A Framework for Integrating Automated Software Verification Tools

OSU_CISRC-4/12-TR05

Dustin Hoffman
Department of Computer Science and Engineering
The Ohio State University
hoffman.373@osu.edu

May 4th, 2012

Abstract

Introduces a framework for integrating VC generators, tools, and provers in order to more easily research automated theorem proving and to create verification applications.

1 Introduction

This paper describes the Modular Verification Environment (MVE), a generic framework designed to allow researchers in the field of automated verification to easily integrate Verification Condition (VC) Generators, tools, and provers into applications. Current solutions provide a fixed toolset for verifying software. These systems are designed with developers of production software in mind. The MVE framework is provides researchers an easier method of experimenting with verification tools or creating fixed toolsets for other developers to use.

The technique for integrating these things can be thought of like the plumbing of a house. Programs flow into the system, are directed to fixtures, which perform some useful action, and then continue to flow through the system until they reach the end. In 1 you see programs and specifications flowing into the system on the upper left hand side. These items first flow into the VC generation fixture, before flowing out of that into a SAT solver on the left. On the right, the VCs first flow into a tool for injecting lemmas and then finally into a prover.

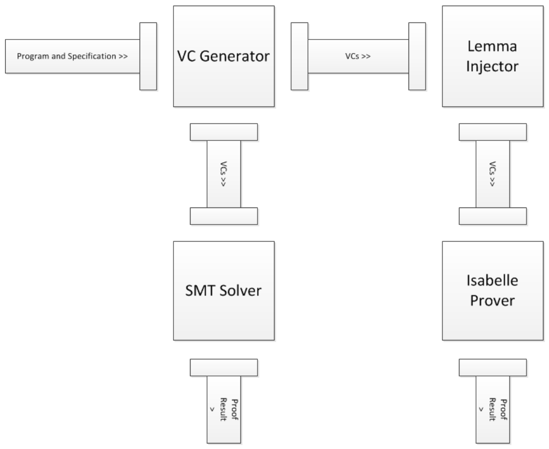

Figure 1: “Piping” of data through the framework.

2 Motivation

Why would someone need the MVE? The MVE originated during a project in which the Resolve group was studying techniques for increasing the number of valid verification conditions (VC) successfully proven. Our technique involved taking a VC, manipulating it, and then attempting to prove the resulting VC. We wanted to test our techniques using multiple provers but were unable to find an easy mechanism which would allow us to implement our logic in a single location and easily integrate it with all of the provers. Thus, being able to create tools capable of manipulating VCs prior to verification, despite the prover, was set as our first requirement.

Creating a web based user interface capable of editing and verifying components was our next major requirement. This is functionality which has been provided by multiple verification projects, including Microsoft Research's Dafny [3] [4] or Clemson's variant of Resolve[7]. We felt that this form of interface provides an excellent environment for studying techniques for both verification itself, as well as interacting developers. Despite excellent products by others, there did not exist a framework geared specifically for creating interactive websites for automated verification.

These two goals (enabling the creation of VC tools and a web based verification IDE) became the minimum requirements needed in order to carry our research forward. However, we realized that in the processes of creating our tools, it would be possible to create a generic framework which can benefit verification researchers and tool creators. Doing so lowers the barrier of entry for researchers without access to a large pool of graduate research assistants.

Creating a standard mechanism for connecting VC generators to tools and provers fosters the creation of an environment to enable researchers to study and improve individual pieces of a verification system. For example, assume you want to add support for formal specification and verification to a domain specific language (DSL) for trading stocks. With the MVE all you need to develop is a new VC generator for your language. Once you have done this, you are able to use provers and tools which already exist for the MVE framework in order to begin verifying programs in your DSL. This contrasts to the current state of separate and incompatible input constructs for individual tools and provers. In that environment, you need to create an interface to each tool and prover you wish to use.

Continuing with DSL example, assume that another researcher, lets call her Sue, is using the aforementioned DSL language. However, assume that the automated theorem provers are having trouble with VCs containing expressions involving pork bellies. Sue realizes that if she converts the expressions to another form, they are able to quickly go through any of the provers she is using. Sue is be able to use this system to create a tool capable of altering, splitting, or otherwise changing to VCs prior to sending them to the theorem provers. Sue can do this for all theorem provers in a single go, and does not have to worry about defining the DSL language or generating VCs.

In the third section, we provide an overview of the specific design goals for the MVE and how they were achieved. In the subsequent section we provide examples of how the environment is being used to provide specific functionality. Section five explains how a developer can use and extend the framework. The final two sections provide an overview of previous work and some goals for future work related to the MVE.

3 Previous Work

Previous work in frameworks and toolchains for automated verification focus on attempting to create a product that a developer would be capable of using to verify their product. The MVE framework differs in two significant ways. First, it provides researchers to ability to create applications like the toolchains and frameworks others have made. Secondly, it provides a great deal of flexibility in the configuration of the fixtures in order to allow researchers to study the verification tools. In the rest of this section we demonstrate how previous works have focused on creating products.

3.1 A Software Framework for Automated Verification

Raedts et al. [6] present a framework for verification focused on model verification. They provide a web site which allows users to submit models which are then sent through individual tools. Their approach differs from ours in several key areas. First, they do not attempt to provide a method of integrating tools together in a standard fashion. The tools do not appear to be able to feed into one another. Secondly, they do not appear to provide a means of creating applications (e.g. IDEs) on top of the verification tools.

3.2 Verified Software Toolchain

In his paper Verified Software Toolchain, Appel [2] describes a complete tool chain for verifying programs in C *minor* [5]. The paper differs from this work in that it focuses on a tool chain for a specific language and proof system were as our tool can be used to create verification packages such as VST.

4 Modular Verification Environment Overview

In this section, the high-level design for the MVE is provided in order to provide the necessary background before we examine its details. A key requirement added to the design was that it needs to be able to support different languages and verification tools. This feature greatly impacted the design of the MVE and these places will be highlighted at appropriate times throughout this, and subsequent sections.

The framework was implemented in Java. This was chosen because of its cross platform support and wide developer base. The design and techniques used to implement the framework could easily be ported to another language or platform.

The framework is designed in two layers, shown in Figure 2. The bottom layer is the core of the framework. The top layer is segmented into two logical compartments. The left compartment contains applications built on top of the framework while the right contains fixtures (e.g. VC generators, tools, and provers) which can be used with the framework. Communication between pieces in separate compartments on the top layer is expected to go through the core.

4.1 MVE Core

The core of the MVE framework consists of few important pieces upon which verification applications are built. The first of these is the verification tool tree. The next is the representation used for moving verification data through the tool tree. The final core piece is the component repository and abstract component reference.

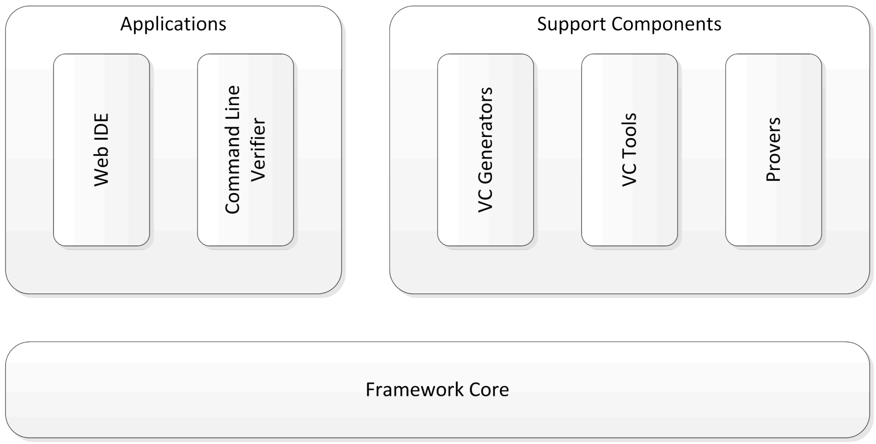

Figure 2: High level block design.

4.1.1 Tool Trees

The verification tool tree consists of a tree of fixtures. When the tool tree is executed with a given input component, the results of one fixture is fed into the next fixture(s). The tree structure allows users of the framework to create configurations which direct VCs through multiple providers in a single pass. For example, a researcher can test the performance of an experimental prover against that of an existing SMT solver.

The core provides support for defining tool trees as XML configuration files or as Java objects created by applications built on top of the framework. The framework also includes the plumbing necessary to execute the tool tree.

Results for each fixture in the tool tree are provided through observers. Observers are implemented using the traditional observer pattern and allow data to be extracted from the tool chain in a manner suited to the application's needs. For example, an IDE wishing to highlight sections of unverified code may only care if the final verification stage was successful or failed. Its observer need not do anything with intermediate results. A command-line tool, on the other hand, may have a requirement of logging the results of each step in the process for later debugging. Its observer can handle recording the VCs as they pass from each fixture in the tool tree.

4.1.2 VC Representation

Data are passed between the providers in a tool chain using a standard VC representation component. The unit of information exchange between fixtures is a mathematical statement along with some state. An object based mathematical representation was lifted from an XML schema used for the same purpose by previous Resolve tools. However, the representation used in the MVE separates itself from being tied to the XML structure used in those tools. This requires marshaling to and from different text representations of a mathematical language to be handled externally of the components used to store VCs. This VC representation provides a rich enough feature set to express the theorems, definitions, type information, and VC statements needed for automated verification, as far as we know how. If more expressiveness is needed, the VC representation can be expanded with minimal impact to other major pieces of the framework.

4.1.3 Component Repository and Component References

A key part in ensuring that the framework can be used for different languages, without changing prover interfaces or tools, is the component reference and the component repository interfaces.

A component reference is an implementation of a special empty Java interface. The framework is built around accepting these component references when dealing with items to be verified instead of something more concrete such as file names. When the framework is adapted for a specific language, the developer performing the adaptation must provide an implementation of a component reference which matches the language's model. Then defining a component reference, the developer must also provide an instance of a Java interface that performs conversion between component references and strings.

Component repositories serve as an abstract method of accessing components. The repositories provide methods which allow you to navigate the components in a tree-like fashion. Additionally, the repositories are designed to be layered in order to allow support for features such as having certain components read only and others writable. Despite these features, the key benefit given by repositories is the ability to enable to the tool chain to access components stored in various locations. For example, assume you were trying to add a real time verification support to an IDE. The repository feature allows you to seamlessly access component that have been edited in memory and not yet saved to disk without modifying any of the tools.

5 Example Applications

This section provides an overview of two applications developed using this framework, along with a third application which this framework enables that, as far as we know, is not being supported by any tools to date.

5.1 Web IDE

One of the original goals which initiated this project was to create a web based IDE for the Resolve language and tools. To that end, we have built a web IDE on top of the MVE framework which allows users to navigate a repository, view/edit components, and to then verify the components. This is functionality common in web based IDEs for other verification systems such as Dafny and the Clemson edition of Resolve.

What makes our solution stand out is that a researcher is able to access the modular nature of the verification engine. For example, we allow researchers to configure tool trees and then use those tool chains to verify their components. This allows alternate setups to be easily tested in the laboratory without ever leaving the IDE. The next important difference between our IDE and other web based verification IDEs is that we have not tied the software to the Resolve language. In fact, the IDE does not link against the library of Resolve providers and support. During the development of the IDE, the framework was designed to help ensure other languages could provide the support necessary for the easy creation of this IDE or another.

The web IDE provides a test bed for research into methods for interacting with developers. Techniques such as those espoused by Adcock[1], or integrated into Dafny's Visual Studio[4] interface, can be expanded upon and enhanced. The web IDE is available with the MVE in order to provide an open, language neutral environment for researching verification IDEs.

5.2 Command Line Application

A command line application was created on the framework for verifying components using tool chains specified in XML configuration files. The application allows developers run tools outside of a graphical environment. The application could be helpful in tasks such as integrating verification into automated builds and source control pre-checks. From this MVE project's standpoint, the application provides a simple sandbox in which developers of providers can test tools and provers from within a Java IDE.

5.3 Verification as a Service

Automated verification systems rely on theorem provers to prove that code matches a specification. Configuration, development of theorem libraries, and other tasks associated with provers can be a daunting task for developers. The framework allows for the inclusion of remote providers, and can help in two major ways. The first, and least surprising, is that it can alleviate the overhead associated with prover maintenance. In this manner, theorem provers can become a service of the cloud. This can allow small development firms and individual researchers with limited resources to work with verification tools without being an expert in each tool they use.

The second benefit to being able to send verification tasks away from the developer's computer is the inclusion of mathematicians. The Resolve methodology has long acknowledged that developers need support from trained mathematicians and logicians in order for automated verification succeed. Having the ability to forward troublesome VCs, from within the development environment, directly to mathematicians (either within an organization or via a cloud service) takes a key step in bridging this gap.

6 Developing with the Framework

In this section we explain how a developer can use the framework. First, we cover what is required to create a prover or tool provider. Next, we explain what is necessary to add support for a new language and the integration of a VC generator for the language. Finally, we explain how the framework can be integrated into other tools requiring access to its provers and tools.

6.1 Integrating Provers and Tools

Provers and tools are fixtures that take VCs as input and either attempt to prove them (in the case a prover) or modify them (in the case of a tool). It is a design goal of the system that, when possible, these providers should depend solely upon the information provided in the VCs they take as input. This goal is intended to help ensure that the provers and tools can work with new languages without modification.

This guideline does not to apply to theorem libraries, although the VCs themselves are expressive enough to carry lemmas with them. It is expected that, for performance and ease of development, provers would have access to theorems in a manner consistent with the prover being integrated. Additionally, if situations arrise in which additional information needs to flow through the tool tree, each VC is bundled with a name value map.

The steps necessary to get a minimal prover fixture into the system are as follows:

1. Create a Java project.

2. Link against the libraries MVEBase, VCModel, and components.jar.

3. Create a class that implements IVCProver. This class handles accepting the VC that is to be proved and interfacing with a prover in order to perform the proof attempt.

4. Create a service entry in the META-INF services folder for the project or JAR file.

5. Add the project or resultant JAR file to the class path of the web IDE, command line tool, or other application wishing to use the fixture.

These steps are all that are necessary to integrate a prover with the system from a plumbing stand point. The MVE project includes a project called NullTools which contains sample toy tools, provers, and VC generators which serve as examples of how to develop for the platform. A tool (a fixture which takes VCs as input and generates one or more VC as output) is created in the same steps, except in step 3 the class must implement IVCTool.

Once the new fixture is building and the applications can see it in their class path, the next step is to actually convert from the VC representation to the format appropriate for the given prover. Finally, the prover can be added to a given tool tree in order to test it out.

6.2 Adapting for a New Language

The framework has been designed from the ground up to be adaptable to a new language. The basic steps necessary to support a new language are:

- Create an implementation of IComponentReference for components in the target language.
- Create an implementation of the IComponentReferenceFactory for the newly created Component Reference.
- Create an implementation of IComponentRepository (or provide support for the included FileRepository implementation, see below)
- Create a VC generator which is capable of processing components from your target language and generating VCs from them.
- Configure the target application to use your new fixture (Typically, one must simply add to the class path and create tool chain wich uses them.)

Through the rest of this section we provide an overview of how each of these steps can be completed. Additionally, the MVE project includes an adaptation for the Resolve language which serves as an example of how to integrate a language.

First, we shall discuss is the implementation of IComponentReference. A component reference must be capable of uniquely referencing a component in the system. Fixtures (VC generators, tools, and provers) do not have any means of examining the component reference. They have some instance (provided by you) of the IComponentReferenceFactory which allows them to marshall component references to and from Strings and they have a IComponentRepository which is capable of retrieving an input stream to the component. This allows the framework to remain language agnostic while still being able to uniquely reference components.

The implementation of a the component reference you provide is dependent upon the language. For Resolve, components are referenced by kernel[enhancement][realization] (where the enhancement and realization are optional). Therefore, the ResolveComponentReference deals with storing these three pieces appropriately. For example, if you implement a component reference for Java classes, it very well make sense to store the fully qualified class name inside of the component reference.

The next step is the creation of the component repository. The component repository allows applications built on the framework to enumerate the components in the repository and to retrieve the content of the components themselves. Typically, you can avoid creating a repository from scratch and use the FileRepository class provided in MVEBase. In order to use the FileRepsitory, you must provide an implementation of the IComponentReferenceResolver which provides suport for determining which files and folders constitute components, as well as, provide a mapping between a component and the file which *should* contain it.

The third step, is the creation of a VC generator. The VC generator is created in exactly the same steps as Section 6.1, except the new class must implement the IVCGenerator interface instead of IProver or IVCTool.

Once these steps have been completed, the application you wish to integrate the language into must be configured to use the new components. For the provided web IDE, one must simply add the project or JAR to the websites class path, and then add the repository to the repository configuration file in the web site folder.

7 Future Work

7.1 Adaptation to Logic Education System

A project is underway to adapt the Syrus logic education system to utilize the MVE tool chain system as its underlying architecture. This update to the Syrus application will help it to gain flexibility through access to the verification resources afforded to applications built using MVE. Secondly, this project will demonstrate the suitability of the MVE for adaptation into applications that are neither specifically focused on software verification nor utilizing the Resolve programing language.

7.2 Integrate New Verification Source Language

Developing a VC Generator and the associated components necessary to verify a new language would service as an interesting exercise for several reasons. First, this would demonstrate our believe that the framework provides enough support to adapt to different languages. Secondly, if the prover interfaces used for Resolve were reused, it would demonstrate how a language can designer can focus on finding efficient methods of specification instead of focusing on verifiers.

7.3 Expand Resources for VC Generators, Tools, and Prover Interfaces

Another important opportunity for expansion of the framework is to take the idea of creating generic components to help tool creators and apply it to VC generators, tools, and prover interfaces. For example, the successful creation of a framework for creating VC generators could significantly reduce the time and effort required for adding suport for a new language or adding features to an existing one.

7.4 Abstract the Fixture Exchange Data Type

The current version of the framework uses a VC (A mathematical statement with context) as the unit of exchange between fixtures. Currently, the framework is designed to only use the included VC model. Adapting the framework to allow fixtures to accept different data types could open the framework up to even more applications. However, this enhancement must take caution not to increase the flexibility too far at the cost of complexity.

References

[1] *Working Towards the Verified Software Process*. PhD thesis, The Ohio State University, Columbus, Ohio, 2011.

[2] Appel, A. W. Verified software toolchain. In *Proceedings of the 20th European conference on Programming languages and systems: part of the joint European conferences on theory and practice of software* (Berlin, Heidelberg, 2011), ESOP'11/ETAPS'11, Springer-Verlag, pp. 1–17.

[3] Leino, K. R. M. Dafny Web Interface. `http://rise4fun.com/Dafny/nrM`.

[4] Leino, K. R. M. Dafny: an automatic program verifier for functional correctness. In *Proceedings of the 16th international conference on Logic for programming, artificial intelligence, and reasoning* (Berlin, Heidelberg, 2010), LPAR'10, Springer-Verlag, pp. 348–370.

[5] Leroy, X. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. *SIGPLAN Not. 41*, 1 (Jan. 2006), 42–54.

[6] Raedts, I., Petković, M., Serebrenik, A., van der Werf, J. M., Somers, L., and Boote, M. A software framework for automated verification. In *Proceedings of the 2007 ACM symposium on Applied computing* (New York, NY, USA, 2007), SAC '07, ACM, pp. 1031–1032.

[7] Sitaraman, M. Clemson RSRG Web IDE. `http://resolve.cs.clemson.edu/interface/`.