A User Interface for Algorithmic Debug

By

Juan Miguel Rodriguez Gallegos
S.B. EECS. MIT, 2017

Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science
at the
Massachusetts Institute of Technology
May, 2018

© 2018 Massachusetts Institute of Technology. All rights reserved.
1 Abstract

The hardware verification pipeline for server-scale system on chips (SoCs) is as complex as the SoCs themselves. Intel's Validation Acceleration through Shared Expertise (VASE) tool allows teams at different stages of the verification pipeline—from subsystem development to system integration—to share key architectural, test, and debug knowledge. By enabling system experts to automate the detection of common simulation failures, VASE allows the engineers who are inheriting a subsystem to instantly have access to their expertise whenever a simulation failure occurs. The VASE tool has the potential to greatly increase validation efficiency, but its adoption into Intel's verification work flow is put at risk by usability issues. To address these shortcomings, I developed a user interface that facilitates the creation of the simulation failure debug decision trees by the system experts. This GUI aims to increase usability for tree definition and manipulation, to enforce node subtree coherence across all trees, and to provide a tree building solution which scales with the complexity of the system.
2 Introduction

My VI-A M. Eng thesis work was carried out at Intel with the Scalable Performance CPU Development Group, working on their Validation Acceleration through Shared Expertise (VASE) tool. VASE allows teams at different stages of the hardware verification pipeline—from subsystem development to system integration—to share key architectural, test, and debug knowledge. It provides a means for engineers to express and automate their debug process so that others working with the same system or subsystem can harness their expertise and experience.

VASE has the potential to vastly improve validation efficiency, and is in the early stages of deployment across multiple teams at Intel; whether or not it will be effective, however, is highly dependent on the willingness and ability of engineers to understand it and to incorporate it into their workflow. One major barrier to
this is the current text-based method of user-input; in order to create a decision
tree in VASE for others to make use of, an engineer must follow a strict syntactical
structure to create a text-based representation of their debugging process. This is
unintuitive, error-prone, and time-consuming.

To address this problem and increase the accessibility of this powerful tool, I
designed and implemented an improved user interface built on the VASE tool. The
interface layer provides a graphical method of creating and modifying decision trees
that lowers VASE’s learning curve and allows engineers to represent their ideas in
a way that is more similar to how they naturally think about their debug flow. It
aims to both increase the rate of adoption of VASE at Intel and provide current
VASE users with a faster, less error-prone method of using the tool.

The main design goals of the interface are: to provide a simple, intuitive user
interface; to be quick and easy for new users to install and get started using; to
provide real benefits, in terms of time saved and errors committed; and to be im-
plemented in such a way that is easy to understand and modify. The final interface
was implemented as a desktop app using the Python Tkinter package. The model-
view-controller programming paradigm was used for simplicity and readability.

3 Background

3.1 Hardware Validation Workflow

The key to the successful development of any complex system is robust verification
procedures; engineers must ensure that a system design correctly and fully imple-
ments the specification of the product. As a result, more than half of an engineer's
time can be spent debugging [1]. Despite this, “debugging” remains an ill-defined
and project-specific process that typically requires intimate knowledge of the system
in question. This becomes a particularly problematic in the engineering of complex
System on Chips (SoCs), where multiple teams are involved in the implementa-
tion and integration of multiple subsystems. Particular hardware subsystems may
be passed from one team to another, and hardware validation engineers often find themselves tasked with debugging a system that they themselves did not design. They cannot simply treat a received subsystem as a black box, since their work on the system often re-introduces failures. When problems occur, the new team is heavily dependent on the expertise of the original team, and their debugging efforts may become redundant as they solve problems already encountered by the original team. This dependency is a burden for the original team, who have likely moved on to another project and don’t want to waste time solving what are now another team’s problem.

There is therefore a need for a better system of knowledge sharing between hardware validation teams in order to reduce redundant work and help speed up the debugging process.

3.2 Existing Knowledge Sharing Systems

Many knowledge sharing systems do exist, and have been adopted into company workflows in order to facilitate communication and reduce redundant work. Popular examples of collaboration software include Slack, Zoho Projects, and Asana [2]. These services provide employees with a platform for casual conversations, asking questions, and sharing their knowledge with each other. While helpful, they are meant for general, and not necessarily technical, information, and are therefore not tailored to the needs of hardware validation engineers looking to share knowledge about the system they are building.

At Intel specifically, multiple platforms are used for communication and knowledge sharing: Skype is used for casual conversations and video-conferencing; Wiki pages provide a source of general proprietary information useful for new hires; and technical documentation is often compiled into PDF form and linked on Wiki pages. Therefore, the main source of shared knowledge is PDF documentation and Wiki pages. Although these pages do provide valuable resources, they are a source of static information that may not answer all debugging questions.
Question and answer systems such as Stack Overflow and Quora provide another knowledge sharing framework, popular among programmers, that provides an opportunity for experts in a topic to share their knowledge [3]. Redundant work can be reduced by sharing solutions to common problems. Although they are valuable resources, especially for many introductory level developers, the question and answer format of these sites is not ideal for the hardware validation use case. For instance, the timeline for answering a question is open-ended, meaning that some questions remain unanswered for months. Also, the question and answer format assumes that the person asking the question does not know where to direct their question. However, in the case of Intel’s hardware validation team, each group has specified knowledge. Therefore, a given query can likely only be answered by a small handful of people. Specifically, when a subsystem is passed from team A to team B, it is team A that will have the specified knowledge team B may need in order to debug any problems that arise in the subsystem. Effective documentation may provide the answers to some questions that that team B may have, but is often not useful when components do not function as expected.

3.3 The VASE tool

The VASE tool takes a proactive approach to knowledge sharing appropriate to hardware validation. It allows teams to encapsulate valuable debug information in an easy-to-understand manner that can then be shared with others working on the same subsystem further down the pipeline. It does so by providing a method for engineers to automate the detection of common system failures encountered during hardware validation, along with their solutions. The goal of VASE is to allow engineers to record solutions to common problems in the system as they are working on it, so that those who later inherit the system can harness the ability to diagnose these common failures. This prevents engineers from having to return to a project after they have completed it, instead taking advantage of their knowledge of the project when it is still fresh in their minds. It also eliminates the need for
engineers further down the pipeline to either reach out for information or figure out the problem on their own, thus increasing efficiency and reducing redundancy.

3.4 Shortcomings of the VASE tool

Although the VASE tool provides a more effective knowledge sharing system that is tailored to hardware validation, the tool’s current implementation has several shortcomings related to how engineer’s input debugging information about their system. Input to the current VASE tool is in the form of a text file, which is cumbersome to modify and error-prone; large systems are difficult to describe; and it is on the user to maintain coherence between subtrees. These shortcomings, discussed in more detail in Section 3.5, were the motivation for my M.Eng. thesis project: the creation of a better user interface for VASE to improve usability, reduce errors, and provide engineers with a more accessible tool for knowledge sharing.

4 Description of the current VASE tool

4.1 Goals of VASE

The VASE tool allows validation teams to avoid redundancy by providing them with a method of sharing key architectural, test, and debug knowledge. By enabling engineers to embed their hard-won knowledge of a system into an automated post-processor of verification test results, it aims to achieve the following:

1. **Effort savings**: By automating the common first steps that hardware validators carry out while debugging a new failure, VASE transfers monotonous work from the validator to a machine.

2. **Accelerated ramp**: VASE assists new team members in learning valuable debugging skills, by drawing their attention to the highest-value debug clues that others have found.
3. **Enhanced communication:** VASE enhances the sharing of key insights about specific events and flows in the system between engineers and teams.

4. **Bucketing:** VASE deciphers the failure signature of known bugs to save engineers time when dealing with similar problems.

The VASE tool aims to provide a means for people to share their debugging expertise, by promoting a debug algorithm that applies to any system. It is built on the paradigm that all debug can be encompassed as a decision tree.

![Figure 1: Graphical representation of a debugging decision tree](image)

Each node in the tree represents an unexpected system behavior that should be checked for, with the children of a given node representing possible causes of the behavior. The root node of the tree is a failure signature, which defines when the debug process will start. If no nodes are missing, traversal of the tree will yield a path to a leaf node representing the cause of the issue under debug. This process is analogous to the mental process of debugging that, whether they realize it or not, both expert and novice debuggers of a given system already carry out. The experts, however, already have an intuition for the structure of the tree and what assumptions they should check to prune a number of dead ends. The VASE tool is meant to encourage experienced validators to think about their debug process in...
this way and enable them to express and automate their debug decision trees so that they can share them with other teams.

4.2 The role of developers and users

The VASE tool has two types of users: the VASE tree ‘developer’ and the VASE tree ‘user’. The developer is typically a system expert who often finds themselves consulted by other engineers to figure out the cause of simulation failures. The developer can make use of the VASE to alleviate this consultation load by using the tool to express a debug decision tree that diagnoses common failures for their system. The VASE tool accepts a text-based description of a debug decision tree as input, and in turn generates template Python classes for each node in the decision tree. The Python class for each node defines the conditions under which the node evaluates to true, as well as the next steps to be taken if the node evaluates to true. The developer is responsible for filling out each Python class with the appropriate definitions, by processing the data that results from a simulation run for their system. The developer can also embed messages that will be provided to the user in the case of detection of a common failure, or in the event that none of the common failure types were the cause of an issue. Once the tree is complete and nodes are defined, the developer submits their completed Python collateral into the VASE repo, a centralized Git repository, for other users to access.

The VASE tree ‘user’ is typically someone who does not have deep knowledge for a given system, and often finds themselves needing to consult experts when a system simulation run results in failures. Examples of tree users would be engineers who inherit a subsystem and are tasked with integrating it into a larger system, or novices on the system’s developer team. If system experts have taken the time to develop VASE debug decision trees for their respective systems, the VASE tree user can now wield all their respective system knowledge without an in-person consultation. The VASE tool allows the user to execute any decision trees found in the centralized VASE repository on their simulation run data. If the failures experienced
are among the common failure buckets detectable by the decision trees, execution of the trees will inform the user of the types, causes, and debug hints/instructions for the given failures, all without the need to consult the system experts. If the failures experienced are not among the common failure buckets, execution of the trees will still collect and provide meaningful diagnostic information. In the latter case, the user can then reach out to the experts with all the information the expert would’ve manually collected already in hand.

A graphical overview of the role of VASE developers and users is shown in Figure 2.

![VASE Tree Developer and User Diagram](image)

**Figure 2:** VASE developers and users

### 4.3 Current text file input

When a VASE developer wants to generate Python classes for the nodes in their debug tree, they must first provide a file definition of the tree to the VASE tree code generator. In the current implementation of VASE, trees are defined in a text file. The text file tree definition consisted of nodes delimited by newline characters, and node depth defined by the number of tab characters preceding the name of the node. Since VASE trees must have a single root node, the first node in the file must be of depth zero, and all other nodes must be of depth one or greater. Node names need
to be Python identifier compliant, since the VASE tree code generator generates a
Python class definition file per node. An example of a text-based tree definition for
PCI Express hardware failures is shown in Figure 3.

PcieTestDidNotFinish
  PcieHasTrafficNearEOT
    LinkWidthIsLimitingFlow
    CreditsAreInsufficient
  PcieIsHung
    LongUpstreamIsAnIssue
      LinkWidthIsLimitingFlow
      CreditsAreInsufficient
    TooManyUpstreamNpSeqsOnController
  PostedWriteIsAnIssue
  BusFunctionalModelStuckInL1

Figure 3: Example of a PCI Express Debug Tree

This tree aims at diagnosing why the PCI Express tests did not complete, so the
root node is labeled PcieTestDidNotFinish and given a tab depth of zero. The de-
developer identified outstanding end of test traffic and test timeout as the two common
reasons why PCI Express failures occur, so they created PcieHasTrafficNearEOT
and PcieIsHung as the two children of PcieTestDidNotFinish with tab depths of
one. Note that children are always at a tab depth of one greater than that of their
parents.

4.4 VASE Python Code

When a VASE developer has completed a text file definition for their VASE tree,
they run the VASE tree code generator to create Python template code, in which
they can embed the logic required for diagnosing the simulation from the test run
results. The tree file and the Python collateral are automatically organized into
a single folder within the local clone of the VASE repo. Once the developer has
completely implemented and debugged the decision tree, it can be checked into the
remote VASE repo for other users to access. An example of a class definition for
decision tree node is shown in Figure 4.

The example shown in Figure 4 defines the PcieIsHung node from Figure 3.
class PcieIsHung (DecisionTreeNode):
    def nextStepsIfTrue(self):
        return [LongUpstreamIsAnIssue, PostedWriteIsAnIssue, BusFunctionalModelStuckInLL0]
    def isTrue(self):
        didPcieHang = False
        ...
        return didPcieHang

Figure 4: Example of Python class definition

Since all the Python collateral is located in the same folder in the VASE directory structure, the class definitions for the children nodes can be automatically imported at the top of the file. The two main components of a node’s class definition are the isTrue and nextStepsIfTrue methods. The isTrue method evaluates to true if the identifying conditions for this node have been met. In this case, isTrue returns true if the PCI express test has timed out. The nextStepsIfTrue method returns pointers to the class definitions of the children nodes, therefore encoding the structure of the tree. VASE automatically populates the nextStepsIfTrue method from the information in the decision tree text file. During VASE tree runtime, if isTrue returns true, VASE will generate instances of the node class definitions returned by nextStepsIfTrue and recurse over them in a breadth first search manner. If isTrue returns false, then the current node’s subtree will not be explored, since the conditions for that node were not met.

4.5 Shortcomings of the current VASE tool

This section describes the main shortcomings of the current VASE tool implementation with text-based user input.
4.5.1 Node Subtree Coherence

In many cases, a developer may want to re-use a node in another part of the tree or in another tree altogether. Node names are unique, so if two nodes share a name then they must be identical. For instance, in the tree shown in Figure 3, LinkWidthIsLimitingFlow is present as a subnode of both PcieHasTrafficNearEOT and LongUpstreamIsAnIssue. In other words, it is possible that LinkWidthIsLimitingFlow is true if either PcieHasTrafficNearEOT or LongUpstreamIsAnIssue is true. Since the tree structure is maintained by having each node’s Python class definition point to the class definitions of its children, and since multiple occurrences of the same node will merely be separate instances of the same Python class, VASE decision trees require that the all occurrences of the same node have identical subtrees. In other words, every instance of a given node must have the same children. This also disallows any node from being in its own subtree, since this would create a cycle of infinite execution. Therefore when the developer defines the tree in the input text file, they must manually ensure that all modifications to a node’s subtree are reflected in the subtrees of any other instance of that node. If they fail to do so, the tree will fail in any execution that reaches the incoherent nodes.

4.5.2 Scalability and Formatting Issues

If VASE is to be adapted into the Intel workflow for System on Chips of enormous size, the decision trees that diagnose those systems will be enormous as well. System level experts could leverage and aggregate trees from the subsystem experts, providing significant productivity boosts during system integration. However, as trees grow in depth and number of nodes, the text file tree definitions will become cumbersome and difficult to maintain. For example, inserting a subtree will require scrutinizing the appropriate number of tab characters for each node. Each instance of a node will have to be manually checked for coherence. Furthermore, simply typing a text file of the necessary size will become cumbersome. A simpler, more intuitive, and scalable method of input is necessary if the VASE tool is to be employed.
on large-scale systems.

5 Design of the VASE GUI

The general goal of the VASE GUI is to increase the rate of adoption of the VASE tool at Intel, and to improve workflow productivity of those already making use of VASE. According to [4], the likelihood of user acceptance of new technologies is strongly influenced by two factors: the perceived ease of use of the technology, and its perceived usefulness. In this case, the users we are speaking of are Intel hardware validation engineers. It's important to make a distinction here between the VASE tool and the VASE GUI itself. The VASE GUI aims to increase adoption of the VASE tool, but there is also the question of the adoption of the VASE GUI itself, in other words, whether engineers who have already integrated the text-based VASE tool into their workflow will see the benefits of the GUI. Therefore, the VASE GUI must accomplish two things: first, if an employee is already making use of the VASE tool, they should adopt the VASE GUI into their workflow; and second, a first time user should be more likely to adopt VASE if they make use of the GUI rather than the text-based input. In order to achieve this, four overarching design goals, described below, were identified.

5.1 Ease of use of the GUI itself

One of the main goals of the VASE GUI is to increase the perceived ease of use of the overall VASE tool, so that engineers, upon their first attempt at using the tool, will be more likely to adopt it into their workflow. Therefore the perceived ease of use of the GUI itself should be high, and should in particular be much higher than that of the current text-based system. In this context, ease of use is defined as follows: The GUI should be easy to navigate and understand; it should not cause unexpected behavior; and the barrier to understanding the interface should be low. In general, the interface is meant to be usable without requiring the user to undergo
any additional training or tutorials. Therefore, a user should be able, on a first use of the GUI, to navigate appropriately to what they wish to accomplish with minimal or no aid from an outside source. The interface, of course, should accomplish what the user instructs it to do without unexpected or undefined behavior; if any errors occur they should be reported to the user in an easy-to-understand format.

5.2 Ease of installation

Even if the GUI is designed to be simple to learn and use, it will still not be adopted if engineers do not try it out in the first place. Therefore, the VASE GUI must be designed in such a way that the barrier to installation and first-time use is low. First of all, installation should be simple and ideally packaged with the current VASE tool so that engineers do not need to run two separate installations. Second, all dependencies should be packaged together and included, again so that engineers do not have to run multiple installations. Finally, the VASE GUI must be compatible with all mainstream computing environments.

5.3 Perceived benefits of use

The perceived benefits of using the VASE GUI rather than the original VASE tool with text-based input is arguably the most important metric in ensuring that the VASE GUI is adopted by engineers already making use of the VASE tool. It is not enough that the GUI is easy to learn and make use of; previous studies have shown that perceived usefulness is a stronger indicator of user satisfaction that learnability alone [5]. Considering that the target user in this case is an Intel engineer, someone who is likely accustomed to adopting new technologies and ideas into their workflow, it seems even more likely that the perceived benefits of the GUI will be the largest determiner of its adoption. Note that the GUI will most likely not change the perceived usefulness of VASE itself, since it merely provides a new interface for the tool and does not change the function of the tool itself. The goal, then, is to ensure that the perceived benefits of the GUI are strong enough that engineers will adopt
it into their workflow and so benefit from a simpler, less error prone interface to the VASE tool itself.

5.4 Ease of understanding and modification of the code itself

A fourth consideration is the ease of understanding and modification of the GUI's implementation. Although not directly related to user experience, it is imperative for the GUI's long-term adoption at Intel. Previous work has identified three main indicators of the maintainability of a piece of software: the readability of the source code, the documentation associated with the software, and the coherence between the software documentation and source code itself [6]. Since source code that is highly readable tends to also be well-functioning [7] and easy to modify, the main focus for the VASE GUI will be to ensure that all source code is simple to read, understand, and ultimately modify. Another argument for the focus on source code readability is the need to maintain coherence between the software documentation and source code itself; when code is modified, as the VASE GUI code most certainly will need to be if it is to be maintained at Intel, documentation must be modified accordingly in order to maintain coherence. However, if multiple people are modifying the same piece of software, it is likely that source code-documentation coherence will not always be maintained. In these cases, the source code must be the first authority for the behavior of the software; if it is easily readable, there is no need for supporting documentation.

6 Implementation

6.1 Integration with current VASE tool

The VASE GUI was designed as a simple, lightweight, desktop application that engineers can seamlessly integrate into their VASE tree development workflow. The VASE GUI will reside alongside the VASE tool and the decision tree database in the centralized VASE repo. When a VASE 'user' wishes to invoke a decision tree
on their simulation run results, they simply open a terminal in the directory of the run results and run a VASE command. Such users will not have to deal with the VASE GUI or the VASE repo at all. However, when a VASE 'developer' wishes to create a new tree or modify an existing tree, they must first create a local copy of the VASE repo. Since the GUI will be housed in the repo, this will automatically give them access to the GUI alongside the VASE tool and the VASE trees. From the root directory of the local repo, they can then invoke a terminal command to run the VASE GUI. From within the GUI, they can start a new tree or modify any existing tree stored in the central repo (described in more detail in Section 6). Once the developer has completed their changes to the given tree, they use the VASE GUI to submit the tree to the local copy of the VASE tool tree code generator, which will generate the Python classes in the local repo as well. Once the Python classes have been filled out and debugged, the developer can then push their changes to the centralized repo. Before accepting changes to the VASE repo, the Intel version control tools run every tree in the repo to validate that they run to completion to ensure that malfunctioning trees do not interrupt the engineering workflow.

6.2 Graphical User Interface implementation tools

The VASE GUI is designed for seamless integration into the VASE workflow and for compatibility with a variety of system configurations. Since the VASE tool is implemented in Python, the VASE GUI was also implemented in Python and is compatible with all Python 2 and Python 3 versions. It has been tested and shown to be compatible on Windows and Linux platforms. Additionally, the only Python packages that were used in both VASE and the VASE GUI are those that are included in the Python standard library. Among the standard libraries is Tkinter: Python's standard graphical user interface development package. Tkinter is shipped with all versions of Python, and the minor differences between Python 2 and 3 versions can be accounted for by querying the system for the user's Python version at the start of the VASE GUI execution. Tkinter provides an object oriented framework with
all the GUI primitives necessary for the development of a scalable, easy to use user
interface.

Tkinter has a methodology that differs from that of standard Python [8]. While a
standard Python application typically involves a linear program flow from the first to
the last line of code, after which the program terminates, Tkinter provides an event
driven framework. A Tkinter application execution starts with the instantiation
and integration of all the GUI components, but eventually quiesces into an event
processing loop. Further program activity only occurs in event handlers that are
triggered when the user interacts with GUI components such as buttons, text fields,
and drop down menus. Execution is complete when the event processing loop is
terminated in software, usually as a result of an event handler associated with a GUI
component, such as an exit button, for closing the application. In short, Tkinter
requires asynchronous software development, whereas standard Python traditionally
consists of synchronous programming practices.

The Tkinter framework provides a variety of core GUI primitives—such as en-
try fields, buttons, and menu components—called widgets. Tkinter also supports
user-defined widgets, so that developers can achieve a wide range of sophisticated
interaction by creating complex widgets that integrate and augment the function-
ality of the core widgets. An entire Tkinter application can then be seen as a tree
hierarchy of widgets with the main application as the root of the tree. A parent
widget instantiates, integrates, and positions its child widgets at any point during
execution. For example, Tkinter has a treeview widget that can display a tree
structure to the user. However, the treeview widget only supports tree structure
manipulation via method calls on the object, and does not support binding event
handlers to clicking on a specific node. To allow the VASE GUI user to add or
delete nodes from the tree, I created an augmented treeview widget that incorpo-
rated the core treeview, right click menus, text entry fields, and custom node click
event detection to achieve the desired functionality.
6.3 The Model-View-Controller methodology

A major design motivation for the VASE GUI, as identified in Section 4, is that the code base be highly readable so that it is simple for those who inherit it to understand and modify. To achieve this, the VASE GUI was implemented using the model-view-controller software framework, shown in Figure 5. The goal of the model-view-controller paradigm is to promote a clear separation of concerns between the information, the way information is displayed to the user, and the link between the two [10]. This section describes the functionality of these three components—respectively called the model, view, and controller. The VASE GUI was implemented with this model in mind, and so maintains software compartmentalization between the model, view, and controller components.

6.3.1 Model

The model holds information that represents the fundamental state of the system as perceived by the user of the software. It should be able to respond to both requests for information and commands to alter the information it holds. A properly implemented model should not make any assumptions about the system’s view or controller. It should also enforce any rules that might apply to the dataset it holds.
In large scale systems, the 'model' is oftentimes a database or some other information storage system. In the case of the VASE GUI, the state of the system is encoded in the current decision tree. The model therefore provides the following functionality to the user: a representation of the entire tree, as well as the ability to insert a node under a parent, remove a node, and check if a new node's name is a valid Python identifier. It enforces the fact that no node may be a descendant of itself, and that subtree coherence must be maintained between nodes. When a user opens an existing tree, that tree's structure is extracted from the tree executable and imported into the GUI model. After that, any tree modifications will be reflected only in the GUI model; in other words, only data structures internal to the GUI will be modified. Only once the user has explicitly finalized the tree by publishing it are their changes reflected in the Python executable code for that tree.

6.3.2 View

The view serves as the visual representation of the model. It allows the software system to highlight and suppress certain aspects of the model, acting as a presentation filter for the data in the model [10]. A proper view should not make any assumptions about the model or the controller. The view typically encapsulates all of the user facing components of the application. Multiple views are possible for a fixed model and controller, such as a pie chart vs a bar graph for the same data. In a Tkinter application, the core widgets primarily serve as the backbone for the view. Along with displaying information to the user, Tkinter and its core widgets also provide the crucial starting point of the user interaction: event detection. After detecting an event, however, the event handler routines are located in the controller, as discussed below. Therefore the user interface components of the VASE GUI, such as buttons and text boxes are encapsulated within the model, while the event handlers for these components are encapsulated within the controller.
6.3.3 Controller

The controller is the link between the view and the model. Unlike the view and the model, the controller can and should make every assumption necessary about the other system components. The controller is responsible for processing user input, once the detection of that input has been handled by the view. Once the controller has determined what system response to carry out based on the input, it updates the view and the model to reflect the changes in the system state. In the VASE GUI, the tree controller adds and removes nodes from the view and the model. It also carries out complex operations, like moving an entire subtree and maintaining node subtree coherence, by making multiple add and remove node requests to the model and view. Finally, the controller confirms the validity of all requested user actions, such as ensuring names adhere to conventions and requests do not lead to an invalid tree structure. In the case of an invalid request, the controller instructs the view to deliver error messages to the user.

7 Results

Two versions of the VASE graphical user interface were created, in order to provide compatibility with both the original VASE tool and VASE II, an updated version that was in development at the time of writing. Although both versions provide the same general functionality, their implementation and exact capabilities vary. Since it was created first, the VASE I GUI provides a richer range of functionality. This section provides a walk-through of the VASE I GUI and a discussion of how it addresses the shortcomings of VASE's original text-based input. The VASE II GUI, as well as possible functionality extensions, are described in Section 7.

7.1 Installation

The VASE GUI and all of its dependencies will be included in the VASE repo. When an engineer creates an instance of the VASE repo on their machine, they
automatically have access to the GUI. All they need to do is open a terminal and run the GUI as a Python executable. Therefore, it will be readily available and accessible to all engineers currently making use of the VASE tool; for new users, the overhead to install and get started on the VASE GUI will be no greater than that for the original implementation with text-based input.

7.2 Overview of the GUI

![Overview of VASE GUI](image)

Figure 6: Overview of VASE GUI

The VASE GUI, shown in Figure 6, has two main components: the toolbar and the workspace. From the toolbar, the user can create a new tree, open an existing tree, and publish their current tree. They can also expand or collapse all of the subtrees of the currently selected tree. The workspace provides the user with a simple, graphical interface for manipulating trees (described in more detail
There are two main components to the workspace: the VASE Tree space, which displays the currently selected tree, and the Unused Nodes space, which displays all unused nodes for the currently selected tree. Both the VASE Tree and Unused Nodes spaces provide the following information for each node:

1. Tree: The name of the subtree, as well as its location in the overall tree hierarchy
2. Last Edited By: The name and email address of the last person to edit the subtree
3. Last Edited On: The date and time the subtree was last implemented
4. IsTrue Implemented: Indicates whether or not the true-false condition for the subtree has been implemented. Will default to 'no' for new trees.

### 7.3 Creating a new tree

![Image of creating a new tree](image)

Figure 7: Creating a new tree

To create a new tree, the user opens the VASE GUI main page and selects File > New tree. A dialogue box, shown in Figure 7, will open and will prompt the user to select a project for their new tree. The VASE GUI will automatically connect to the VASE repo and will display all existing projects in the repo. The
user can either navigate to their desired project, or can type the name of the project in the dialogue box. They can either choose an existing project, or can create a new project by naming a project that doesn’t currently exist. The user then names the tree, selects ‘New tree’, and the tree will automatically be created and brought up in the GUI (Figure 8). When a new tree is created (Figure 8), it does not contain any subtrees or unused nodes. By right-clicking on the top level tree, the user will be able to add subnodes and begin construction of their tree (described in section 6.4)

7.4 Opening an existing tree

To open an existing tree, the user opens the VASE GUI main page and selects File > Open tree. A dialogue box will open (Figure 9) and will prompt the user to select an existing tree from the VASE repo, organized by project. The user double-clicks on their tree of choice, and the tree will automatically be brought up in the GUI.

Figure 8: Newly created tree is opened in GUI
7.5 Modifying a tree

The VASE workspace provides a simple graphical interface for the user to add, remove, and reorganize the subnodes of their tree.

7.5.1 Adding and removing subnodes

To add a subnode, the user right clicks on the node that will be the parent for the new subnode and chooses ‘Add subnode’ from the dropdown that appears (Figure 10). They then provide a name for the new subnode. The GUI will automatically check if the provided name is a valid Python identifier, and will throw an error if not, since VASE will elaborate the tree into Python code when it is published. The newly created node, shown in Figure 11, will be the child of the selected parent node.
To remove a node, the user simply right clicks on the node they wish to remove, and chooses ‘Remove Node’ from the dropdown that appears (Figure 10).

7.5.2 Rearranging nodes

Existing nodes in the tree can be rearranged to modify the tree structure. To do this, the user clicks on the node they wish to rearrange and drags and drops it on the desired new parent node. The node will now be a child of this new parent, and not of their previous parent.

7.5.3 Publishing a tree

Once the user is done modifying their tree, they must publish it. To publish their tree from the VASE GUI, the user navigates to File > Publish Tree. The VASE GUI takes the graphically specified tree and creates a text-based representation of it. This text-based representation is of the same format as that originally required by VASE. It is provided to the VASE tree generated in order to create the corresponding Python template files for each node.

In many cases, the user may publish a tree for which Python node definition files already exist. In these cases, the VASE tree generator does not overwrite the existing files, but merely updates their children to reflect the new tree structure.

7.5.4 Unused nodes

When a developer opens an already existing tree, the GUI must scan that GUI’s Python files in order to infer the tree’s structure. In some cases, files may exist
for nodes that were originally defined, but later deleted, from the tree. This occurs when a user publishes a tree that includes a node X, then later opens the tree, deletes node X, and republishes it. The Python node definition file for X will still exist, but X will not be the child of any other node. When reading in an existing tree, the VASE GUI will identify those nodes that exist but do not have any parents. They will be included in the unused nodes section of the VASE GUI workspace, shown in Figure 6.

The user may simply ignore those unused nodes, or may choose to re-include them in the tree. To re-include a node in the tree, the user makes use of the same drag-and-drop mechanism described in Section 6.5.2, clicking on an unused node and dropping it on top of the desired parent.

When a tree is published, the files for its unused nodes are not deleted. This is useful for the case when an engineer believes that a node or subtree might be useful at some point, but is not sure exactly where to include them. The unused nodes section effectively provides a space for saving nodes for later, so that the work put in to defining the behavior of a node is not lost if that node is deleted. As a result, the only way to permanently delete a node from a tree once it has been published is to delete the generated Python file for that node.

7.5.5 Correctness checks

The VASE GUI provides two main guarantees in order to ensure that the generated tree does not raise errors upon execution. First, no node may be a descendant of itself. This ensures that the VASE tool does not hang in an infinite loop, and also reflects the fact that cycles make no logical sense in a decision tree. Second, the GUI ensures that subtree coherence is maintained between multiple instances of the same node. Once again, this ensures that no errors occur upon execution of the defined tree by the VASE tool. If the user attempts to violate either of these guarantees, an error message will be shown and they will be prevented from completing the action. This allows the user to immediately take action to either rethink their tree,
if the invalid input was intentional, or—in the case of an accidental click—perform the action they originally mean to perform.

7.6 Evaluation

This section evaluates how the VASE GUI addresses the issues identified in Section 3.5 and provides a more user-friendly experience for the engineer making use of VASE.

7.6.1 Usability of interface

The VASE GUI is designed to be simple and intuitive to use. All of the functionality is accessible from a single main page; additional messages and selections, such as that shown in Figure 10, are included as small pop-up boxes that overlay the GUI’s main page. From the main page, most actions can be accomplished in 4 or fewer clicks.

The workspace is designed to make modifying trees simple and intuitive. Rearranging of the nodes is accomplished using the familiar drag-and-drop mechanism. Right-clicking on a node, another familiar mechanism, brings up extra options for that node. To reduce clutter, all subtrees can be expanded or collapsed, so that the user can focus on what they are currently modifying.

7.6.2 Scalability

The VASE GUI provides a tree manipulation experience that scales with the complexity of the tree. The developer will have the ability to collapse and expand branches of the tree in order to properly focus on the relevant portion of the tree at any given time. When adding a node, VASE will automatically handle proper tree formatting and node subtree coherence. The developer will be able to drag branches with the mouse to relocate them rather than reformattting a text file.
7.6.3 Fault Tolerance

The VASE GUI provides a less error-prone interface than the original method of text-based input. In the text-based version, heredity was indicated by tab depth, making errors easy to commit and difficult to detect. In the GUI, the user must click on the node they would like to add a child to; therefore heredity is explicit. The only way for the user to create an erroneous tree structure is if they add a node as a child to the wrong parent. This is in general a much more difficult error to commit than a simple typo.

The GUI also ensures that errors that might cause the VASE tree generator to fail—a recursive subtree, or multiple instances of a node with non-coherent subtrees—are prevented. While it is still possible for a user to accidentally misidentify the parent of a node, they will at least be prevented from doing so in cases that would cause errors.

7.6.4 Backwards compatibility

During the publication step, the VASE GUI automatically generates a text-file of the same format as the original VASE tool’s text-based input, which is then provided to the VASE tool generator. The interface layer is therefore completely separate from VASE tool itself. This has two major advantages. First, there is no need to modify the VASE tool in order to interface with the GUI. Second, the user who is accustomed to the text-based input still has the option to review or edit their work using this generated file. Using the GUI to generate the text file, however, provides correctness guarantees that no longer hold if the user modifies it by hand.

8 Next steps

8.1 VASE II

During the implementation of the VASE GUI, a new version of VASE, VASE II, was being developed. The VASE GUI was modified to create a VASE II compatible
version, maintaining most of the same functionality as the original GUI. Since VASE II was still being developed and had not yet been released at the time of writing, some implementation details had yet to be finalized. This section highlights the differences between VASE I and VASE II, as well as their implications for the GUI.

8.1.1 Coherence between nodes no longer necessary

VASE II does not require that coherence be maintained between the subtrees of two instances of the same node. This essentially allows the developer to create multiple instances of a node with the same `isTrue` method but different children. As a result, the GUI no longer needs to check for coherence between nodes since un-coherent nodes will no longer cause errors when the VASE tree builder is run.

Note that this means that a node may now be a child of itself. However, since multiple instances of the same node no longer necessarily have the same subtree, recursion does not occur.

8.1.2 No central repository

VASE II removes the need for a central repository. In the original implementation, the repository contained both the VASE itself and a directory structure for the storage and retrieval of tree definitions. VASE II is packaged as a Python package, which developers can simply import in order to make use of. The VASE II GUI will be included in this package. Once a developer has imported the package, they will be able to run a command in order to invoke an instance of the GUI. As with the VASE I GUI, this should ensure that the overhead to developing with the GUI is no higher than the overhead to developing without it.

Since trees are no longer stored in the central repository, they must be kept locally. As a result, when a user opens a tree file in the VASE II GUI, rather than choosing from a list of trees stored in the VASE repo, they are prompted to select the directory from their own file system that contains the corresponding tree definition. In order to share trees, developers are responsible for sending that tree to those who
may need it, for instance by packaging the tree with the system it describes.

8.2 Possible functionality extensions

This section describes possible functionality extensions that could be applied to either the VASE I or VASE II GUI.

8.2.1 Tree annotation

It would be beneficial if the VASE GUI provided some method of annotating tree nodes. Analogous to commenting code, this would increase the readability of the tree by providing quick descriptions of what each node describes. Although in the current implementation comments can be included in the Python class definitions for each node, this does not help users understand the overall tree structure; a person wondering about the functionality of a tree would have to actually open and read each individual Python file in order to understand the function of each node in it. Allowing annotations would provide readability at a glance.

8.2.2 Modifying node definitions directly from the GUI

Another useful extension would be to allow the user to modify the Python class definition for a node directly from the GUI. This could be accomplished by providing an 'edit' option for each node that automatically opens its Python class definition in a text editor. Doing this would prevent the user from having to search for the appropriate file in their file system.

8.2.3 Dependency management for Python template classes

Results of a simulation run include not only log files but more complex information such as digital waveforms. Therefore, many isTrue methods might involve complex operations or data processing to parse this information. As a result, developers will often want to include either third party packages or their own libraries in their Python node definitions. The VASE GUI could provide users with a method
of specifying which packages they want included in each of the generated Python templates. The GUI would then automatically include them in all of the resulting templates, saving the user from doing so manually.

9 Conclusion

VASE has the potential to greatly improve hardware verification efficiency at Intel. However, teams won’t choose to adopt VASE into their workflow if the tool is not easy to use. I implemented a graphical user interface with the goals of both increasing usability of VASE to increase its rate of adoption, and providing existing users with a more intuitive, and less error prone, interface. The implementation prioritizes both the usability of the interface itself and the readability and ease of modification of the code base.

10 Acknowledgements

I’d like to thank Beth Markey, Reinhard Zippelius, and Erik Berg for the all of the guidance during my time at Intel. I’d also like to thank Professor Arvind Mithal, my thesis advisor at MIT, for the supporting my research and my thesis writing. Lastly, I’d like to thank my family – Nancy, Miguel Sr., Paola, Andres, and Jen – for the support, encouragement, and motivation you give me every day.

References


