustrated in Chapter 3, the central element to the ACL is a contract. A contract is applied to a type. The type is not defined within the ACL, but rather is part of the IUT. Each contract may contain several elements that describe responsibilities, scenarios, static checks, dynamic checks, metric evaluators, and binding rules. Examples of each element used to compose a contract have already been presented in Chapter 3, and will not be repeated here.

As contracts are applied to a type defined within the IUT, they are restricted to express constraints and execution grammars that are centered on a type. To express relations between contracts and to specify constraints on their execution the ACL defines the notion of an interaction. Interactions allow for the definition of inter-scenario relationships, where contracts allow for intra-scenario relationships. Put another way, an interaction can be viewed as a use case diagram, denoting the relationships between use cases (contracts). A TRM may contain any number of interactions. Each interaction is automatically applied to the candidate IUT that is being executed against the TRM. Unlike contracts, interactions are not bound to types within the IUT. Rather, an interaction is a standalone ACL element that does not have an IUT counterpart. An example of an interaction and its corresponding relation section was presented in Chapter 3, and will not be repeated here.

The ACL defines the entirety of the TRM, no additional information is required for the representation of a TRM. For a detailed description and explanation of the ACL, its syntax, and semantics see the ACL Specification Document [8].

4.4 Implementation Under Test

The Validation Framework (VF) executes a Testable Requirements Model (TRM) specified using the Another Contract Language (ACL) against a candidate Implementation Under Test (IUT). The candidate IUT is the second required input element. A single TRM can be executed against multiple candidate IUTs without modification. Implementations are not created within the VF. Each IUT is provided in binary form. That is, source code for the IUT is not required. The VF uses the aforementioned Phoenix [144] technology to open the IUT and automatically construct both structural and behavioural models that are used during execution of the TRM. Precise execution details will be provided shortly. The VF supports IUTs implemented using three different implementation technologies. The following subsections will provide a brief discussion of each implementation technology supported by our VF.

4.4.1 .NET Executables

Any executable targeting the .NET runtime [141] is supported by the VF. That is, any system implemented using a .NET compatible language (i.e., C#, VB.NET, etc.) or a language for which a compiler targeting the .NET runtime exists. Such support is possible because the .NET runtime executes a common intermediate language known as the Microsoft Intermediate Language (MSIL) [63]. Phoenix is able to open and load any binary implementation that is represented in MSIL. A .NET executable is a standalone application that is executed by launching the Portable Executable (PE) (.exe) file. That is, the VF executes the IUT by launching its PE file.

4.4.2 C++ Executables

Unlike .NET executables, applications written in C++ are compiled directly to machine language [99]. Elements such as type and variable names are lost in the code generation process. Phoenix is able to open an executable written in C++ as long as a set of debug symbols are provided in Program Database (PDB) format. The required debug symbols are automatically generated by Visual Studio when an application is compiled in debug mode [137]. Thus, the VF is able to support executables written in C++ for which debugging information is provided. The VF executes this type of IUT by launching its PE file.

4.4.3 ASP.NET Web Applications

In addition to the standard executables discussed in the previous two sections, the VF also supports web applications implemented using Active Server Page (ASP) .NET technology. The result of such web applications is one or more web pages (.aspx) and a Dynamic Link Library (DLL). The DLL is represented with MSIL, and the web pages are represented with Hyper Text Markup Language (HTML). The VF executes an ASP.NET web application by launching a specialized web server that we have implemented as part of the VF. The specialized web server allows an ASP.NET web application to execute against the TRM. Figure 33 illustrates execution of our VF web server. When the VF web server is launched the user accesses the web IUT by navigating through its web (.aspx) pages. The actions of the user represent the test driver for the IUT. Once the user has finished executing the IUT, the *Stop Server* button is clicked and the resultant Contract Evaluation Report (CER) is generated.

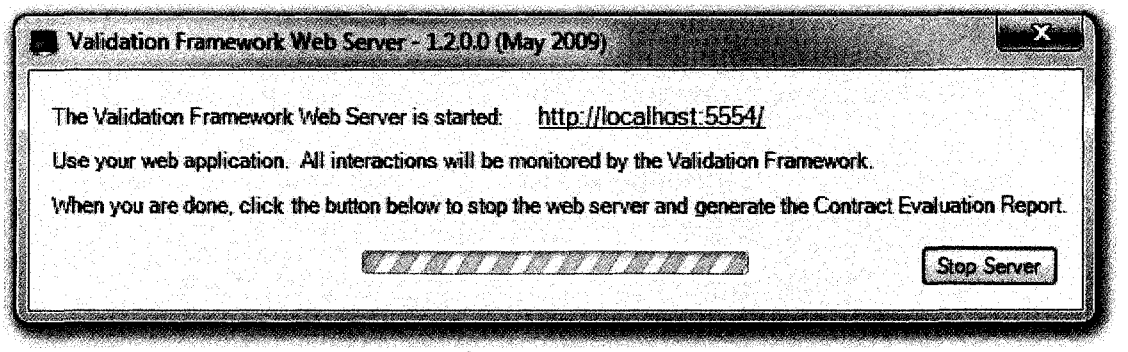

Figure 33 - Validation Framework Web Server

Regardless of the type of candidate IUT provided to the VF, a unique set of bindings is required to bridge the gap between an implementation independent TRM and a concrete candidate IUT. It is these bindings that make up the third and final input element to our VF.

4.5 Bindings

In order to execute a candidate IUT against a set of contracts and interactions (i.e., the Testable Requirements Model (TRM)) a set of bindings is required. Bindings represent a mapping between various elements within a contract and structural elements within the IUT. A set of bindings is represented by an XML file that contains tags linking contract elements to their IUT counterparts. Each IUT that is to be executed against a TRM must have a corresponding binding file.

Rather than having to directly edit the XML binding code, we have integrated a binding tool into the VF. The binding tool provides a graphical way to view and specify the binding information. Figure 34 provides an example of our binding tool. As illustrated in Figure 34, the binding tool consists of three views all comprised within the binding tool window. The following subsections will present each of the three views.

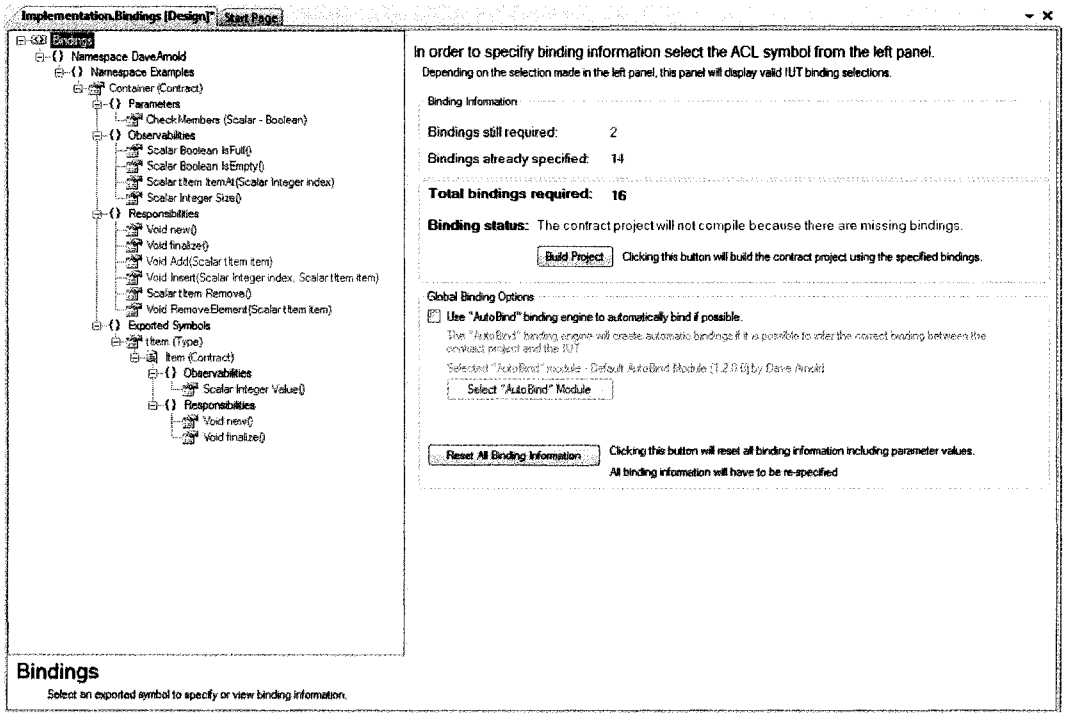

Figure 34 - Validation Framework Binding Tool

4.5.1 The Binding Tree

The binding tree displays each contract element that must be bound to an IUT counterpart. These elements include: contracts, parameters, observabilities, responsibilities, and exported types. The contents of the binding tree are generated by the ACL compiler as it compiles a TRM. The binding tree for the container example in Chapter 3 is shown on the left side of Figure 34. Each element within the binding tree is grouped according to namespace, contract, and type. Once the grouping is complete, a colour is assigned to each element denoting the binding status. Colours are assigned as follows:

- Black - Grouping element, no binding required.
- Green - Binding complete, the element has been bound to an IUT structural element.

- Red - Binding incomplete, the element has not yet been bound.
- Gray - Binding prerequisite required, the element requires another element to be bound before the current element is bound.
- Purple - Binding not required, the element's binding is based on the binding of a different element.

Each time binding data is modified, the binding tree is refreshed, and the colours are updated. To view or specify a binding, select an element within the tree. When an element is selected, the other two binding views are updated to reflect the selection. Every item that is displayed within the tree can be selected.

4.5.2 Binding Status

Below the binding tree is a small status pane shown in Figure 34. The binding status displays information pertaining to the currently selected tree item. The displayed information indicates if the selected item is bound, and if so, the corresponding IUT structural element. The binding status pane provides a quick way to see which IUT element is bound to a given artifact within the TRM.

4.5.3 The Binding Viewer

The binding viewer is located to the right and on top of the binding status pane. The binding viewer displays binding information for the item currently selected in the binding tree. The binding viewer allows for the specification, modification, and deletion of a binding. There are 37 possible binding views that can be displayed, depending on the selected item. The following subsections will examine some of the 37 views. The omitted views are rather simple and self-explanatory.

4.5.3.1 Main View

When a file of bindings is first opened, the main view is displayed as shown in Figure 34. The main view is also displayed when the top node, *Bindings*, is selected in the tree view. The main view begins with an overview of the number of bindings that are still required, the number of bindings that have already been specified, and the total number of bindings. In the case of the Container example from Chapter 3, there are a total of 16 bindings and all of the bindings have been specified. The *Build Project* button can be used to invoke the ACL compiler to begin the process of executing the TRM against a candidate IUT using the specified set of bindings.

Below the *Build Project* button is a set of binding options. The binding options apply to all elements contained within the binding file. The first option is to use the Automated Binding Engine (ABE) or *AutoBind* for short.

4.5.3.1.1 Binding Inferences

The ABE allows for the automatic inference of bindings between a Testable Requirements Model (TRM) and a candidate Implementation Under Test (IUT). The ABE does not use a specific binding algorithm. Rather, it is open in nature, and uses extension modules that are developed using a Software Development Kit (SDK) [13]. As bindings provide a mapping from the TRM to a candidate IUT, details regarding the structure of the IUT are required. Obviously, these details are implementation technology-specific, and as such different binding algorithms may be required. As an example, consider when an obfuscator is applied to the IUT. The purpose of an obfuscator is to apply one or more transformations to the IUT without affecting its

functionality, while making it difficult or impossible to understand any structural information recovered from the IUT. In this case, a specialized binding extension module would be needed to reverse the transformation before any binding inference could be performed. Furthermore, having an open binding system allows for experts in the domain of type inference to enrich our Validation Framework (VF).

Practically, each extension module is implemented as a DLL and is placed in a specific location relative to the *VF.exe* file. The SDK provides an interface that must be implemented within the DLL for it to be used as an ABE extension module. The interface is defined in Figure 35. The interface begins with four read-only accessors that must be implemented. The first, *Name*, specifies the name of the binding extension module. The module's name is displayed in the main binding view and in the module selection dialog box. Next, the *Author* accessor specifies the author of the given module. The *Version* accessor denotes the version number of the current module. Finally, the *Description* accessor provides a short description of the binding extension module that is shown in the module selection dialog box.

```
public interface IAutoBind
{
   string Name { get; }
   string Author { get; }
   Version Version { get; }
   string Description { get; }

   List<IIUTNode> FilterForContract(IContractBindPoint bindPoint,
      IIUTNode rootNode);
   List<IIUTMethodMap> FilterForObservability(
      IObservabilityBindPoint bindPoint, IIUTNode rootNode);
   List<IIUTMethodGroupMap> FilterForResponsibility(
      IResponsibilityBindPoint bindPoint, IIUTNode rootNode);
   List<IIUTNode> FilterForField(IFieldBindPoint bindPoint,
      IIUTNode rootNode);
   List<IIUTNode> FilterForMethod(IMethodBindPoint bindPoint,
      IIUTNode rootNode);
   List<IIUTNode> FilterForType(ITypeBindPoint bindPoint,
      IIUTNode rootNode);
}
```

Figure 35 - IAutoBind Interface

Following the read-only accessors, six methods that are invoked by the Automated Binding Engine (ABE) to infer the given binding type are defined. Each method accepts two parameters; the first is an interface representing information about the Testable Requirements Model (TRM) element for which a binding is requested. The second parameter represents the starting point within the structural representation of the IUT where the inference should begin. For example consider the *FilterForObservability* method, the second parameter would be the structural node (provided by Phoenix) representing the IUT type bound to the contract where the given observability (first parameter) is defined. Each method returns a list of candidate bindings. If the resultant list contains a single element, then a binding is automatically made between the given TRM element and the lone IUT element within the resultant list. Otherwise, the binding is left for manual specification. For additional information regarding implementation of the *IAutoBind* interface and an example module see the SDK [9].

The VF automatically loads and registers any ABE extension modules it encounters. Under the *AutoBind* check box in the main binding view, the name of the currently selected ABE module is specified, along with the *Select AutoBind Module* button. The button when clicked allows for the selection of the ABE extension module that will be used. Only one extension module can be used at a time. However, multiple modules can be used successively. That is, one module could be selected to infer as many bindings as possible, then a second module could be selected to infer any bindings not recognized by the first module.

We have implemented two such extension modules as part of the VF. The first binding module takes into account type and method names, where the second module uses only structural information such as return type and parameter type/ordering to infer a binding. Our first binding module uses the following approach to infer bindings:

First, each contract within the TRM is bound to a type within the IUT. The ABE uses Phoenix to examine all types defined within the IUT. The types are compared by name and structure to determine the correct binding. Structural comparison entails looking for methods within the type to determine if the observabilities and responsibilities defined within the contract could also be bound.

Once all of the contracts have been bound, the ABE binds observabilities. As each observability represents an observation requirement imposed on the IUT, the corresponding IUT method must be side-effect free. That is, invocation of the IUT method bound to the observability must not alter the state of the IUT. To enforce this,

the ABE uses Phoenix and the technique outlined in Section 4.2.3 to ensure any candidate IUT method is indeed side-effect free. Once a set of candidate query-methods is selected, method name, return type, and parameters (number, types, and order) are all examined to select a corresponding IUT method for binding.

Following the observability bindings, the ABE binds responsibilities. Each responsibility is bound to one or more IUT methods. The ABE begins by looking for a single method that can be bound to the responsibility. The method name, return type, and parameters (number, types, and order) for each IUT method are examined to find a corresponding match. If an individual IUT method cannot be located, the ABE begins to analyze groups of methods that could be used in combination to create the required responsibility binding. Once all contracts, observabilities, and responsibilities are bound, the ABE will bind any remaining exported symbols using the same methodology for binding contracts to IUT types.

The Automated Binding Engine (ABE) uses the selected extension module to infer a correct binding between each Testable Requirements Model (TRM) element and a corresponding IUT counterpart. The correctness of a binding is determined by the binding results displayed within the aforementioned binding tree. If at any point the selected ABE module is unable to determine a binding for an element within the TRM, the module will skip the binding and move on. Once the automatic binding process is complete, any unspecified or incorrect bindings can either be specified manually or specified through a different ABE module. We will illustrate the manual binding process

in the next section. Each of our two implemented ABE modules have correctly bound approximately 95% of the required bindings found in our five case studies (approx. 200 bindings).

There are a few things to note about the ABE before continuing. First, each time the binding data is updated, the ABE will run to see if the updated binding data allows for additional bindings to be inferred. That is, if a binding is completed manually, it is possible that the ABE will infer several additional bindings at the same time. Secondly, because the ABE runs each time the binding data is updated, it is possible that if a binding is removed, the binding will be reestablished instantly because the ABE found a match. Finally, once a binding has been established, either manually or via the ABE, future executions of the ABE will not change the binding. That is, the ABE only operates on bindings that have yet to be specified.

If the *AutoBind* check box is selected on the main view (see Figure 34) the ABE will use the currently selected extension module to infer the bindings, otherwise all bindings must be manually specified. We will now discuss some of the other binding views used for the manual specification of bindings.

4.5.3.2 Contract View

The contract view is displayed when a contract is selected in the binding tree. The contract view allows a contract to be manually bound to a type defined within the IUT and is shown in Figure 36. The contract view displays a list of types that are defined within the candidate IUT. To make or change a binding the user can double-click on the

requested type. As shown in Figure 36, under the list there is a check box named *Show Recommended Bindings Only*. If the box is checked the currently selected ABE module is used to prune the list of IUT types displayed in the list. The algorithm used to prune the IUT type list is dependent on the selected Automated Binding Engine (ABE) module.

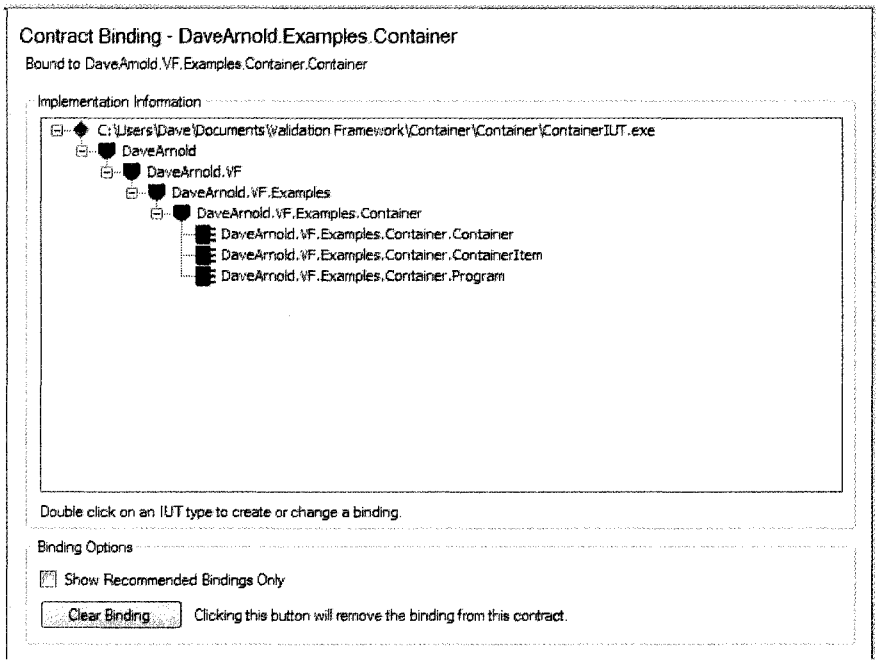

Figure 36 - Binding Tool: Contract View

4.5.3.3 Observability View

The observability view shown in Figure 37 is displayed when an observability is selected in the binding tree. As observabilities reside within contracts and may accept parameters, sometimes the prerequisite view is displayed instead. The prerequisite view (not shown) indicates the names and types of contract symbols that must be bound before the selected observability can be bound. Once any prerequisite bindings have been completed, the observability view is displayed.

The observability view only displays methods that reside within the type bound to the containing contract. That is, as a contract is bound to a type within the IUT, any elements defined within the body of the contract must be bound to elements within the corresponding IUT type. In addition, the binding tool will remove any methods that do not have the required parameters, return types, or methods that generate side-effects. A binding is created or changed, by double-clicking on the desired IUT method.

If the selected observability does not accept any parameters, the binding is completed after double-clicking on the desired IUT method. However, if there are parameters in either the observability or the bound IUT method, a second view is displayed for parameter specification.

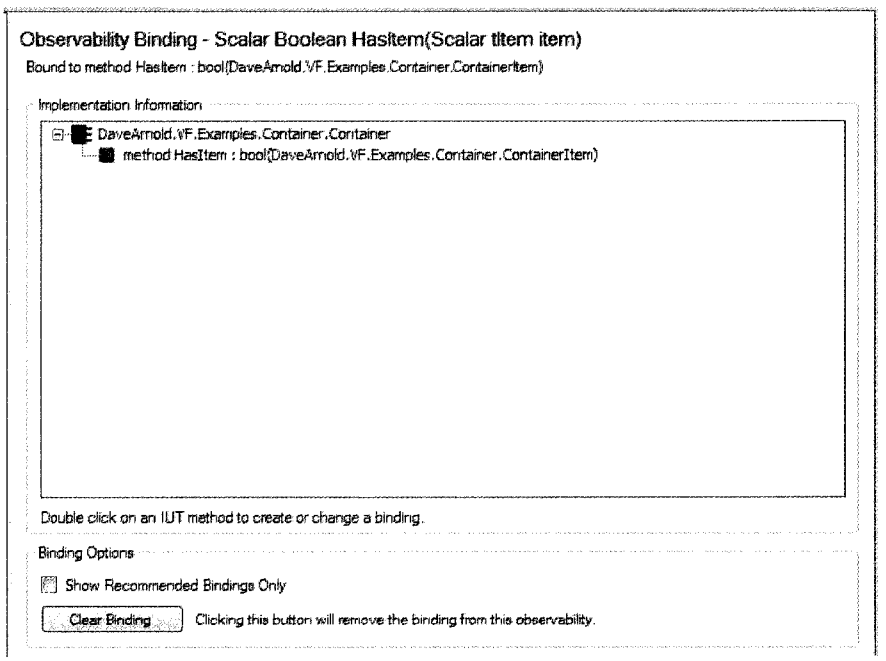

Figure 37 - Binding Tool: Observability View

4.5.3.4 Parameters View

Figure 38 displays the observability parameters view, the view is used for the specification of a parameter mapping between the parameters of the observability and the selected IUT method. Each parameter accepted by the IUT method must either be mapped to a parameter provided to the observability or be directly hardwired by the binding tool. Hardwired parameters can be used if the observability does not have enough parameters to support the selected IUT method.

Figure 38 contains two tree views. The view on the left represents each parameter within the selected IUT method, and the view on the right represents each parameter within the observability. IUT parameters displayed in green have already been bound, red indicates a binding is still required. There are two ways to bind an IUT parameter. The first is to select the IUT parameter and double-click on an observability parameter to create a binding. Once the parameter binding has been created, the IUT parameter will turn green. Note that only observability parameters whose type matches the selected IUT parameter can be selected. Parameters that do not match are grayed out. The second way to bind an IUT parameter is to select the parameter, and then type a fixed value for the parameter into the *Hardwire Parameter Box*. Once the value is entered, it is checked against the required type, and if the types match, the IUT parameter will turn green. Once all of the required IUT parameter bindings have been created the *Save Bindings* button will be enabled, clicking it will save the completed observability binding.

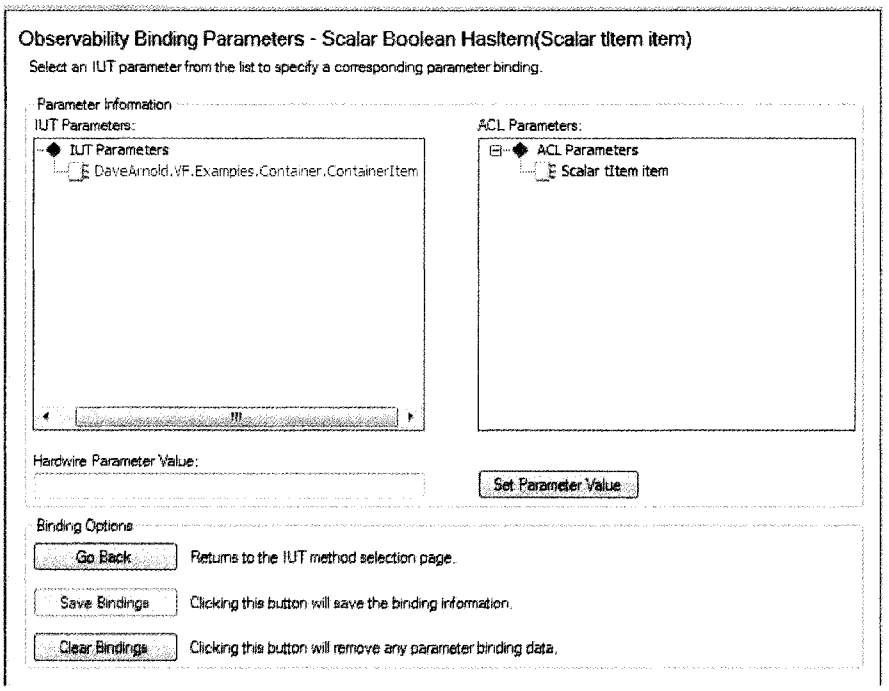

Figure 38 - Binding Tool: Observability Parameters View

4.5.3.5 Responsibility View

When a responsibility is selected in the binding tree, the binding view shown in Figure 39 is displayed. Like with observabilities the prerequisite view will be displayed if either, the containing contract, the responsibility's return type or one of the responsibility's parameter types has yet to be bound.

In the responsibility view only methods that reside within the type bound to the responsibility's containing contract are displayed. Unlike observabilities, all methods defined within the contract are displayed. The rationale here is that because a responsibility can be bound to a group of methods, rather than just a single method, all methods are displayed. An Implementation Under Test (IUT) method can be added to the responsibility binding by double-clicking the method name. Once a method is added to the binding list, the *Move Up*, *Move Down*, *Remove* and *Remove All* buttons can be

used to order the methods accordingly. The first method specified in the binding list must contain sufficient parameters to support the responsibility definition. The last method must have the return value required by the responsibility. Such conditions are checked when the *Set Binding* button is clicked. If the selected responsibility does not have any parameters, the binding is completed after clicking the *Set Binding* button. However, if there are parameters a second view is displayed for the parameter binding specification.

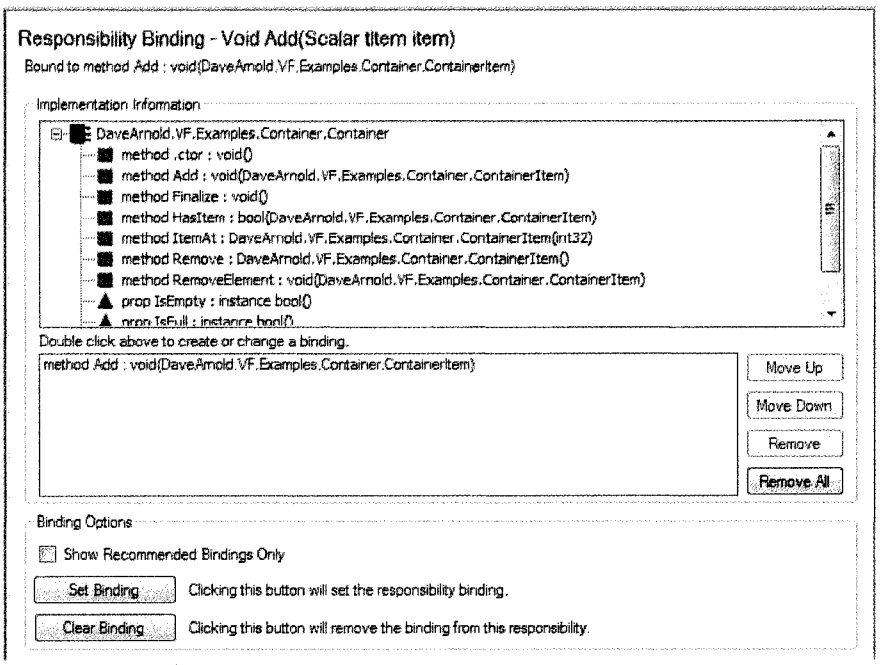

Figure 39 - Binding Tool: Responsibility View

4.5.3.5.1 Parameters View

The responsibility parameters view is used to specify a binding between parameters of a responsibility and the selected IUT method. If the responsibility binding contains several IUT methods, the parameters are bound to the first IUT method only. The responsibility parameters view is displayed in Figure 40.

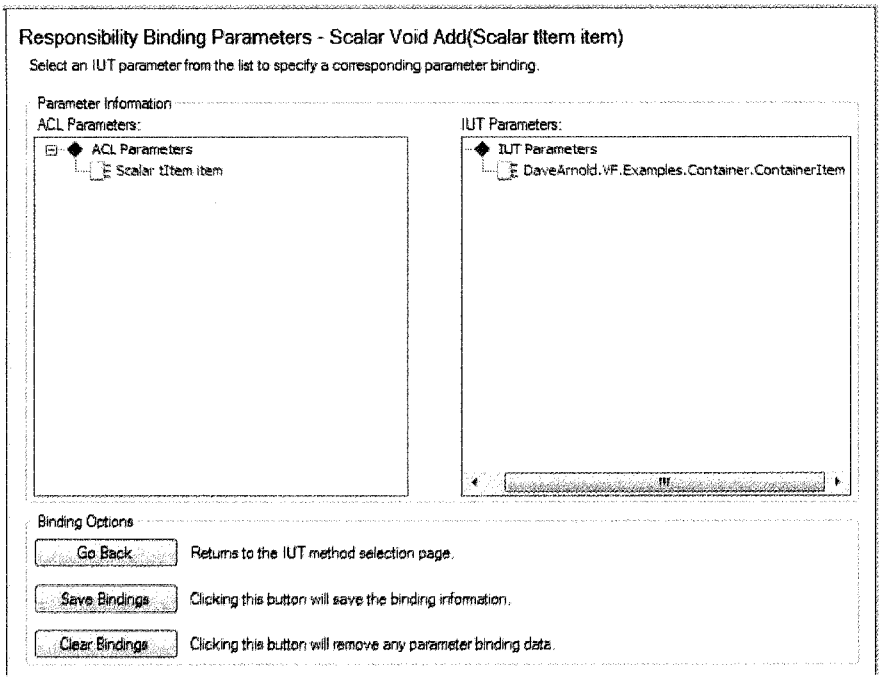

Figure 40 - Binding Tool: Responsibility Parameters View

The tree on the right represents each parameter within the first (or only) chosen Implementation Under Test (IUT) method, and the left tree represents each parameter within the responsibility. As with observability parameters green indicates a completed parameter binding, where red denotes a parameter that still requires binding information. A responsibility parameter is bound by first selecting the responsibility parameter, and double-clicking on a parameter supplied by the IUT method. Note that only IUT parameters whose type matches the selected responsibility parameter can be selected. IUT parameters that do not match are grayed out when a responsibility parameter is selected. Once all of the required responsibility parameter bindings have been created the *Save Bindings* button will be enabled, clicking it will save the completed responsibility binding.

4.5.3.6 Type View

The type view is displayed when an exported type is selected in the binding tree. Recall that an exported type is a type used within a contract that must be bound to an actual type within the IUT. The type view shown in Figure 41 allows for the specification of such a binding. The type view displays all of the namespaces and types defined within the IUT. A binding is made by double-clicking the requested IUT type. Again the *Show Recommended Bindings Only* checkbox can be used to allow the currently selected Automated Binding Engine (ABE) module to prune the number of types displayed.

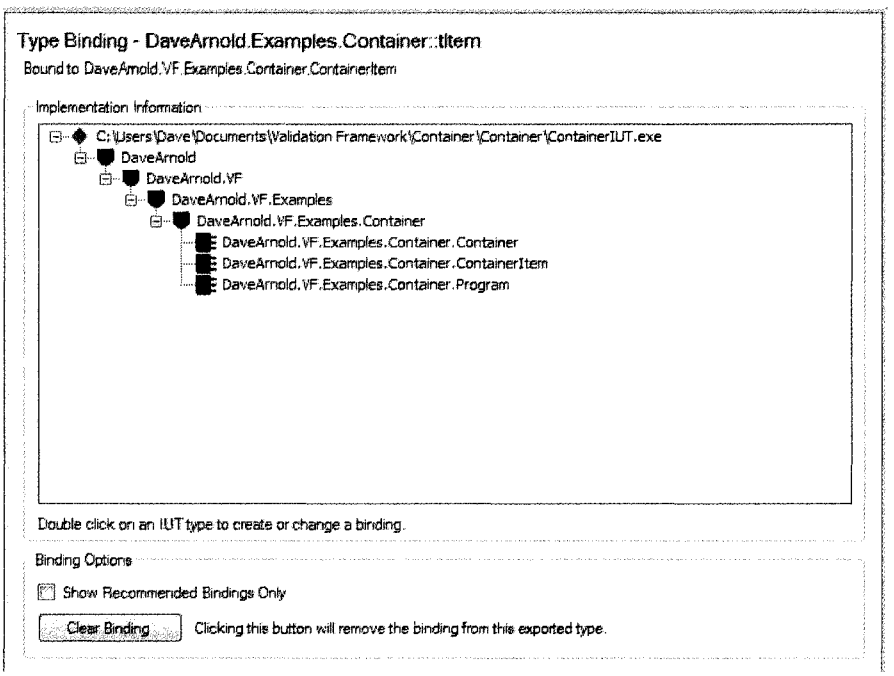

Figure 41 - Binding Tool: Type View

As discussed in Chapter 3, various binding conformances and rules can be applied to exported types. Our binding tool automatically applies any conformances and enforces all binding rules. The user will be notified if a binding is created that breaks a given rule specified within the contract and the binding will be aborted. Any contracts bound to

the IUT as a result of a conformance rule will automatically be bound. Recall that such bound contracts are displayed in purple on the binding tree.

4.5.3.7 Summary

We have presented some of the views that allow for display and manual specification of a set of bindings that bridge the gap between an implementation independent Testable Requirements Model (TRM) and a concrete candidate IUT. The ABE along with a selected extension module allow for automatic binding inference. Recall that a single TRM can be bound to, and thus executed against, *several* candidate IUTs. Each IUT would have a separate binding table that our VF uses during execution.

Once a set of bindings between a TRM and a candidate IUT are complete, the binding tables are saved. Saving the binding tables removes the need for re-specification each time the VF executes an IUT against the TRM. The binding data only needs to be re-specified if either the IUT or TRM are modified. If such a modification does occur, our binding tool only requires re-specification of the modified portions. With binding specification complete, the VF has all of the elements required for execution of a TRM against a candidate IUT. We will now discuss how such execution is performed.

4.6 Contract Intermediate Language

The process of executing a Testable Requirements Model (TRM) against a candidate Implementation Under Test (IUT) begins with the Another Contract Language (ACL) compiler compiling the TRM. The ACL compiler takes any ACL files that were used to specify the TRM along with the set of bindings and the IUT to perform semantic analysis.

The semantic analysis validates types used within the ACL against the actual types within the IUT as specified by the binding set. If there is an inconsistency between any ACL type and corresponding IUT type, or missing binding data is encountered the compilation fails and one or more errors are generated. The ACL compiler is able to generate 35 warnings and 713 errors depending on the type of error encountered. For a detailed explanation of each warning and error generated by the ACL compiler see the ACL Specification Document [8].

The result of a successful compilation is a single listing of contract intermediate instructions [14]. The contract intermediate instructions are specified using the Contract Intermediate Language (CIL) [14]. The CIL is a low-level stack-based language that combines the ACL representation of the TRM with the binding data in a single listing. The CIL is the language executed by the VF. The CIL serves as a layer of abstraction between the high-level TRM specification language and the VF runtime. The purpose of such an abstraction layer is to allow for the inclusion of other TRM specification languages and binding techniques into the VF without modification to the VF runtime. Other TRM specification languages could be graphical in nature and/or domain-specific. Ideas for such languages are briefly discussed in the future work section of Chapter 5. For additional information regarding the CIL see the CIL Instruction Set Document [14].

Following generation of a CIL listing by the ACL compiler, the CIL compiler performs syntactic and semantic analysis of the CIL instructions as a final check before TRM

execution against the candidate IUT. Analysis performed by the CIL compiler ensures that the CIL instruction set is valid and serves as a validation mechanism for our ACL compiler. The CIL compiler does not produce any concrete output, but rather directly feeds the VF runtime. That is, the result of a successful CIL compilation process is executed by the VF runtime. The execution process begins with the execution of any static checks defined by the TRM.

4.7 Static Checks

As previously stated, static checks are represented by code placed in the structure section of an ACL contract. Execution of all static checks is performed before the candidate IUT is launched. The execution of each static check begins by using Phoenix to gather structural and behavioural elements of the IUT. Once the structural and behavioural information is gathered, the actual static checks are executed. A static check is able to examine the structural and/or behavioural elements within the IUT to produce a result. Put another way, the execution of a static check acts like a query on the IUT. All static checks must produce a pass or fail result. That is, the success or failure of a check is not a rhetorical exercise open to debate. Instead, each check is executed against the candidate IUT, to generate a deterministic result. Such a pass or fail result is displayed in the Contract Evaluation Report (CER). The report is displayed following the execution of a candidate IUT against the compiled TRM.

To allow for an open and extensible set of static checks available to authors of TRMs, the VF employs a plug-in architecture for the implementation of static checks. That is, the VF does not provide a fixed set of static checks for use, but rather features an open

architecture so that experts in the domain of static testing can create specialized static checks that are executed by our framework. As such, the VF is able to support any number of static checks. For the purposes of our research we have implemented all of the static checks used in our five case studies. These static checks are automatically installed during installation of the VF [23]. The following subsection will illustrate how a new static check can be created for use within the VF.

4.7.1 Creation of a Static Check

Technically a static check is implemented as a managed Dynamic Link Library (DLL). A managed DLL is a DLL that has been implemented using a .NET compatible language such as C# or VB. To aid with the creation of static checks we have created a static check SDK as part of the VF [18]. The SDK includes interfaces and helper classes that can be used when creating a static check. Figure 42 displays the code listing for the *HasMemberOfType* static check that was used in the Container example from Chapter 3. Recall that the *HasMemberOfType* static check is used to determine if the containing type has a field that is declared to be of a given type.

All static checks must implement the *IPlugin* interface. It is this interface that the VF looks for when loading checks at runtime. The *IPlugin* interface defines three read-only accessors. The first accessor, *Name*, returns the name of the static check. It is this name that is used within the Testable Requirements Model (TRM) for invoking the given static check. The second accessor, *Namespace*, denotes the namespace location where the check will reside. Each static check must reside within a namespace for grouping purposes. The *HasMemberOfType* static check is located in the *Core* namespace. Check

namespaces are delimited by a dot (.) and follow the namespace rules found in most object-oriented programming languages.

The final accessor, *Version*, returns the current version of the static check. Both ACL and CIL compilers keep track of, and verify check versions to ensure that a TRM compiled with a given version of a static check is executed against the same version of the check.

```
public class HasMemberOfType : IPlugin
{
   public string Name { get { return "HasMemberOfType"; } }
   public string Namespace { get { return "Core"; } }
   public Version Version { get { return new Version(1, 1, 0, 0); } }
   [StaticEntryPoint]
   [EntryPointDescription("Determines if the containing type has at
      least one field that is of the given type.")]
   public StaticEvaluationResult DoHasMemberOfType(
      Phx.Types.TypeSymbol owner, Phx.Types.TypeSymbol fieldType)
   {
      if (fieldType.Type.TypeSymbol != null)
         fieldType = fieldType.Type.TypeSymbol;
      try
      {
         Field fx = owner.Type.FieldList;
         while (fx != null)
         {
            if (Helper(fx.Type, fieldType))
               return new
                  StaticEvaluationResult(StaticEvaluationResults.Pass);
            fx = fx.Next;
         }
         return new StaticEvaluationResult(StaticEvaluationResults.Fail);
      }
      catch (Exception e)
      {
         return new StaticEvaluationResult(StaticEvaluationResults.Error,
            e.Message);
      }
   }
   private bool Helper(Phx.Types.Type left, Phx.Symbols.TypeSymbol right)
   {
      if (left.IsPointer)
         return Helper(left.AsPointerType.ReferentType, right);
      if (left.ToString().Contains("System.Collections.Generic.List"))
      {
         if(left.AsAggregateType.ArgumentTypeList.Head.Data.
               IsPointerType)
         {
            if(left.AsAggregateType.ArgumentTypeList.Head.Data.
               AsPointerType.ReferentType.TypeSymbol == right)
               return true;
         }
         else
         {
            if(left.AsAggregateType.ArgumentTypeList.Head.Data.TypeSymbol
               == right) return true;
         }
      }
      if (left.ToString() == right.ToString()) return true;
      return false;
   }
}
```

Figure 42 - Implementation of the HasMemberOfType Static Check

The three accessors represent the requirements imposed on each static check by the *IPlugin* interface. In addition, each static check must supply one or more static entry points. A static entry point is a method that provides the behaviour for the static check. Static entry points are invoked by the VF to execute the requested static check. A static entry point is denoted by use of the *StaticEntryPoint* attribute, and may have an optional description as shown in Figure 42. The description is used within the VF to aid authors of TRMs. Each static entry point must return an instance of the *StaticEvaluationResult* class. This instance contains information regarding the outcome of the static check and will be displayed on the CER. The method name of the static entry point is irrelevant as the name of the static check is denoted by the previously mentioned *Name* accessor. The static entry point must accept at least one parameter. The *HasMemberOfType* static entry point accepts two parameters. The first, required, parameter is an instance of the *TypeSymbol* class provided by Phoenix to represent a type within the IUT. The *owner* parameter contains the structural and behavioural information gathered by Phoenix for the IUT type that is bound to the contract where the static check is invoked from. Recall that each static check must be placed within a structure section of a contract, and that each contract is bound to a type within the IUT. It is this bound type that is provided as the required parameter to the static entry point. The second parameter, *fieldType*, represents the single parameter passed to the actual static check within the Testable Requirements Model (TRM). This is the IUT type we are checking to see if there is a field of that type defined within the containing type. The body of our static entry point begins with an *if* statement to ensure that we have the

correct field type. We then iterate through each of the fields defined within the containing IUT type and, using the *Helper* method compare if the field's type matches the type we are looking for. If so, a match has been found and the static entry point returns a passing result. That is, the IUT has passed the static check. If all of the fields have been checked and a match has not been found, the entry point returns a failure result, indicating that the static check has failed. If an exception is generated during execution of the static check an error result will be returned. An error result does not indicate a pass or failure of the static check, but rather instructs the VF that there was an error in the static check itself. If more than one static entry point is provided, the VF determines which entry point to call based on the number and type of parameters provided to the static check within the TRM. That is, multiple static entry points allow for polymorphic static checks.

The *Helper* method is used to determine if two types are equal or not. If the types are equal the method returns true, false otherwise. The body of the method begins by checking if the type represented by the *left* parameter is a reference type, if so the type is dereferenced and execution continues. Note that due to the design of the static check system, it is impossible for the *right* parameter to be a reference type, and thus it is not checked. Next, the method determines if the *left* type represents a list, if so the type of element stored by the list is used to determine if there is a match. Otherwise, the type names are compared, via the *ToString* method, to determine equality. The *Helper* method completes implementation of the *HasMemberOfType* static check.

Once all of the static checks have been executed and their results recorded, the Validation Framework (VF) executes the Implementation Under Test (IUT). By execute we mean that the VF launches the IUT as a new process, and attaches itself to the process. Put another way, the VF acts as a specialized debugger keeping track of instance creation, method invocation, and instance destruction. In the case of a standard Portable Executable (PE) file (.exe), the IUT is launched by executing the PE file. In the case of an ASP.NET web application, the previously mentioned VF web server is launched in a new process and the VF attaches itself to the web server process. The web server in turn is used to execute the web IUT.

In addition to attaching itself to the candidate IUT, the VF will pause execution of the IUT as needed in order to execute the various checks defined within the Testable Requirements Model (TRM). At the same time the IUT is launched, the VF also maintains a separate execution environment for the TRM. That is, the VF acts as a virtual machine that executes the provided Contract Intermediate Language (CIL) instructions. The TRM execution environment contains any contract and scenario instances that have been created as a result of IUT execution. We will now present our execution method.

With the VF acting as a specialized debugger, we are able to capture events as the IUT executes. That is, execution of the IUT acts as the test driver. Rather than generating test cases or execution paths in the VF, execution is defined by the behaviour of the IUT. The generation of other test cases or execution paths lies beyond the scope

of our work as stated in Chapter 1. As the IUT executes, each time a type is instantiated, the VF checks to see if any contracts within the TRM are bound to that IUT type. If so, a new contract instance is created to represent the new IUT type instance. Once the new contract instance is created and connected to the IUT type instance, the *new* responsibility (if specified) is executed. As the IUT continues to execute, any responsibilities that are bound to methods within the corresponding IUT type will be executed as their corresponding IUT methods are invoked. Finally, when the IUT instance is destroyed, the *finalize* responsibility (if specified) will be executed. Details regarding the execution of a bound responsibility are now presented.

4.8.1 Execution of a Bound Responsibility

The execution of a responsibility bound to a method within the IUT begins when the IUT method is invoked. Before the bound IUT method begins to execute, the IUT is suspended to allow execution of the responsibility. If the containing contract has one or more invariants, they are evaluated first. That is, the invariant is checked upon entry to the responsibility. Invariants are usually composed of one or more *check* statements. Each check statement is executed independently, based on the order specified within the TRM. As an example consider the *check* statement shown in Figure 43.

```
Check(context.size == Size());
```

Figure 43 - Check Statement Example

The VF evaluates the check by getting the value stored in the *size* contract variable that is attached to the current contract instance. The current contract instance is determined by the IUT instance on which the bound IUT method is being invoked.

Recall, there is one contract instance for each IUT instance whose type is bound to a contract within the TRM. The right-hand side of the check requires the VF to invoke the *Size* observability method. The invocation is performed by looking in the binding table to find the corresponding IUT method. Once the IUT method is located, the VF will execute that method using the same IUT instance bound to the containing contract. Recall that the VF uses static analysis along with the behavioural model provided by Phoenix at binding time to ensure that the IUT method bound to the observability does not result in the generation of side-effects within the IUT. The result of the invocation (i.e., return value) is then compared to the *size* contract variable to create a true or false value. If a true value occurs the check is said to have passed, otherwise the check fails. The result is written to the Contract Evaluation Report (CER) along with the definition of the check itself, the specific contract instance, and invocation information that lead to execution of the check.

Once all of the invariants have been evaluated, the body of the responsibility is evaluated. Evaluation begins with any preconditions that were specified within the TRM. Each precondition is evaluated using the same technique as invariant checks. The result is written to the CER. After the preconditions have been evaluated, any other statements, such as incrementing a contract variable or executing a dynamic check, are performed by the VF. Details regarding the creation and execution of dynamic checks are presented in the next section.

When the VF encounters an execute statement, denoted by *Execute()* in ACL, execution of the IUT continues allowing the bound method(s) to execute. When the last (or only) bound method has finished execution, the IUT is paused again so any statements, dynamic checks, or post-conditions located after the execute statement can be evaluated. Finally, any invariants are re-evaluated for responsibility exit. When the entirety of the responsibility's body has been executed, the VF notifies each scenario that the given responsibility has executed, and then resumes execution of the IUT. The IUT continues to execute until another method that is bound to a responsibility is invoked and then the process repeats.

The process of pausing and resuming the candidate Implementation Under Test (IUT) happens each time any of the following occurs:

- A new IUT instance is created using a type bound to one or more contracts in the TRM, and any constructors defined on that type have finished executing.
- An IUT instance with a corresponding contract instance is about to be destroyed. That is, the destructor has been invoked but has yet to execute.
- An IUT method that is bound to a responsibility within the TRM has been invoked but has yet to execute.
- An IUT method that is bound to a responsibility within the TRM has finished executing but has yet to return to the caller.

4.8.2 Dynamic Checks

During execution of a responsibility, the responsibility may invoke one or more dynamic checks. Recall that a dynamic check is one that operates on the IUT during execution. That is, execution of a dynamic check queries the execution of the IUT. Each dynamic check has read-only access to any active object instances, memory, and other resources currently in use by the IUT. A dynamic check can then analyze and interpret such runtime information to produce a result. As with static checks, dynamic checks must produce a deterministic pass or fail result. Such a pass or fail result is displayed in the Contract Evaluation Report (CER).

The implementation of dynamic checks is similar to that of static checks. That is, the VF does not provide a fixed set of dynamic checks for use, but features an open architecture so that experts in the domain of dynamic testing can created specialized checks that are executed by the VF during execution of the candidate IUT. We have implemented all of the dynamic checks that are referenced within our five case studies. These dynamic checks are installed as part of the VF [23]. The following subsection will illustrate how a dynamic check can be created for use within the VF.

4.8.2.1 Creation of a Dynamic Check

Like static checks, a dynamic check is implemented as a managed DLL. To help with the creation of dynamic checks we have created the dynamic check SDK as part of the VF [15]. The SDK includes interfaces and helper classes that are used to create a dynamic check and to access the runtime data generated by a candidate IUT and the VF

runtime. Figure 44 displays a code listing for the *UniqueValue* dynamic check. The check is used to ensure that a given value is unique across all active IUT instances.

As with static checks, dynamic checks must also implement the *IPlugin* interface. The three required accessors name the dynamic check *UniqueValue*, place it in the *Core* namespace and assign it a version number of 1.1.0.0. In addition the *UniqueValueDynamicCheck* class defines a static dictionary named *values*. The dictionary will be used to store values within the check so that we can test for uniqueness. Each dynamic check must have at least one dynamic entry point. The VF invokes the dynamic check through a dynamic entry point. Dynamic entry points are denoted by the *DynamicEntryPoint* attribute applied to the corresponding method. An optional description can be specified if desired. Dynamic entry points must have a predefined signature. The method must return an instance of the *DynamicEvaluationResult* class, indicating the result of executing the dynamic check. Recall that static check entry points were required to accept at least one parameter, and that the first parameter was a Phoenix object representing the static structure and behaviour of the IUT type bound to the contract where the static check was being invoked from. Dynamic checks have a similar required parameter, except that the dynamic entry point accepts a parameter of type *IRTContractInstance*. The *IRTContractInstance* interface represents an active contract instance being maintained by the VF runtime. The contract instance object contains methods for accessing the IUT instance bound to the contract instance, as well as information pertaining to memory and other resources used by both the IUT and the contract instance. For information on

the *IRTContractInstance* interface and other types provided as part of the dynamic check SDK, see the SDK documentation [15]. Our dynamic entry point method also accepts a single Boolean parameter named *value* that represents the value we are checking uniqueness for. The body of the dynamic entry point is straightforward. If the given value is stored within the *values* dictionary and the current instance matches the one the dynamic check is storing, the check returns a passing result. That is, the value is unique as it is stored by the same instance. Otherwise, the dynamic check returns a failing result. If the given value is not found in the *values* dictionary it is added, and the dynamic check succeeds.

```
public class UniqueValueDynamicCheck : IPlugin
{
   public static Dictionary<object, IRTContractInstance> values =
      new Dictionary<object, IRTContractInstance>();

   public string Name { get { return "UniqueValue"; } }
   public string Namespace { get { return "Core"; } }
   public Version Version { get { return new Version(1, 1, 0, 0); } }

   [DynamicEntryPoint]
   [EntryPointDescription("Determines if the given value is unique across
      the instance space.")]
   public DynamicEvaluationResult DoUniqueValue(
      IRTContractInstance instance, bool value)
   {
      if (values.ContainsKey(value))
      {
          if (instance == values[value])
             return new
                DynamicEvaluationResult(DynamicEvaluationResults.Pass);
          else
             return new
                DynamicEvaluationResult(DynamicEvaluationResults.Fail);
      }
      values.Add(value, instance);
      return new DynamicEvaluationResult(DynamicEvaluationResults.Pass);
   }
```

```
    [DynamicEntryPoint]
    [EntryPointDescription("Determines if the given value is unique across
        the instance space.")]
    public DynamicEvaluationResult DoUniqueValue(
        IRTContractInstance instance, char value)
    {
       if (values.ContainsKey(value))
       {
           if (instance == values[value])
              return new
                 DynamicEvaluationResult(DynamicEvaluationResults.Pass);
           else
              return new
                 DynamicEvaluationResult(DynamicEvaluationResults.Fail);
       }
       values.Add(value, instance);
       return new DynamicEvaluationResult(DynamicEvaluationResults.Pass);
    }
     ...
}
```

Figure 44 - Implementation of the UniqueValue Dynamic Check

A second dynamic entry point follows to illustrate the notion of a polymorphic check. The first dynamic entry point will be invoked if the value to check is a Boolean value, the second dynamic entry point will be invoked if the value to check is a character value. Additional entry points would be specified to handle additional types (not shown). The second dynamic entry point concludes the *UniqueValue* dynamic check.

4.8.3 Scenario Execution

Returning to the execution of a candidate Implementation Under Test (IUT) against a Testable Requirements Model (TRM), as each responsibility completes execution all scenarios located within the same contract are notified of the responsibility execution. Each scenario maintains a set of scenario instances. When a scenario is notified of a responsibility execution, it forwards the notification to each of its scenario instances. Each scenario instance maintains the responsibility it is currently waiting for (based on its execution grammar). If a scenario instance discovers a match, it moves on to the

next element specified in the execution grammar. If not, the scenario instance records observation of the unexpected responsibility and continues waiting. Regardless of the outcome, execution of the responsibility is recorded on the Contract Evaluation Report (CER) along with any scenarios that were advanced as a result. If none of the scenario instances were expecting the given responsibility to be executed, the scenario's triggering event is examined. If a match occurs, a new scenario instance is created accordingly. Scenario instances are active until their execution grammar is completed and the termination event occurs. At that point, the scenario instance is terminated and the results of scenario execution, including any checks, responsibilities, or observable events that occurred during the lifetime of the scenario instance are written to the CER. There is no limit to the number of scenario instances that may be active during execution of a candidate IUT.

Our approach to scenario execution does not involve the creation of a state machine representing the IUT as is done in many of the approaches discussed in Chapter 2. Rather, we employ a pattern matching technique, which takes the current position within a scenario instance and then based upon the execution grammar, determines a set of expected responsibilities. As responsibilities execute within the TRM the scenario instance advances through the execution grammar. In the case where multiple paths through an execution grammar exist, the VF runtime keeps track of all paths and will select one based on responsibility execution. If a specific path cannot be determined, the VF runtime will follow all matching paths in parallel until a single path can be determined based on the order of responsibility execution. During compilation, the

Another Contract Language (ACL) compiler analyzes each execution grammar and will generate an error if a non-deterministic path is discovered. That is, the ACL compiler prevents intractable paths from occurring within the VF runtime.

Once a scenario instance has completed execution, any active relation instances are notified of the successful scenario execution. Recall that the body of a relation is specified using an execution grammar of scenarios. As such, relation execution operates in the same fashion as scenario execution; except scenarios are notified of responsibility executions and relations are notified of scenario executions. Execution of responsibilities, scenarios, and relations continues until the IUT stops executing in the case of non web IUTs, or the *Stop Server* button is clicked on the VF web server in the case of web IUTs.

Once the IUT finishes execution, the VF notifies any scenario or relation instances that have yet to terminate that the IUT has finished, and thus the scenario or relation instance has failed. Next, the result of executing each scenario and relation instance is written to the CER. Such information includes the execution grammar specified in the TRM, the actual execution trace, and any unexpected responsibilities or observable events. Once all execution related information has been written to the CER one final task remains: analysis of any gathered metric information.

4.9 Metrics

Evaluation of the reports section found in each contract is the final step in executing a Testable Requirements Model (TRM) against a candidate IUT. Evaluation consists of

the VF invoking metric evaluators to analyze and report on the metric information gathered during IUT execution. The gathering of metric data is performed by the VF runtime in conjunction with any metrics gathered within the TRM itself. As the analysis of metric data is inherently domain-specific and requires specialized knowledge of non-functional requirements such as performance, security and usability. Our VF supports an open model for the specification of metric evaluators. That is, we do not claim to be experts in the analysis of such metric data, but rather provide a metric evaluator SDK for the creation of specialized metric evaluators [16]. The following subsection will outline how a metric evaluator can be created and used within the VF. We have implemented all of the metric evaluators used within our five case studies and they are automatically installed as part of the VF [23].

4.9.1 Creation of a Metric Evaluator

Metric evaluators are created in a similar fashion as static and dynamic checks. They reside within a managed DLL and are created using the metric evaluator SDK that provides interfaces and helper classes that help with the evaluation of metric data. Figure 45 displays the code listing for the *AvgMetric* metric evaluator used in the Container example presented in Chapter 3. The *AvgMetric* metric evaluator is used to return the average value from a list of metric values.

```
public class AvgMetric : IPlugin
{
    public string Name { get { return "AvgMetric"; } }
    public string Namespace { get { return "Core"; } }
    public Version Version { get { return new Version(1, 1, 0, 0); } }

    [MetricEntryPoint]
    [EntryPointDescription("Provides the average value for a series
        of metrics.")]
    public MetricEvaluationResult DoAvgMetric(out double result,
        List<int> values)
    {
        result = 0.0;
        if (values != null)
        {
            int total = 0;
            foreach (int i in values)
                total += i;
            result = ((double)total / (double)values.Count);
        }
        return new MetricEvaluationResult(MetricEvaluationResults.Pass);
    }
}
```

Figure 45 - Implementation of the AvgMetric Metric Evaluator

The *AvgMetric* class implements the *IPlugin* interface, and begins with the implementation of the required accessor methods. The accessor methods indicate that the metric evaluator is named *AvgMetric,* is located in the *Core* namespace, and has a version number of 1.1.0.0. Each metric evaluator must define at least one metric entry point. As with static and dynamic entry points, a metric entry point is the method that will be invoked by the VF to execute the metric evaluator. Multiple metric entry points allow for polymorphic metric evaluators. A metric entry point is denoted by use of the *MetricEntryPoint* attribute. Next, an optional description is specified to provide more information about the analysis performed by the evaluator. Each metric entry point must have a specific method signature. The entry point method must return an instance of the *MetricEvaluationResult* class indicating the result of executing the given metric evaluator. Next, the metric entry point must accept at least one parameter, the

required parameter is an *out* parameter and is used to represent the value returned by the execution of the metric evaluator. That is, the metric evaluation result returns a pass, fail, or error condition that is reported on the CER, where the *out* parameter returns the actual value calculated by the evaluator. Any additional parameters supplied to the metric entry point represent the parameters specified in the TRM. In the case of the *AvgMetric* metric evaluator, a single integer list is provided containing the individual metric values to average. The body of the metric entry point is straightforward and calculates the average value contained within the given integer list. As the *AvgMetric* metric evaluator simply calculates an average value execution of the evaluator always results in a successful (pass) result.

The results generated by the metric evaluator plug-ins are formatted and written to the CER using the previously discussed *Report* and *ReportAll* statements. Evaluation of each contract's reports section completes the process of executing a candidate IUT against a TRM. We will now present the report generated as a result of such execution.

4.10 Contract Evaluation Report

The Contract Evaluation Report (CER) displays results covering all aspects of executing a candidate Implementation Under Test (TRM) against the Testable Requirements Model (TRM). The results of each contract, contract instance, observability, invariant, responsibility, scenario, metric analysis, interaction, relation and relation instance are shown. Each distinct candidate IUT executed by our VF will have a separate CER. Multiple CERs can be compared to find differences between candidate IUTs.

The CER is saved as a single XML file containing all of the results generated during execution of the candidate IUT against the TRM. While the XML file could be interpreted directly, or through a third-party tool, the VF contains a built-in report viewer. The report viewer provides a graphical way to visualize all of the elements contained within the CER. The following subsections will present some of the views and information displayed by the report viewer. For a complete list see the Validation Framework Installation Usage Guide [23].

4.10.1 The Report Tree

Like the binding tool, the report viewer is divided into a tree and main view. The report tree contains information pertaining to each contract and interaction that was executed. These elements include: contracts, structure sections, contract instances, observabilities, responsibilities, invariants, metrics, scenarios, scenario instances, reports sections, interactions, relations and relation instances. A portion of the report tree from the Container example is provided in Figure 46.

Each element is grouped according to the contract or interaction where the element was defined. Each element is then assigned a colour to denote its execution result. Colours are assigned as follows:

- Black - A reports section. The root binding node is also coloured black.
- Green - The element executed correctly. That is, no checks, beliefs, pre- and post-conditions, static checks, dynamic checks, or metric evaluators failed.

- Red - The element or one of its child elements did not execute correctly or contained a check that was violated by the candidate IUT.

In addition to the colours, different icons are used to indicate the result. That is, the *check mark* icon shown in Figure 46 indicates that the corresponding element has executed successfully. To view additional details for any of the nodes displayed in the report tree, select the node. When a node is selected the main view will be updated to reflect the selection. Each node displayed within the report tree can be selected.

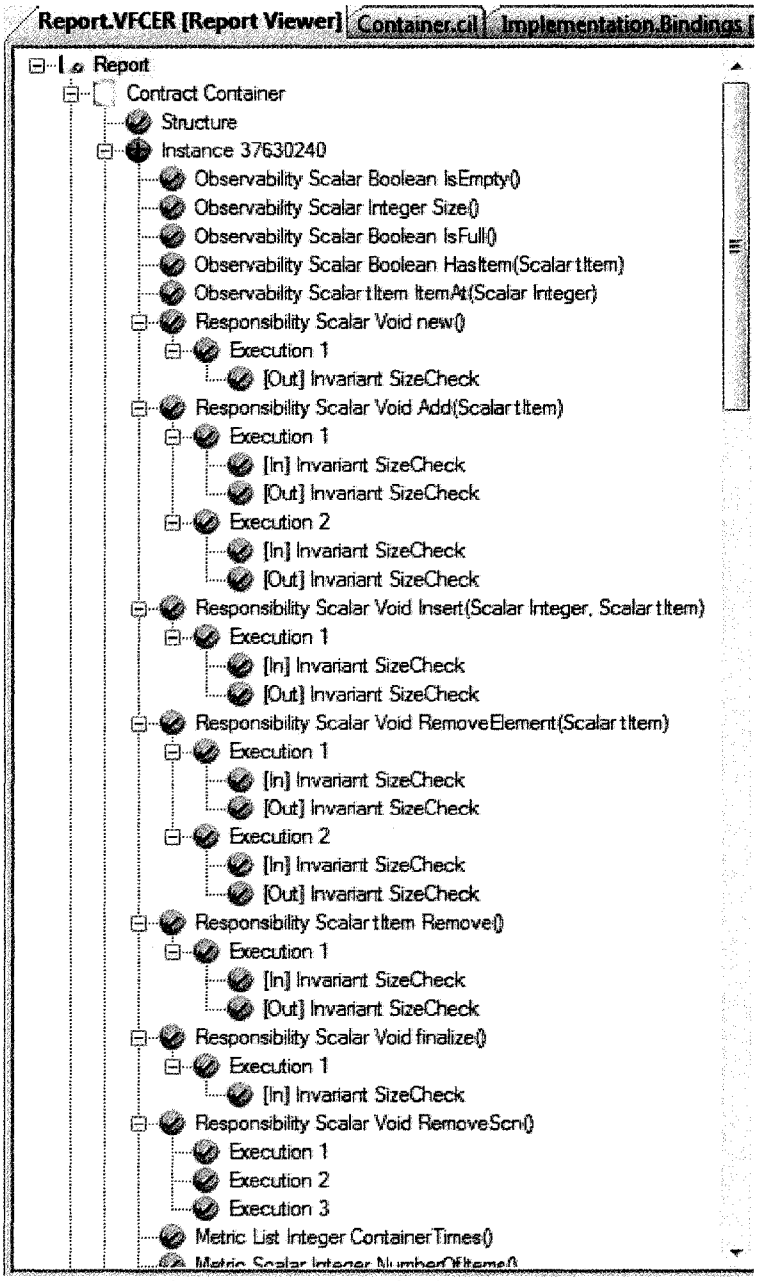

Figure 46 - Contract Evaluation Report Tree

4.10.2 The Main View

The main report view is located to the right of the report tree. The report view displays detailed information for the selected node. There are 21 possible report views that can be displayed. The following subsections will present some of the common report views. A complete list and corresponding discussion can be found in [22].

4.10.2.1 Report View

When a CER is first opened the report view is displayed. The report view provides an overview of the contents found within the CER. Figure 47 displays the report view generated after executing the Container example from Chapter 3.

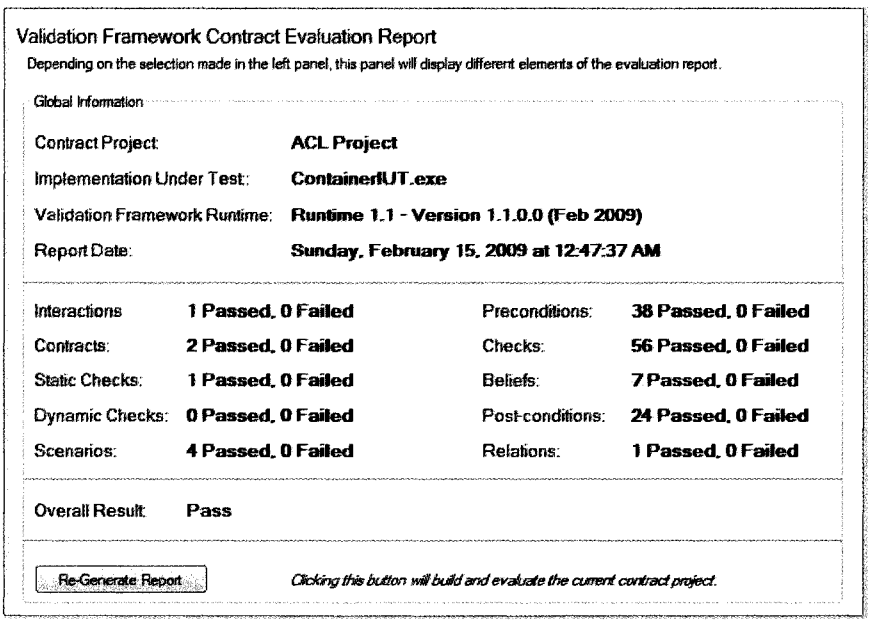

Figure 47 - Contract Evaluation Report: Main View

The report view begins by displaying the type of project that has been executed, the file name of the candidate IUT that the Testable Requirements Model (TRM) was applied to, the Validation Framework (VF) runtime that was used, and the date the report was generated. The next section of the report view provides a series of totals, indicating the number of each type of element that can be found within the CER. The report view also contains the overall result of execution. The result will be a *Pass* if each element found within the TRM has executed correctly. If a single element generates a fail result, the overall result will be *Fail*. Finally, the report view contains a single button entitled *Re-Generate Report*. As the name suggests, when the button is clicked it will re-

compile, and then execute the current TRM against the candidate IUT producing a new CER.

4.10.2.2 Contract View

The contract view is displayed when a contract node is selected in the tree. The contract view displays summary information regarding the outcome of executing the selected contract against the IUT. The contract view is shown in Figure 48. The view begins with the name of the contract that the node represents followed by the IUT type to which the contract was bound to. In addition the number of contract instances that were created as a result of executing the IUT against the TRM is displayed. The next block of information contains a summary of all the checks, metrics, scenarios, preconditions, beliefs, and post-conditions that were executed by all instances of the selected contract. Finally, the overall result of executing all contract instances is displayed.

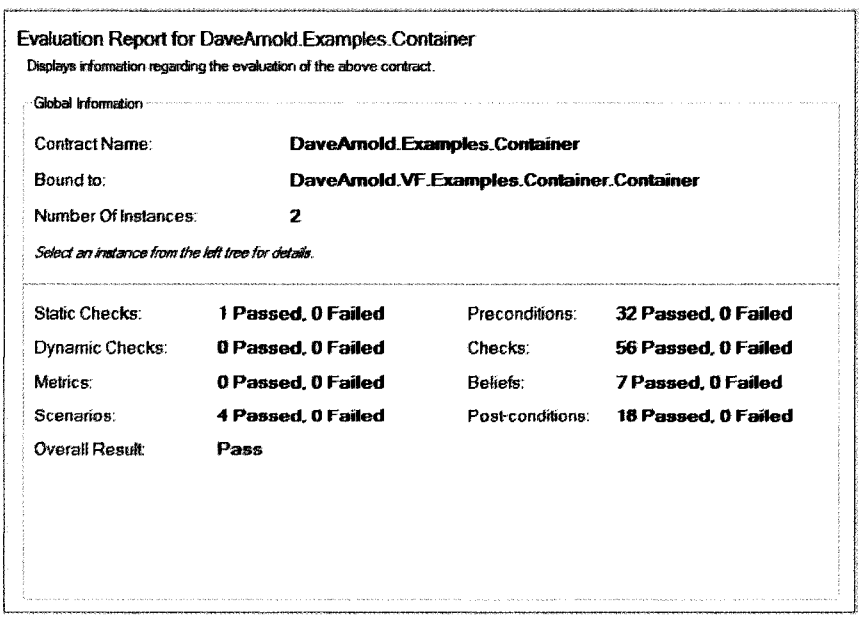

Figure 48 - Contract Evaluation Report: Contract View

4.10.2.3 Structure View

The structure view is displayed when a contract's structure node is selected in the tree view. The structure view displays detailed information regarding the execution of the contract's structure section as shown in Figure 49.

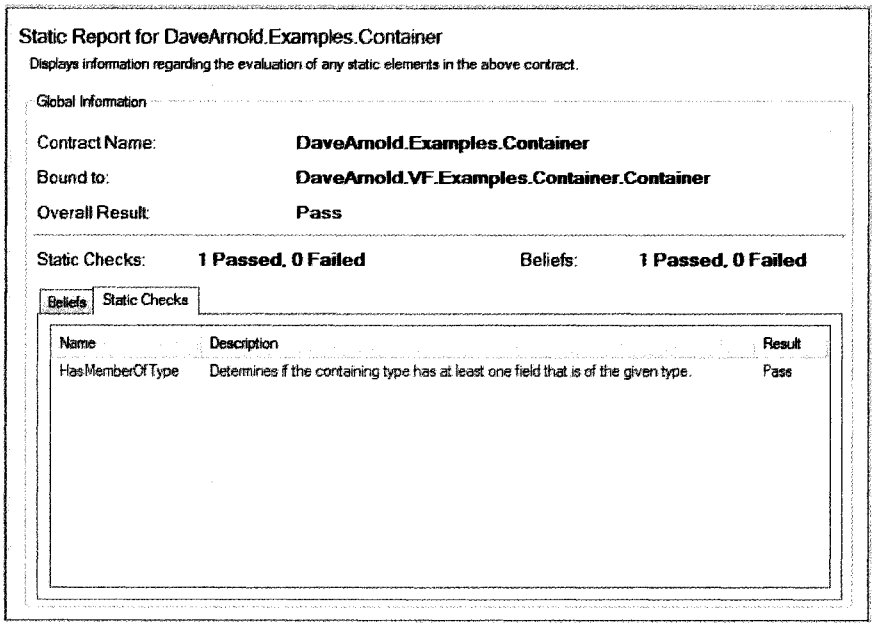

Figure 49 - Contract Evaluation Report: Structure View

The structure view begins with the name of the corresponding contract, and bound IUT type. The overall result is the result of executing all static checks and beliefs within the structure section. Next, the total numbers of beliefs and static checks are displayed, followed by a tab view displaying the details. The single static check, *HasMemberOfType*, is shown in Figure 49. For each static check, the name of the check, a short description of the check, and the result of executing the check is shown. If the check passes, the line will be displayed in green, otherwise red will be used. In the case of beliefs, the belief name, description, and execution result are displayed.

4.10.2.4 Bound Responsibility View

The bound responsibility view is displayed when a responsibility that is bound to one or more IUT methods is selected in the report tree. The view provides an execution summary of the selected responsibility as shown in Figure 50. The bound responsibility view begins with the name of the contract, and the first (or only) method that is bound to the responsibility. The overall result, displays the result of all executions of the selected responsibility. The number of executions follows. The next section provides the numbers of preconditions, checks, beliefs, post-conditions, and dynamic checks executed by all executions of the responsibility.

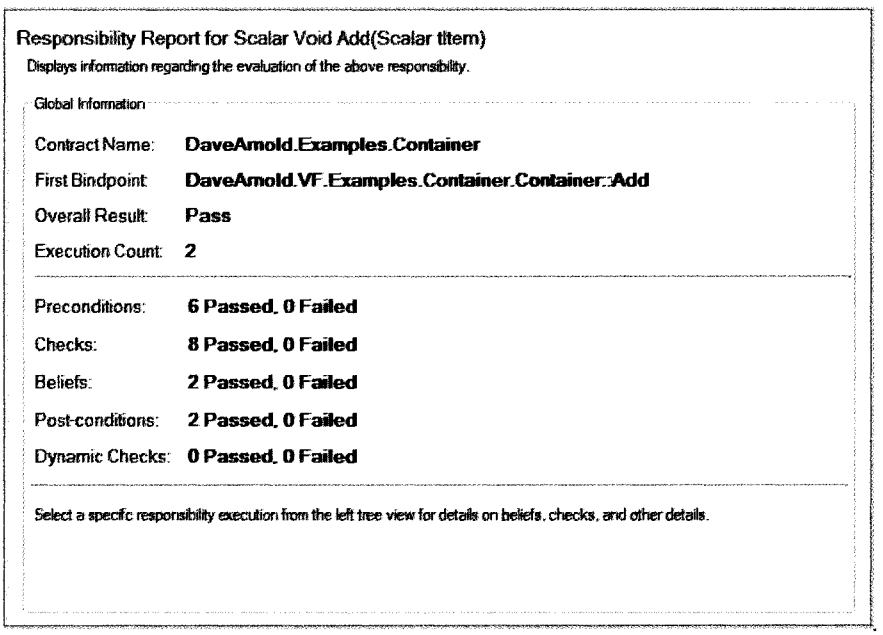

Figure 50 - Contract Evaluation Report: Bound Responsibility View

4.10.2.5 Bound Responsibility Execution View

The bound responsibility execution view is displayed when a specific responsibility execution is selected in the report tree. The responsibility execution view displays detailed information regarding a specific responsibility execution. An execution of the

Add responsibility is provided in Figure 51. The bound responsibility execution view begins with the containing contract, followed by the name of the first (or only) IUT method that the responsibility is bound to. The overall result of executing the responsibility is displayed along with the assigned execution number. Responsibility execution numbers begin at one and increase with each execution of the responsibility. On the right side, a list of total preconditions, checks, beliefs, post-conditions, and dynamic checks is displayed. Finally, the tab view shows details pertaining to the individual elements. Displayed in Figure 51 are the three preconditions that were defined as part of the *Add* responsibility from Chapter 3.

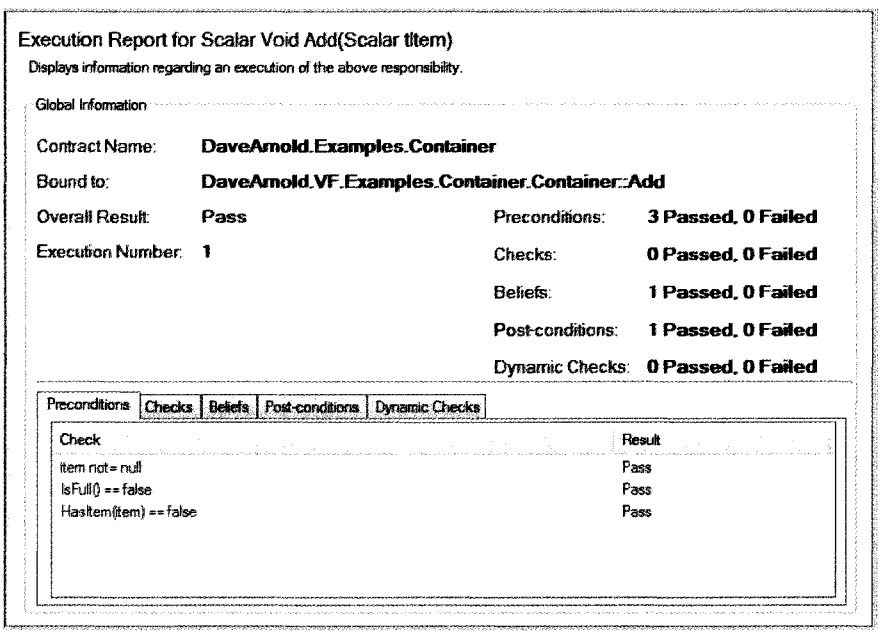

Figure 51 - Contract Evaluation Report: Bound Responsibility Execution View

4.10.2.6 Invariant View

The invariant view is displayed when an invariant execution is selected in the report tree. Invariant executions are located as children of each responsibility. That is, an

invariant defined within a contract is replicated for each responsibility.[9] The invariant view contains information pertaining to its execution as shown in Figure 52. The invariant view displays the containing contract name and the overall result of executing the invariant. The overall result is based on the execution of the beliefs, dynamic checks, and check statements found within the body of the invariant. Next, the execution type is provided. Invariants can have one of two execution types: *Entry* and *Exit*. The *Entry* type indicates that the selected invariant execution occurred upon entry to the containing responsibility. Likewise, the *Exit* type indicates that the execution occurred after the containing responsibility did. The view also provides the number of beliefs, dynamic checks, and regular checks that were executed; details of which can be found in the tab view.

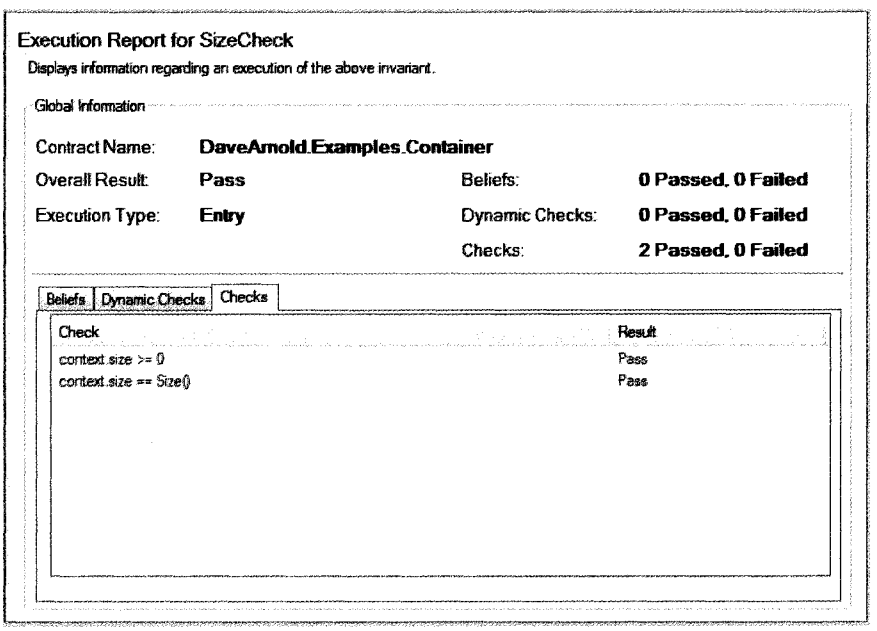

Figure 52 - Contract Evaluation Report: Invariant View

[9] The invariant is not actually replicated, but rather is displayed in the CER as such. Only one set of CIL instructions is generated by the ACL compiler.

4.10.2.7 Scenario View

The scenario view shown in Figure 53 is displayed when a scenario is selected in the report tree. The view begins with the name of the containing contract, the overall result of all scenario instances, and the number of instances that were created during execution of the candidate IUT. Total numbers of preconditions, checks, beliefs, post-conditions, and dynamic checks are also displayed. At the bottom of the view, the scenario grammar, in ACL, as specified in the TRM is displayed. The scenario displayed in Figure 53 is from the Container example.

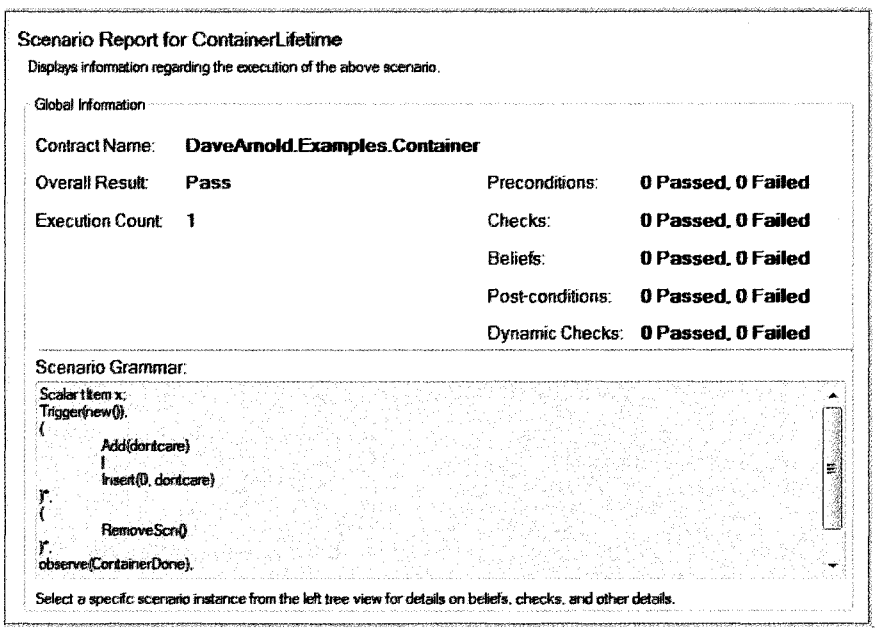

Figure 53 - Contract Evaluation Report: Scenario View

4.10.2.8 Scenario Instance View

The scenario instance view is shown when a specific scenario instance is selected in the report tree. The scenario instance view displays the actual execution trace generated by the candidate IUT during execution. Figure 54 illustrates an example of the scenario instance view.

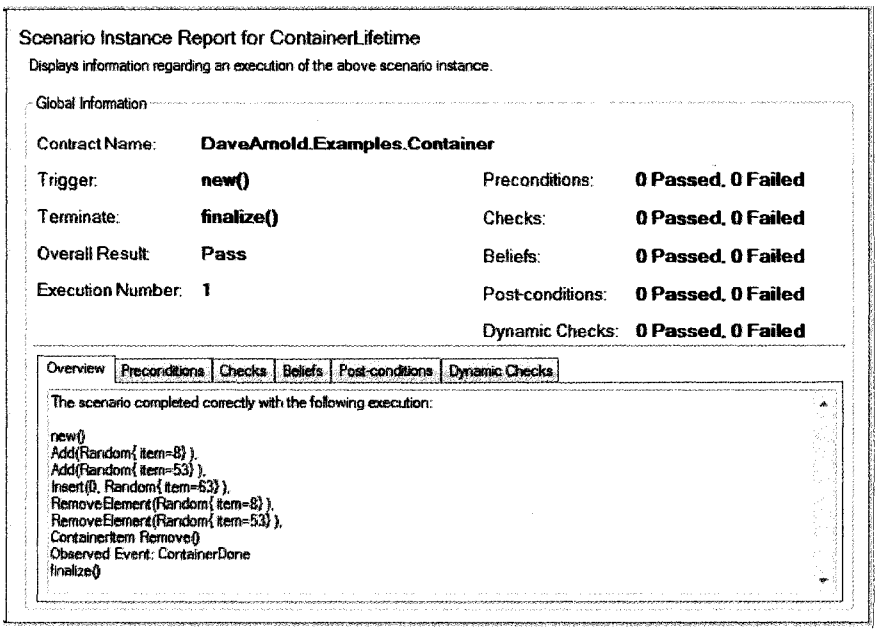

Figure 54 - Contract Evaluation Report: Scenario Instance View

The scenario instance view displays the name of the containing contract. Next, the actual triggering and terminating events are displayed. Note that if the scenario grammar is specified using multiple triggering and/or terminating events, only the event that is actually responsible for triggering and/or terminating the selected scenario instance will be shown. The overall result is displayed along with the scenario instance execution number. Each scenario instance is assigned a unique execution number, based on the ordering of triggering events. The number of preconditions, checks, beliefs, post-conditions, and dynamic checks (both passed and failed) is displayed. Finally, the tab view contains the actual execution trace for the selected scenario instance, along with details for any checks and beliefs defined as part of the scenario body.

4.10.2.9 Reports View

The reports view is displayed when a *Reports* node is selected in the report tree. The report view displays the results generated by the reports section of a contract. An example reports view is shown in Figure 55. The reports view displays the resultant string after executing any *Report* or *ReportAll* statements found within a reports section of a contract.

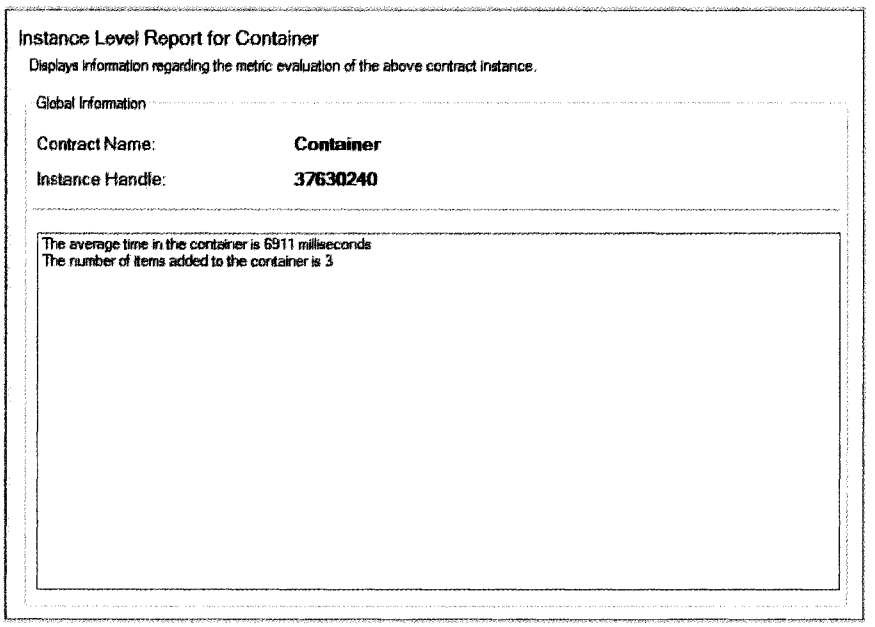

Figure 55 - Contract Evaluation Report: Reports View

The report tree and main views provide a graphical way to view and interpret the CER generated by our VF during execution of a candidate IUT against a TRM. The report view uses a tree view and 21 specialized views to provide the user with a complete picture of how their TRM executed against an IUT. The generation and display of the CER concludes the execution process outlined in the VF process overview shown in Figure 27.

4.11 Summary

The Validation Framework (VF) serves as a complete toolkit for the specification of a Testable Requirements Model (TRM) through the use of our text-based general-purpose contract specification language known as Another Contract Language (ACL). The VF then accepts one or more candidate IUTs and a set of bindings that connect the implementation independent TRM to a concrete IUT. The bindings can be specified automatically or manually through the use of a specialized binding tool included within the VF. Once a set of bindings has been specified the VF compiles and executes the TRM against the candidate IUT. Execution begins with the evaluation of static checks followed by launching the candidate IUT and monitoring its execution. As execution of the IUT progresses the VF pauses and resumes the IUT in order to execute dynamic checks, and evaluate preconditions, invariants, and post-conditions. Furthermore as the IUT executes, a pattern matching approach is used to create and maintain scenario instances to ensure that the IUT conforms to the execution grammars contained within the TRM. Once the IUT has finished executing, metrics gathered during execution are analyzed using domain-specific metric evaluators. Complete results of the execution process are written to the Contract Evaluation Report (CER). The CER is displayed in a graphical format using the report viewer that has been built into the VF. We will now discuss how we validated the VF before presenting some concluding remarks.

4.12 More on Validation

In addition to the validation provided by our five case studies discussed in Chapter 3, additional lower level validation was performed on the ACL and CIL compilers as well as

the VF runtime and scenario engine. For the ACL compiler, tests were performed for each of the 713 errors and 35 warnings that the compiler is able to generate. Each test resulted in a block of ACL syntax that generated the error or warning being tested. Once an error or warning was generated by the ACL compiler, the TRM was modified so that the error or warning was no longer generated. The same process was repeated using the CIL compiler for the generation of its 764 errors and 4 warnings. That is, a total of 1,516 compiler tests were performed.

The code generation portion of the ACL compiler was validated by feeding the resultant CIL directly into the CIL compiler to ensure correct code generation occurred. Once the CIL compiler was able to compile CIL instructions, the execution of each instruction by the VF runtime was validated by executing the instruction and observing the result written to the CER. Validation of the binding tool, scenario engine, and report viewer was accomplished by running our five case studies and variations thereof within the VF and observing the results.

5 Conclusion and Future Work

The following will summarize our research and contributions to the state-of-the-art. We will also discuss limitations of our Validation Framework (VF) and suggest how they can be overcome. Finally, areas for future work will be addressed. We begin with a recapitulation of our work.

5.1 Summary of our Approach

Our work closes the *gap* between abstract models and concrete implementations for the purpose of automated validation. We have implemented a VF that is able to close this *gap*. The VF takes, as input, a Testable Requirements Model (TRM), a candidate Implementation Under Test (IUT), and a set of bindings that map artifacts defined within the TRM, specified as a set of contracts and interactions, to corresponding types, and methods within the IUT. The VF then executes the candidate IUT against the TRM to produce a Contract Evaluation Report (CER) containing the results of validation. The following subsections will briefly review our VF.

5.1.1 Framework Input

The VF operates on three input elements. The first element is the TRM. Such a model captures the contract of the system being validated. The VF includes a language Software Development Kit (SDK) [13] for the creation of languages that can be used for the specification of a TRM. The SDK helps promote the openness of our framework and allows for creation of a TRM that is syntactically similar to a specific problem domain. For the purposes of our research we have defined and implemented support for a general purpose high-level TRM specification language known as Another Contract

Language (ACL) [8]. The ACL is closely tied to requirements, and defines constructs for the representation of goals, beliefs, scenarios, relations, and lower-level constructs such as pre and post-conditions. Additional functionality can be added to the ACL through the use of static check, dynamic check, or metric evaluator plug-ins.

The second input element is the candidate IUT that will be executed against the TRM for validation. The VF supports IUTs that are implemented using a .NET [141] language, C++, or target the Active Server Page (ASP) .NET web platform. Our VF uses the Phoenix Research Development Kit (RDK) [144] to load and analyze binary IUT representations. As such, we do not require the availability of source code for validation. Using a process known as binding our VF is able to validate multiple IUTs against a single TRM.

Such bindings represent the third and final input element to the VF. Before a TRM can be executed the types, observabilities, and responsibilities exposed by the contract must be bound to types and methods within the candidate IUT. Our binding tool uses information provided by Phoenix to automatically infer the majority of bindings required by a given TRM. In addition to the supporting of multiple candidate IUTs against a single TRM, bindings allow for the abstraction of implementation details from the TRM.

5.1.2 Plug-ins

Plug-ins build on the openness provided by the language SDK allowing for the inclusion of user specified static checks, dynamic checks, and metric evaluators. Plug-ins can be invoked by any TRM specification language to provide additional framework

functionality. As shown in Chapter 4, the VF includes a plug-in SDK for the creation of static checks [18], dynamic checks [15], and metric evaluators [16].

Static checks perform a check on the candidate IUT that can be accomplished without execution of the IUT. Static checks are invoked by code placed in the *structure* section of a contract. Each static check has a return type and accepts any number of parameters.

Dynamic checks perform a check on the candidate IUT during execution. That is, a dynamic check can only be evaluated while the IUT is being executed. Dynamic checks are invoked by placing code in the *observability*, *responsibility*, or *scenario* sections of a contract. As with static checks, dynamic checks can be viewed as an operation with a return type and parameter set.

Metric evaluators are used to analyze and report on metrics gathered while the IUT was executing. Metric gathering is performed by the VF runtime and by the TRM itself. Once the candidate IUT has finished execution and metric gathering is complete, the metric evaluators are invoked. Metric evaluators can only be invoked by placing code in the *reports* section of a contract.

Through the use of domain-specific contract specification languages, static checks, dynamic checks, and metric evaluators, our VF supports a high degree of openness. Such openness allows for not only domain-specific constructs, but also allows for static testing, dynamic testing, and metric capture and analysis researchers to contribute

without having to create a specialized method or process for the integration of their work.

5.1.3 Compilation

Once the Testable Requirements Model (TRM) has been specified and bound to a candidate IUT, the TRM is compiled into an intermediate language known as the Contract Intermediate Language (CIL) [14]. The CIL is a low-level stack-based language that is accepted by the VF runtime. The purpose of the CIL is to allow for a layer of abstraction between the high-level TRM specification languages and the VF runtime. That is, any future TRM specification languages simply have to target the CIL instruction set [14], thus removing the need for direct modification of the VF runtime.

5.1.4 Execution

The execution of a TRM against a candidate IUT begins with an analysis of the structural composition of the IUT, and the execution of any static checks. Following execution of the static checks, the IUT is launched by the VF. The VF acts as a specialized debugger and attaches itself to the candidate IUT. The IUT then executes against the VF as it tracks and records the execution of scenarios, dynamic checks, and gathers metric data. Following execution of the IUT, metric evaluators are used to analyze the metric data gathered during execution. The result of executing a candidate IUT against a TRM is the generation of a Contract Evaluation Report (CER). The CER includes the outcome of each static check, dynamic check, scenario, and metric evaluator in a graphical format. The CER also indicates the overall result of validating the candidate IUT against the TRM.

5.1.5 Contributions

Our TRM and supporting VF contribute in the areas of requirements engineering and validation by:

- Proposing a new set of requirements for a requirements model that supports operational validation. This set is the first to include all of the following:
  - Capture of functional and non-functional requirements.
  - Testability of the requirements model.
  - Executability of checks generated from this testable model.
  - Semantics rooted in the notions of responsibilities and scenarios.
  - Abstraction of the testable model over several possible implementations.
  - Openness to support specific static checks, dynamic checks, and metric evaluators.
- Defining a TRM that satisfies these requirements (the ACL).
- Providing an open VF supporting the specification and execution of the TRM.

The above contributions along with our notion of bindings allow for the automated connection between an abstract implementation independent TRM and a concrete IUT. Thus, we have provided the ability to close the *gap* between abstract models and concrete IUTs. Let us elaborate, bindings remove the need for *glue code* between model and implementation, while still preserving the abstractness of the requirements model. Keeping the TRM grounded in the notion of responsibilities and scenarios allows for its review by stakeholders and its application to multiple IUTs, potentially using

different implementation technology. We now discuss some limitations of our VF and hint at possible solutions before presenting areas for future work.

5.2 Discussion of Technical Limitations

The following section will discuss some technical limitations of our approach, including distribution, performance, and scalability concerns.

5.2.1 Inputs

As previously discussed, the Validation Framework (VF) accepts three types of candidate IUTs: .NET binaries (.exe Portable Executable (PE) files), C++ binaries (.exe PE files), and ASP.NET web applications (.dll files). IUTs created using other implementation technology are not supported. Such a limitation has occurred as a result of our use of Phoenix [144]. As previously stated, we use Phoenix for opening and interacting with binary IUTs. Phoenix does not support other types of implementation technologies. If the use of additional implementation technologies is required, the support provided by Phoenix would have to be added to our implementation of the VF. Such support includes the opening and construction of a model from a binary implementation, such as a Java .class file [175]. In a previous incarnation of our VF, we implemented such support for .NET binaries [13], before we chose to use Phoenix. Such implementation proves that support for additional IUT types is possible.

5.2.2 Distribution

As the VF acts as a specialized debugger, attaching itself either directly to the IUT or indirectly via the specialized web server (in the case of ASP.NET IUTs), the VF must:

1. Execute on the same machine as the IUT (or web server) and;
2. Can only execute one IUT at a time.

The first limitation indicates that if the IUT executes on an embedded system or a platform that cannot support execution of the VF, the IUT cannot be validated. The rationale here is that our VF requires access to the memory space and other resources used by the IUT. Such a limitation could be alleviated by modifying the VF runtime to support the notion of *remote debugging* used by popular development tools such as Eclipse [180] and Visual Studio [137].

The second limitation indicates that the VF is unable to execute IUTs that are distributed across several machines. The VF is unable to coordinate between the execution of multiple IUT processes. See Section 5.3.3 for how such distribution could be supported by our VF.

5.2.3 Scalability

In many of the state-based approaches to Model-Based Testing (MBT), including Grieskamp's Spec Explorer tool [76], presented in Chapter 2, the size of either the model or IUT that can be validated is restricted due to the aforementioned state-space explosion problem [77]. Recall, that even simple systems generate a large amount of test cases, and thus states. Most of the MBT approaches use some sort of heuristic to prune the state space so that the approach can be used.

As presented in Chapter 4, the VF uses a matching technique to connect the execution of responsibilities to scenarios. Such a technique does not result in the

generation of a model representing the IUT's behaviour. Thus, our VF does not suffer from the state-space explosion problem or any other restriction resulting from the exploration of possible behaviours exhibited by an implementation. Instead, object instances are created within the VF runtime to represent scenario and relation instances. That is, our VF is able to operate on IUTs of any size or complexity. The only limitation lies in the amount of memory the VF is able to use for representation of the active scenario and relation instances. Such a limitation occurs within any implementation that is allocating and de-allocating object instances.

5.2.4 Performance

As the VF acts as a specialized debugger, with respect to the IUT, execution performance of the IUT is reduced. The reduction in performance results from the pausing and resuming of the IUT during execution for the purpose of executing checks, and updating data stored within the VF runtime. This is similar to when an application is compiled and executed in debug mode within Eclipse [180] or Visual Studio [137]. Observation of our five case studies has shown that the IUT executes approximately five times slower when executed within the VF. The additional execution time is proportional to the complexity and number of checks contained within the TRM. As the VF is designed for use during validation of an IUT and not by the end user of the software system being developed, we feel that such a reduction in performance is acceptable and expected. Additional work in optimizing the VF runtime could improve IUT execution performance. It should be noted that we have already achieved a

performance increase of over 50% compared to the framework presented during the thesis proposal. We will now present some additional areas for future work.

5.3 Future Work

In addition to addressing the limitations presented in the previous section, there are other areas of future work. These areas include work currently being done by other members of our research group, extension of the VF through openness, internal enhancement, graphical Testable Requirements Model (TRM) representations, and interpretation of the Contract Evaluation Report (CER). The following subsections will briefly examine each area.

5.3.1 The Bigger Picture

Our VF serves as the cornerstone for several other areas of research within the software engineering research group. Ongoing research projects are working towards generating a set of ACL contracts and interactions for systems containing a high-degree of variability, and the creation of checks for the discovery and analysis of design patterns and their use [72]. The following subsections will provide a brief look at how our VF is being used in the context of other research projects.

5.3.1.1 Variability in Containers

Bashardoust-Tajali is focusing on sequential containers [i.e., 40, 124, 172] and the creation of corresponding generative contracts [53]. Her work has revealed that though different container libraries share essentially the same *rules* and algorithms, they can vary vastly in terms of implementation, parameterization and traversal strategies offered to their user. Furthermore, she is capturing the *rules* and algorithms for such

containers in the form of a generative TRM. Such a generative TRM would then be specialized according to the selected container to yield a concrete TRM that would be executed against our implemented system for validation.

Bashardoust-Tajali is using the ACL as the target for her generative TRM, and our VF for the execution of a specialized TRM against a concrete implementation of such a specialized container.

5.3.1.2 Discovery and Use of Design Patterns

Radonjic is looking into the design of static and dynamic checks that will focus around design patterns [72]. His research centers around two themes. First, discovering where a specific design pattern should be used within an implementation. Here static checks, in the form of plug-ins, would be created to analyze the structural makeup of a candidate IUT to determine if it would be advantageous to use a particular design pattern.

Secondly, Radonjic is devising *rules* to determine if a given design pattern is being used according to its intended use. The *rules* would then be encoded into both static and dynamic checks that could be executed against a candidate IUT to ensure the correct use of a design pattern.

5.3.1.3 Discussion

Our VF provides the necessary infrastructure for the specification and execution of a TRM. Several research initiatives are already taking advantage of the functionality that is provided by our research. We hope that additional research makes use of our VF, to

both further requirements engineering research, and to validate the usefulness of our framework.

5.3.2 Extension through Openness

The VF was designed to be open so that additional Testable Requirements Model (TRM) specification languages and features could be included. Future work exists in the development of additional high-level TRM languages targeting specific problem domains. Each of these languages would generate CIL to be executed by our VF runtime. Future work also exists in the creation of static checks, dynamic checks, and metric evaluators for specific tests. Our VF implementation already contains a set of plug-ins that either provide common functionality or serve as a proof of concept in a given area. Such future work will enrich the functionality of the VF by providing a larger set of specialized checks for authors of TRMs.

5.3.3 Internal Framework Enhancement

Additional work within the VF exists in several areas. One such area is the addition of language constructs and functionality to handle validation of distributed applications. Such support would include the addition of inter-scenario operators supporting distribution and the coordination of multiple instances of the same IUT within the VF runtime potentially running on different machines. Currently only support for a single IUT is provided by the VF as discussed in Section 5.2.2.

As stated in Chapter 1, the generation, selection, and instantiation of execution paths within the IUT has been intentionally omitted from the VF. Future work would thus include the addition of support within the VF to measure scenario coverage and the

suggestion of execution paths, and ultimately scenarios that should be represented within the TRM. That is, instead of having the author of a TRM specify the scenarios, they could be generated, selected, and instantiated automatically. Such implementation would be a non-trivial exercise.

5.3.4 Use of Graphical Testable Requirements Models

Our VF accepts a TRM that is specified using a textual language: the ACL. With the proliferation of graphical modeling techniques, such as the Unified Modeling Language (UML) [189], and the increased use of Model Driven Design (MDD) [58], the ability to generate a TRM from an augmented UML diagram would be beneficial. Future work could examine currently used graphical modeling notations to determine what additional annotations would be required for the generation of a TRM that could be executed by our VF. Such generation would either result in the creation of an ACL TRM or the direct generation of CIL instructions. The graphical model would also be required to have the ability to produce a binary candidate IUT and a corresponding set of bindings between the TRM and IUT.

5.3.5 Interpretation of the Contract Evaluation Report

The result of our VF executing a candidate IUT against a TRM is a report indicating the outcome of each check, scenario, relation, and metric evaluation performed. Future work would examine the details presented in the Contract Evaluation Report (CER) and apply them to the development process. Such application could be used to determine the quality of a candidate IUT. In cases where development is ongoing, the CER could be

used as a measure of progress. In addition, interpretation of the CER would further validate that sufficient and useful information is being generated by the VF.

5.4 Conclusions

As Bertolino observes [34], current work on Model-Based Testing (MBT) focuses on test case generation from a specific modeling notation but such test cases are generally disconnected from an IUT. This lack of traceability constitutes, in our opinion, one of the major obstacles to the adoption of MBT.

Following Grieskamp [76], we believe MBT must entail the automated execution of tests. Also, in our opinion, a TRM must be semantically rooted in the notions of responsibilities and scenarios, which promote traceability between high-level modeling concepts and implementation constructs. In this dissertation, we have presented the elements of such a TRM and a supporting VF capable of executing the TRM against several candidate implementations. The point to be grasped is that, contrary to existing research in MBT, the semantics of our model were established *from* what can be executed automatically in our VF. In other words, it is testability that drives modeling, not the other way around. This eliminates the semantic *gap* between a requirements model and an IUT. Furthermore, our work is integrated into a widely used development environment and, through plug-ins, enables domain specializations.

References

[1] Abrial, J.R.: *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, October 1996.

[2] Adobe Photoshop: http://www.adobe.com/photoshop, accessed October 2007.

[3] AGEDIS Project: http://www.agedis.de, accessed October 2007.

[4] Aldrich, J.: *Revolutionizing Software Quality through Static Analysis Tools*. Tutorial 3 at Object-Oriented Programming Systems, Languages, and Applications Conference (OOPSLA'07), October 2007.

[5] Agile Alliance: *Manifesto for Agile Software Development*. http://agilemanifesto.org, accessed October 2007.

[6] Amyot, D., Weiss, M., and Logrippo, L.: *Generation of test purposes from Use Case Maps*. In Journal of Computer Networks, vol. 49, no. 5, pp. 643-660, Elsevier, 2005.

[7] Apple: *Mac OS X Leopard*. http://www.apple.com/macosx, accessed February 2009.

[8] Arnold, D.: *Another Contract Language 3.1 Specification Document*. http://vf.davearnold.ca/docs/ACLSpec.pdf, February 2009.

[9] Arnold, D.: *AutoBind Software Development Kit*. http://vf.davearnold.ca, February 2009.

[10] Arnold, D., Corriveau, J.P.: *Automated Instrumentation of Contracts and Scenarios for Requirements Validation in .NET*. In Proceedings of the 3rd International Workshop on Automation of Software Test (AST), Leipzig, Germany, May 2008, pp. 63-66.

[11] Arnold, D.: *C#/OCL Compiler*. http://www.davearnold.ca/csocl, accessed March 2009.

[12] Arnold, D.: *C# Compiler Extension to Support the Object Constraint Language Version 2.0.*, Masters Thesis, School of Computer Science, Carleton University, September 2004.

[13] Arnold, D: *Contract Evaluation Engine Extension Software Development Kit Documentation*. Technical Report, Carleton University, February 2007.

[14] Arnold, D.: *Contract Intermediate Language 1.1 Instruction Set*. http://vf.davearnold.ca/docs/CILInstructionSet.pdf, March 2009.

[15] Arnold, D.: *Dynamic Check Software Development Kit*. http://vf.davearnold.ca, February 2009.

[16] Arnold, D.: *Metric Evaluator Software Development Kit*. http://vf.davearnold.ca, February 2009.

[17] Arnold, D., Corriveau, J.P.: *Modeling Enhanced Scenarios for Automated Instrumentation*. In proceedings of the 6th International Conference on Software Engineering Research, Management and Applications (SERA), Prague, Czech Republic, August 2008.

[18] Arnold, D.: *Static Check Software Development Kit*. http://vf.davearnold.ca, February 2009.

[19] Arnold, D.: *The Container Example*. http://vf.davearnold.ca, February 2009.

[20] Arnold, D.: *The Grocery Store Example*. http://vf.davearnold.ca, February 2009.

[21] Arnold, D.: *The Web Example*. http://vf.davearnold.ca, February 2009.

[22] Arnold, D.: *The University Example*. http://vf.davearnold.ca, February 2009.

[23] Arnold, D.: *Validation Framework Installation and Usage Guide*. http://vf.davearnold.ca/docs/Guide.pdf, February 2009.

[24] Arnold, D., Corriveau, J.P.: *Using the .NET Profiler API to Collect Object Instances for Constraint Evaluation*. In proceedings of the 4th International Conference in Central Europe of .NET Technologies, May 2006.

[25] AutoTest: http://se.ethz.ch/research/autotest, accessed March 2009.

[26] Babes-Bolyai University: *Object Constraint Language Environment*. http://lci.cs.ubbcluj.ro/ocle/overview.html, accessed October 2007.

[27] Barnett, M., Grieskamp, W., Nachmanson, L., Schute, W., Tillman, N., and Veanes, M.: *Towards a Tool Environment for Model-Based Testing with AsmL*. In proceedings of the 3rd International Workshop on Formal Approaches to Testing of Software (FATES'03), October 2003.

[28] Barnett, M., Leino, R., and Schulte, W.: *The Spec# Programming System: An Overview*. In proceedings of Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04), pp. 49-69, March 2004.

[29] Beck, K.: *Extreme Programming Explained: Embrace Change*. Addison-Wesley, October 1999.

[30] Beck, K.: *JUnit Pocket Guide*. O'Reilly Media, September 2004.

[31] Beck, K.: *Test-Driven Development: By Example*. Addison-Wesley, November 2002.

[32] Beck, K., Cunningham, W.: *A Laboratory for Teaching Object Oriented Thinking*. In proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference (OOPSLA'89), pp. 1-6, October 1989.

[33] Beizer, B.: *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, May 1995.

[34] Bertolino, A.: *Software Testing Research: Achievements, Challenges and Dreams*. In proceedings of the Future of Software Engineering, FOSE'07, IEEE Press, Minneapolis, MN, pp. 85-103, May 2007.

[35] Beugnard, A., Jezequel, J.M., Plouzeau, N., and Watkins, D.: *Making Components Contract Aware*. In IEEE Softare, pp. 38-45, June 1999.

[36] Binder, R.: *Testing Object-Oriented Systems*, Addison-Wesley Professional, Reading, MA, 2000.

[37] Bittner, K., Spence, I.: *Use Case Modeling*. Addison-Wesley, August 2002.

[38] Boehm, B.: *Software Engineering Economics*. Prentice Hall, November 1981.

[39] Borland: *Borland Together*. http://www.borland.com/us/products/together/index.html, accessed October 2007.

[40] BOOST Library: http://www.boost.org/libs/graph/doc/table_of_contents.html, accessed October 2007.

[41] Bousquet, L., Ouabdesselam, F., Richier, J.L, and Zuanon, N.: *Lutess: A specification-driven testing environment for synchronous software*. In proceedings of the 21st International Conference on Software Engineering (ICSE'99), pp. 267-276, May 1999.

[42] Bozga, M., Fernandez, J.C., Ghirvu, L., Graf, S., Krimm, J.P., and Mounier, L.: *IF: An Intermediate Representation and Validation Environment for Time Asynchronous Systems*. In proceedings of the World Congress on Formal Methods in the Development of Computing Systems, pp. 307-327, September 1999.

[43] Briand, L., Labiche, Y.: *A UML-Based Approach to System Testing*. In Journal of Software and Systems Modeling, vol. 1, no. 1, pp. 10-42, Springer, January 2002.

[44] Buhr, R.J.A., Casselman, R.: *Use Case Maps for Object Oriented Systems*. Prentice Hall, November 1995.

[45] Burbeck, S.: *Applications Programming in Smalltalk-80: How to use Model-View-Controller (MVC)*. http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html, March 1992.

[46] Campbell, C., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N., and Veanes, M.: *Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer*. Microsoft Research Technical Report #MSR-TR-2005-59, May 2005.

[47] Campbell, C., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N., and Veanes, M.: *Spec Explorer: An Integrated Environment for Model-Based Testing*. Microsoft Research Presentation, July 2004.

[48] Campbell, C., Veanes, M.: *State Exploration with Multiple State Groupings*. In proceedings of the 12th International Workshop on Abstract State Machines (ASM'05), pp. 119-130, March 2005.

[49] Caspi, P., Pilaud, D., Halbwachs, N., and Plaice, J.A.: *LUSTRE: A Declarative Language for Programming Synchronous Systems*. In proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (POPL'87), pp. 178-188, January 1987.

[50] CodeSWAT: *Swat4j*. http://www.codeswat.com/cswat/index.php, accessed October 2007.

[51] Constant, C., Jeron, T., Marchand, H., and Rusu, V.: *Integrating Formal Verification and Conformance Testing for Reactive Systems*. In IEEE Transactions on Software Engineering, vol. 33, no. 8, pp. 558-574, IEEE Press, New York, August 2007.

[52] Corriveau, J.P.: *Testable Requirements for Offshore Outsourcing*. In proceedings of SEAFOOD'07, Zurich, February 2007.

[53] Corriveau, J.P., Bashardoust-Tajali, S.: *Generative Hierarchical Contracts for Conformance Testing of Sequential Containers*. In proceedings of the IASTED Conference on Software Engineering, Innsbruck, Austria, February 2007.

[54] Coverity: *Coverity Prevent*. http://www.coverity.com/html/prod_prevent.html, accessed October 2007.

[55] CppUnit. http://cppunit.sourceforge.net/cppunit-wiki, accessed October 2007.

[56] Crockford, D.: *The JavaScript Verifier*. http://www.jslint.com, accessed October 2007.

[57] Czarnecki, K., Eisenecker, U.: *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, June 2000.

[58] Czarnecki, K., Stahl, T., and Voelter, M.: *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, May 2006.

[59] Davies, A.M.: *Software Requirements: Objects, Functions, and States*. Prentice-Hall, March 1993.

[60] Dechter, R.: *Constraint Processing*. Morgan Kaufmann, May 2003.

[61] Deussen, P., Din, G., and Schieferdecker, I.: *A TTCN-3 Based Online Test and Validation Platform for Internet Services*. In proceedings of the 6th International Symposium on Autonomous Decentralized Systems (ISADS'03), pp. 177-185, April 2003.

[62] Dresden University of Technology: *The Dresden OCL Toolkit*. http://dresden-ocl.sourceforge.net/index.html, accessed October 2007.

[63] ECMA International: *Common Language Infrastructure (CLI) Partitions I to IV*. ECMA International Standard 335, June 2005.

[64] Eiffel Software: *The Eiffel Programming Language*. http://www.eiffel.com, accessed March 2009.

[65] Ertink, E.H., Wolz, D.: *Symbolic Execution of LOTOS Specifications*. In proceedings of the 5th International Conference on Formal Description Techniques (FORTE'92), pp. 295-310, October 1992.

[66] Evans, D., Larochelle, D.: *Improving Security using Extensible Lightweight Static Analysis*. In IEEE Software, pp. 42-51, February 2002.

[67] Fernandez, J.C., Garavel, H., Kerbrat, A., Mateescu, R., Mounier, L., and Signireanu, M.: *CADP: A Protocol Validation and Verification Toolbox*. In proceedings of the 8th International Conference on Computer Aided Verification (CAV'96), pp. 437-440, July 1996.

[68] Fernandez, J.C., Jard, C., Jeron, T., and Viho, C.: *An Experiment in Automatic Generation of Test Suites for Protocols with Verification Technology*. In Science of Computer Programming - Special Issue on COST247, Verification and Validation Methods for Formal Descriptions, vol. 29, pp. 123-146, 1997.

[69] FindBugs. http://findbugs.sourceforge.net, accessed October 2007.

[70] Fortify Software: *Fortify Source Code Analysis*. http://www.fortifysoftware.com/products/sca, accessed October 2007.
[71] FUnit. http://nasarb.rubyforge.org/funit, accessed October 2007.
[72] Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, January 1995.
[73] Garavel, H.: *OPEN/CAESAR: An Open Software Architecture for Verification, Simulation, and Testing*. In proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98), pp. 68-84, March 1998.
[74] Gentleware: *Poseidon for UML*. http://www.gentleware.com/products.html, accessed October 2007.
[75] Gosling, J., You, B., Steele, G., and Bracha, G.: *The Java Language Specification*. Third Edition, Addison-Wesley, May 2005.
[76] Grieskamp, W.: *Multi-Paradigmatic Model-Based Testing*. Technical Report #MSR-TR-2006-111, Microsoft Research, August 2006.
[77] Grieskamp, W., Gurevich, Y., Schute W., and Veanes, V.: *Generating Finite State Machines from Abstract State Machines*. In proceedings of the International Symposium of Software Testing and Analysis (ISSTA'02), pp. 112-122, July 2002.
[78] Gurevich, Y.: *Evolving Algebras 1993: Lipari Guide. Specification and Validation Methods*. pp. 9-37, Oxford University Press, September 1995.
[79] Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D.: *The Synchronous Dataflow Programming Language LUSTRE*. In proceedings of the IEEE, vol. 79, no. 9, pp. 1305-1320, September 1991.
[80] Hartman, A.: *AGEDIS: Model-Based Generation Tools*. http://www.agedis.de, 2002.
[81] Hartman, A., Nagin, K.: *The AGEDIS Tools For Model-Based Testing*. In proceedings of the International Symposium on Software Testing and Analysis (ISSTA'04), July 2004.
[82] Heerink, J., Feenstra, J., and Tretmans, J.: *Formal Test Automation: The Conformance Protocol with PHACT*. In proceedings of the 13th IFIP International Conference on Testing of Communicating Systems (TESTCOM'00), pp. 211-220, August 2000.
[83] Heitmeyer, C.: *The SCR Tool*. U.S. Naval Research Laboratory, http://chacs.nrl.navy.mil/personnel/heitmeyer.html, accessed October 2007.
[84] Helm, R., Holland, I., Gangopadhyay, D.: *Contracts: Specifying Behavioral Compositions in Object-Oriented Systems*. In proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference (OOPSLA'90), pp. 169-180, October 1990.
[85] Hewitt-Packard: *HP Code Advisor - Cadvise*. http://h21007.www2.hp.com/portal/site/dspp/PAGE.template/page.document?ciid=8b08a31f05f02110a31f05f02110275d6e10RCRD, accessed October 2007.
[86] Ho, W., Jezequel, J.M., Le Guennec, A., and Pennaneac'h, F.: *UMLAUT: An Extendible UML Transformation Framework*. In the proceedings of the 14th IEEE International Conference on Automated Software Engineering (ASE'99), pp. 275-278, October 1999.
[87] Hohmann, L.: *Beyond Software Architecture: Creating and Sustaining Winning Solutions*. Addison-Wesley, January 2003.
[88] Holzmann, G.: *The Model Checker SPIN*. IEEE Transactions on Software Engineering, vol. 23, no. 5, pp. 279-295, 1997.
[89] Hovemeyer, D., Pugh, W.: *Finding Bugs is Easy*. In the proceedings of Object-Oriented Programming Systems, Languages and Applications Conference (OOPSLA'04), pp. 92-106, December 2004.

[90] Huber, F., Schatz, B., and Einert, G.: *Consistent Graphical Specification of Distributed Systems*. In proceedings of the 4th International Symposium of Formal Methods Europe (FME'97), Lecture Notes in Computer Science, vol. 1313, pp. 122-141, Springer-Verlag, September 1997.

[91] IBM: *Rational Robot*. http://www-306.ibm.com/software/awdtools/tester/robot, accessed October 2007.

[92] IBM: *Rational Rose*. http://www-306.ibm.com/software/awdtools/developer/rose/index.html, accessed October 2007.

[93] IEEE Standards Board: *IEEE Standard for Software Unit Testing*. ANSI/IEEE Standard #1008-1987 in IEEE Standards: Software Engineering, Volume 2: Process Standards, 1999.

[94] International Standards Organization (ISO): *ASN.1 Standard*. ISO/IEC Standard #8824-1, July 2002.

[95] International Standards Organization (ISO): *Extended BNF*. ISO/IEC Standard #14977:1996, 1996.

[96] International Standards Organization (ISO): *LOTOS Specification*. ISO/IEC Standard #8807, 1990.

[97] International Standards Organization (ISO): *PIXIT Specification*. ISO/IEC Standard #9646-1, 1998.

[98] International Standards Organization (ISO): *The C# Programming Language Specification*. ISO/IEC Standard #23270:2006, August 2006.

[99] International Standards Organization (ISO): *The C++ Programming Language Specification*. ISO/IEC Standard #14882:1998, September 1998.

[100] International Standards Organization (ISO): *TTCN Standard*. ISO/IEC Standard #9646-3, 1992.

[101] International Standards Organization (ISO): *TTCN-2 Standard*. ISO/IEC Standard #9646-3, 1998.

[102] International Standards Organization (ISO): *Z Formal Specification Notation*. ISO/IEC Document #13568, 2002.

[103] International Telecommunication Union (ITU): *Message Sequence Chart (MSC)*. ITU-TS Recommendation Z.120, 1996.

[104] International Telecommunications Union (ITU): *The Evolution of TTCN*. http://www.itu.int/ITU-T/studygroups/com17/ttcn.html, accessed October 2007.

[105] International Telecommunications Union (ITU): *TTCN-3 Standard*. ITU-T Recommendations Z.140-142, 2002.

[106] International Telecommunications Union (ITU): *User Requirements Notation (URN)*. ITU-TS Recommendation Z.151, 2008.

[107] Jackson, D.: *A Comparison of Modeling Notations: Alloy, UML, Z*. Technical Report, MIT, August 1999.

[108] Jackson, D.: *Micromodels of Software: Lightweight Modeling and Analysis with Alloy*. Technical Report, MIT, February 2002.

[109] Jacobson, I.: *Object-Oriented Software Engineering*. ACM Press, New York, 1992.

[110] Johnson, S.: *Lint, A C Program Checker*. Technical Report #65, Bell Laboratories, December 1977.

[111] Jones, C.B.: *Systematic Software Development using VDM*. Prentice-Hall, April 1990.

[112] JUnit. http://www.junit.org, accessed October 2007.

[113] Kazman, R., Klein, M., and Clements, P.: *ATAM: Method for Architecturally Evaluating the Quality Attributes of a Software Architecture*. Technical Report, Software Engineering Institute, Carnegie Mellon University, August 2000.

[114] Kerbrat, A., Jeron, T., and Groz, R.: *Automated Test Generation from SDL Specifications*. In proceedings of the 9th SDL Forum, pp. 135-152, 1990.
[115] Klasse Objecten: *The Octopus Tools*. http://www.klasse.nl/ocl/octopus-intro.html, accessed October 2007.
[116] Klocwork: *Klockwork K7*. http://www.klocwork.com/products/features.asp, accessed October 2007.
[117] Knizhnik, K., Artho, C.: *Jlint*. http://jlint.sourceforge.net, accessed October 2007.
[118] Kotonya, G., Sommerville, I.: *Requirements Engineering*. John Wiley & Sons, April 1997.
[119] KPN: *The KPN Tool*. http://www.kpn.com, accessed October 2007.
[120] Kramer, R.: *iContract - The Java Design by Contract Tool*. Technical Report, Reliable Systems, 1998.
[121] LDRA: *LDRA Testbed*. http://www.ldra.co.uk/testbed.asp, accessed October 2007.
[122] Leavens, G.: *The Java Modeling Language (JML)*. http://www.jmlspecs.org, November 2006.
[123] Lecture Notes in Computer Science: *Model-Based Testing of Reactive Systems*, Volume 3472, Springer LCNS, 2005.
[124] LEDA Library: http://www.algorithmic-solutions.com/enleda.htm, accessed October 2007.
[125] Liu, L., Yu, E.: *Designing Information Systems in Social Context: A Goal and Scenario Modeling Approach*. 2003.
[126] Logilab: *Project pylint*. http://www.logilab.org/857, accessed October 2007.
[127] Marre, B., Arnould, A.: *Test Sequence Generation from Lustre Descriptions: GATeL*. In proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE'00), September 2000.
[128] Mercury Interactive: *WinRunner*. http://winrunner.info, accessed October 2007.
[129] Message Sequence Charts: http://www.sdl-fourm.org/MSC, accessed October 2007.
[130] Meyer, B.: *Design by Contract*. In IEEE Computer, vol. 25, no. 10, pp. 40-51, IEEE Press, New York, October 1992.
[131] Meyer, B.: *Object-Oriented Software Construction*. Second Edition, Prentice Hall, March 2000.
[132] Meyer, B.: *The Unspoken Revolution in Software Engineering*. In IEEE Computer, vol. 39, no. 1, pp. 121-123.
[133] Microsoft: *Code Contracts User Manual*. http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf, April 2009.
[134] Microsoft: *Code Contracts*. http://research.microsoft.com/en-us/projects/contracts/default.aspx, accessed April 2009.
[135] Microsoft: *FxCop Documentation*. http://www.gotdotnet.com/Team/FxCop, Version 1.35, July 2006.
[136] Microsoft: *Microsoft Developer Network (MSDN)*. Version 9.0 (Visual Studio 2008), November 2007.
[137] Microsoft: *Microsoft Visual Studio*, http://msdn.microsoft.com/vstudio, accessed February 2009.
[138] Microsoft: *PREFast for Drivers*. http://www.microsoft.com/whdc/devtools/tools/prefast.mspx, accessed June 2006.
[139] Microsoft: *The Abstract Machine Language (AsmL)*. http://research.microsoft.com/foundataions/AsmL, accessed October 2007.
[140] Microsoft: *The Distributed Component Object Model*, http://www.microsoft.com/com/default.aspx, accessed October 2007.

[141] Microsoft: *The .NET Framework*. http://msdn2.microsoft.com/en-us/netframework/default.aspx, accessed February 2009.
[142] Microsoft: *The .NET Framework Class Library*, http://msdn.microsoft.com/en-us/library/ms229335.aspx, accessed February 2009.
[143] Microsoft: *The Windows Operating System*. http://www.microsoft.com/windows, accessed February 2009.
[144] Microsoft Research: *Microsoft Phoenix Research Development Kit*, https://connect.microsoft.com/Phoenix, accessed February 2009.
[145] Microsoft Research: *Spec Explorer*. http://research.microsoft.com/specexplorer, accessed October 2007.
[146] Microsoft Research: *Spec# Tool*. http://research.microsoft.com/specsharp, March 2005.
[147] MIT Laboratory for Computer Science: *Alcoa: Alloy's Constraint Analyzer*. http://sdg.lcs.mit.edu/alloy, accessed October 2007.
[148] Mitchell, R., KcKim, J.: *Design by Contract, by Example*. Addison-Wesley, October 2001.
[149] Moonen, J.R., Bomijn, J., Sies, O., Springintveld, J.G, Feijs, L., and Koymand, R.: *A Two-Level Approach to Automated Conformance Testing of VHDL Designs*. Technical Report #SEN-R9707, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, December 1997.
[150] Munich University of Technology: *AutoFocus/Quest*. http://autofocus.informatic.tu-muenchen.de/index-e.html, accessed October 2007.
[151] Myers, G.: *The Art of Software Testing*. John Wiley & Sons, February 1979.
[152] Mylopoulous, L., Chung, L., and Yu, E.: *From Object-Oriented to Goal-Oriented Requirements Analysis*. In communications of the ACM, vol. 42, no 1., pp. 31-37, January 1999.
[153] Nebut, C., Fleurey, F., Traon, Y.L., and Jezequel, J.: *Requirements by Contracts allow Automated System Testing*. In proceedings of the 14th International Symposium on Software Reliability Engineering, ISSRE, IEEE Computer Society, pp. 85-105, Washington, DC, November 2003.
[154] Nielson, F., Nielson, H., and Hankin, C.: *Principles of Program Analysis*. Springer, December 2004.
[155] Novel Corporation. http://www.novell.com, accessed February 2009.
[156] NUnit 2.0. http://www.nunit.com, accessed October 2007.
[157] Nuseibeh, B.A, Easterbrook, S.M.: *Requirements Engineering: A Roadmap*. In The Future Of Software Engineering, IEEE Computer Society Press, 2000.
[158] Objecteering Software: *Objecteering UML Modeler*, http://www.objecteering.com/products.php, accessed October 2007.
[159] Parasoft: *Jcontract*. http://www.parasoft.com/jsp/products/home.jsp, accessed October 2007.
[160] Philips: *PHACT: Philips Automated Conformance Tester*. http://www.philips.com, accessed October 2007.
[161] Ploderer, E., Koschke, R.: *The Bauhaus Project*. Institute of Software Technology, University of Stuttgart, http://www.iste.uni-stuttgart.de/ps/bauhaus/index-english.html, accessed October 2007.
[162] Pohl, K., Haumer, P.: *Modeling Contextual Information about Scenarios*. In proceedings of the 3rd International Workshop on Requirements Engineering: Foundations of Software Quality (REFSQ'97), pp. 187-204, June 1997.
[163] Pretschner, A.: *Compositional Generation for MC/DC Test Suites*. In proceedings of the International Workshop on Test and Analysis of Component-Based Systems (TACoS'03), Electronic Notes in Theoretical Computer Science, vol. 82, no. 6, pp. 1-11, April 2003.

[164] PyUnit. Unit Testing Framework, Python Library Reference, http://www.python.org/doc/current/lib/module-unittest.html, accessed October 2007.
[165] Raymond, P., Nicollin, X.Halbwachs, N., and Weber, D.: *Automatic Testing of Reactive Systems*. In proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS'98), pp. 200-209, December 1998.
[166] Rutar, N., Almazan, C., and Foster, J.: *A Comparison of Bug Finding Tools for Java*. In the 15th International Symposium on Software Reliability Engineering, pp. 245-256, November 2004.
[167] Ryser, J., Glinz, M.: *SCENT: A Method Employing Scenarios to Systematically Derive Test Cases for System Test*. Technical Report. University of Zurich. http://www.ifi.uzh.ch/rerg/research/scent/, accessed January 2009.
[168] Ryser, J., Glinz, M.: *Using Dependency Charts to Improve Scenario-Based Testing: Management of Inter-Scenario Relationships*. In proceedings of the 17th International Conference on Testing Computer Software, TCS2000, pp. 1-10, Washington, DC, June 2000.
[169] Schmitt, M., Ebner, M., and Grabowski, J.: *Test Generation with AutoLink and TestComposer*. In proceedings of the 2nd Workshop on SDL and MSC (SAM'00), June 2000.
[170] Selic, B.: *The Pragmatics of Model-Driven Development*. In IEEE Software, October 2003.
[171] Sommerville, I.: *Software Engineering 8th edition*. Addison-Wesley, 2006.
[172] Stepanov, A., Lee, M.: *The Standard Template Library*. Technical Report, HP Laboratories, November 1995.
[173] Stepien, B.: *TTCN-3 in a Nutshell*. http://www.site.uottawa.ca/~bernard/ttcn3_in_a_nutshell.html, accessed October 2007.
[174] Stobie, K.: *Model-Based Testing Practice at Microsoft*. In proceedings of the Workshop on Model-Based Testing (MBT'04), March 2004.
[175] Sun Microsystems: *The Java Programming Language*. http://java.sun.com, accessed February 2009.
[176] Sun Microsystems: *The Solaris Operating System*. http://www.sun.com/software/solaris, accessed February 2009.
[177] Telelogic: *Telelogic Logiscope*. http://www.telelogic.com/products/logiscope/index.cfm, accessed October 2007.
[178] Telelogic: *Telelogic Tau*. http://www.telelogic.com/products/tau/index.cfm, accessed October 2007.
[179] Telelogic: *Telelogic Tau Tester*. http://www.telelogic.com/products/tau/tester/index.cfm, accessed October 2007.
[180] The Eclipse Project. http://www.eclipse.org, accessed February 2009.
[181] The GATeL Tool: http://www.list.cea.fr/labos/gb/LSL/test/gatel/index.html, accessed October 2007.
[182] The Institute of Electrical and Electronics Engineers (IEEE): *IEEE Standard for 802.11b*. September 1998.
[183] The Institute of Electrical and Electronics Engineers (IEEE): *IEEE Standard for 802.11g*. August 2006.
[184] The Linux Operating System. http://www.linux.org, accessed February 2009.
[185] The MathWorks: *Simulink*. http://www.mathworks.com/products/simulink, accessed October 2007.
[186] The Mono Project. http://www.mono-project.com, accessed February 2009.
[187] The Object Management Group (OMG): *The CORBA Standard*. OMG Document #04-03-12, March 2004.

[188] The Object Management Group (OMG): *The Object Constraint Language (OCL) Specification*. Version 2.0, OMG Document #06-05-01, May 2006.
[189] The Object Management Group (OMG): *The UML 2.1.1 Specification*. OMG Document #07-02-03 and #07-02-04, February 2007.
[190] The Object Management Group (OMG): *UML Profile for Software Defined Radio Version 1.0*. OMG Document #07-03-01, March 2007.
[191] The Open Group: *The Unix System*. http://www.unix.org, accessed February 2009.
[192] Tiobe Software: *ClockSharp C# Checker*. http://www.tiobe.com/index.htm?clocksharp, accessed October 2007.
[193] TorX: http://www.purl.org/net/torx, accessed October 2007.
[194] Tretmans, J., Brinksma, E.: *Cote de Resyste - Automated Model-Based Testing*. In proceedings of the 3rd Workshop on Embedded Systems, pp. 246-255, October 2002.
[195] Tretmans, J., Brinksma, E.: *TorX: Automated Model-Based Testing*. In proceedings of the 1st European Conference on Model Driven Software Engineering, pp. 31-43, December 2003.
[196] UMLAUT: http://www.irisa.fr/UMLAUT, accessed October 2007.
[197] Use Case Maps: http://jucmnav.softwareengineering.ca/twiki/bin/view/UCM/WebHome, accessed October 2007.
[198] Utting, M., Legeard, B: *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann, November 2006.
[199] de Vries, R.G., Tretmans, J.: *On-The-Fly Conformance Testing Using SPIN*. In proceedings of the 4th Workshop on Automata Theoretic Verification with the SPIN Model Checker (SPIN'98), pp. 115-128, November 1998.
[200] W3C: *The Simple Object Access Protocol (SOAP)*. Specification Version 1.2, April 2007.
[201] Weidenkaupt, K., Pohl, K., Jarke, M., and Haumer, P.: *Scenario Usage in System Development: A Report on Current Practice*. In proceedings of the 3rd International Conference on Requirements Engineering: Putting Requirements Engineering to Practice (ICRE'98), pp. 222-241, April 1998.
[202] Woodside, M., Franks, G., and Petriu, D.: *The Future of Software Performance Engineering*. In Future of Software Engineering (FOSE'07), pp. 171-187, IEEE Press, New York, May 2007.
[203] Yang, Q., Li, J., and Weiss, D.: *A Survey of Coverage-Based Testing Tools*. In proceedings of the 2006 International Workshop of Automation of Software Test (AST'06), pp. 99-103, September 2006.
[204] Yourdon, E.: *Death March*. Yourdon Press, November 1993.
[205] Yu, E.: *Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering*. In proceedings of the 3rd IEEE International Symposium on Requirements Engineering (RE'97), pp. 226-235, January 1997.