Synchronizing Trees and Text in a Software Model Analyzer

by

Jesse Pavel

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 2002
[June 2002]

Author .....
Department of Electrical Engineering and Computer Science
May 24, 2002

Certified by .....
Daniel Jackson
Associate Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Synchronizing Trees and Text in a Software Model Analyzer

by

Jesse Pavel

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2002, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis describes the work that I did in improving the debugging features and interface of the Alloy software analysis tool. A significant part of writing a model in the Alloy language is debugging it, and often one wants to know why a particular solution adheres to or violates given constraints. To this end, the interface provides a view of an Abstract Syntax Tree that shows the tool's internal representation of a model, and my enhancements aim to provide aids to the user in finding interesting parts of this tree, and showing the corresponding sections of the source text. Additionally, I provide a description of the architecture of the graphic interface I implemented, and upon which these improvements were built.

Thesis Supervisor: Daniel Jackson
Title: Associate Professor

Acknowledgments

I would like to thank Daniel Jackson for supervising my work, Manu Sridharan and Ilya Shlyakhter for being patient with and helping out this most junior SDG developer, Jenny Liao for being beside me throughout the year, and my parents, for always supporting me, and for the lychees.

Contents

List of Figures

Chapter 1

Introduction

Alloy [1] is a language and tool that allows one to write a micromodel of a system and have that model be algorithmically analyzed, to find flaws or unexpected behavior, or simply to understand the system in a more rigorous manner. A model in Alloy is a declarative description of part of a system comprising, at the lowest level, sets of atoms, the relations among the sets, and the constraints placed on these. Given such a description, the user asks the tool to check assertions that he feels should follow from the constraints—and if the assertions do not hold, to generate a counterexample—or to generate an example instance of the system.

The tool works by converting the model into an abstract syntax tree (AST), and then a large boolean formula which is passed to a SAT-solver, the result of which is then translated back into the form of a solution [3]. After a model is analyzed, the user can browse the AST to see what clauses were generated, and how these contributed to the overall truth value of the formula.

An instance of a solution is an assignment of values to the variables used in a description, and can be either generated automatically by the tool when it runs a command, or given explicitly by the user as a candidate to be analyzed. Sometimes a solution cannot be found, and the user must scrutinize her model and the tool's output to determine the cause. Also, the tool has a facility that allows the user to enter a candidate instance, to determine to what degree it satisfies the constraints of the model. In both of these cases, it is likely that the AST contains unsatisfied clauses which may be of interest to the user, and this work focuses on helping the user find these clauses, and also the text to which they correspond.

There were two main challenges that I faced in this project: first, determining which usability features would most aid users in navigating and debugging models—and implementing them; and second, designing a stable and effective graphic user interface upon which such features could be built, and which is modular enough to be easily amenable to including addition features in the future. The construction of such a GUI was a significant part of my work, and a chapter is included which details the architecture of the GUI package and presents a guide to the code, designed to help future maintainers understand and modify it.

Format

I first present to the reader the issues hindering usability in the original version of the tool, and then show my solutions to some of these problems. To illustrate how a user would use the tool to debug his work, I give an example model and scenario for debuggin a particular problem. Finally, I include a description of the GUI architecture.

Chapter 2

Motivation

In earlier versions of the tool, the abstract syntax tree (AST) (of which an example is given in figure 2-1) was presented with little adornment and no automated tools to find unsatisfied clauses. A user had to manually click through the tree and examine a separate value tree, which was itself embedded within the AST, to determine if a clause was false. The tree was generated in full, and one was hindered in debugging by the many levels through which he had to burrow, and by long, hard-to-read expression names.

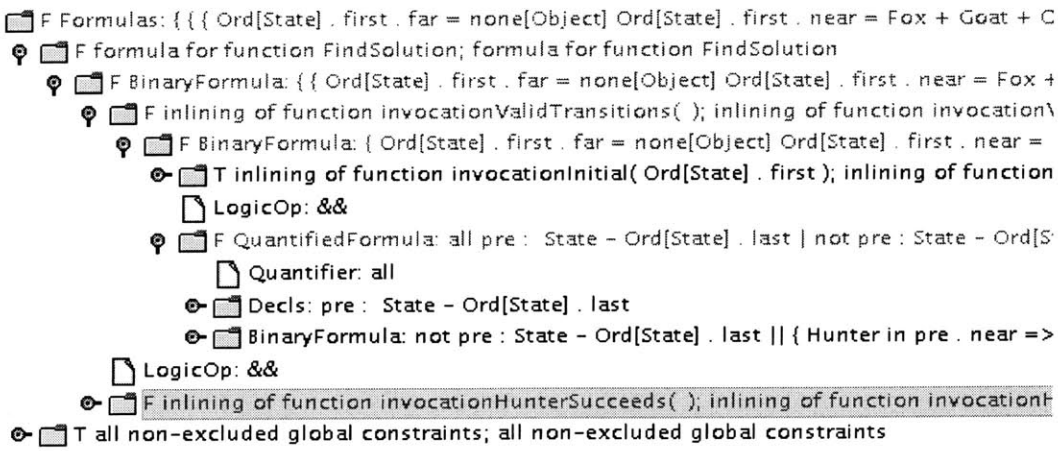

Figure 2-1: Sample Alloy AST

The AST at the top level is grouped into two major sections: a section that corresponds directly to the formula translation of the model, for which the tool had already had synchronization capabilities, and a section that holds global constraints. Global constraints arise from several syntantic features of the language; for example, marking a collection of

signatures with the `disj` keyword results in a constraint that the sets of atoms in those signatures are mutually exclusive. For constraints such as that, it is unclear which, if any, section of the model derives it. There are some clauses in the global constraints, however, which strongly indicate a location in the source, and while there may not be a direct correspondence, it would be well for the tool to be able to synchronize those clauses.

Chapter 3

Solution

This chapter presents various enhancements that are aimed at mitigating some of the difficulties presented earlier. They deal primarily with navigating the abstract syntax tree (AST), but also include some changes to the Alloy GUI.

3.1 Navigation Enhancements

3.1.1 Tunneling in the AST

The capability to have the tool automatically 'tunnel' to, that is, expand the AST to show, interesting nodes alleviates much of the frustration at having to click through dozens of branches to find the source of a false clause. The nodes in which we are interested we call *pivot* nodes, and they are the ones upon whose values the truth value of the formula as a depends. Pivot nodes are themselves atomic formulas that do not contain complex formulas as children.

Determining whether a node is a pivot depends on the context in which it appears: whether it is examined under a negation operator, if it is part of a conjuntion or disjunction, and if it is part of a universally or existentially quantified formula. Below, we examine each of these cases in more detail.

We are always in a boolean context when we examine a formula: whether we are trying to determine why it is true, or why it is false. I present an algorithm for finding pivot nodes of a formula, organized by the boolean context, and the logic operator of the formula.

Let F be the formula the pivot nodes of whose truth value we are searching for, and

let A and B be the subformulas that constitute F, under different operators. The function *pivots(F,b)* returns the pivot nodes of F that cause it to have the value b.

Formula Syntax	Result of *pivots*(F,t)
$F = \neg A$	pivots (A, f)
$F = A \wedge B$	pivots (A, t) $\cup$ pivots (B, t)
$F = A \vee B$	(if eval(A) = t, then pivots(A, t)) $\cup$ (if eval(B) = t, then pivots(B, t))

Formula Syntax	Result of *pivots*(F,f)
$F = \neg A$	pivots (A, t)
$F = A \wedge B$	(if eval(A) = f, then pivots(A, f)) $\cup$ (if eval(B) = f, then pivots(B, f))
$F = A \vee B$	pivots (A, f) $\cup$ pivots (B, f)

By this algorithm, the pivot nodes can be intuitively thought of as the ones which are responsible for the truth value of the formula; by toggling the value of all the pivot nodes, the value of the formula will be toggled. Changing any of the nodes outside the set of pivot nodes will not have any effect on the value of the formula.

The universal and existential quantifiers can be regarded as sugared forms of conjunctions and disjunctions, respectively. A quantified formula of the form '$\forall x : S \mid F$' can be desugared into a conjunction formula '$F(x_0) \wedge F(x_1) \ldots \wedge F(x_n)$', and likewise the formula '$\exists x : S \mid F$' desugars into '$F(x_0) \vee F(x_1) \ldots \vee F(x_n)$'; and then the resultant formulas can be analyzed as above.

In figure 3-1 we give an example AST with values, and one can see how the pivot nodes E and C propagate their values up the tree.

To activate tunneling in the interface, the user can pop up a contextual menu from any AST node and choose to tunnel to pivot nodes, the starting context of the operation determined by the truth value of the top-level node in the search.

3.1.2 Bookmarking Nodes

If there are many nodes throughout the AST between which the user wants to jump, he used to have to tediously open and close branches, or scroll through a large tree: bookmarking saves time by letting him save nodes to a list; then, by selecting a node from the list, the tree expands and scrolls as necessary to display it.

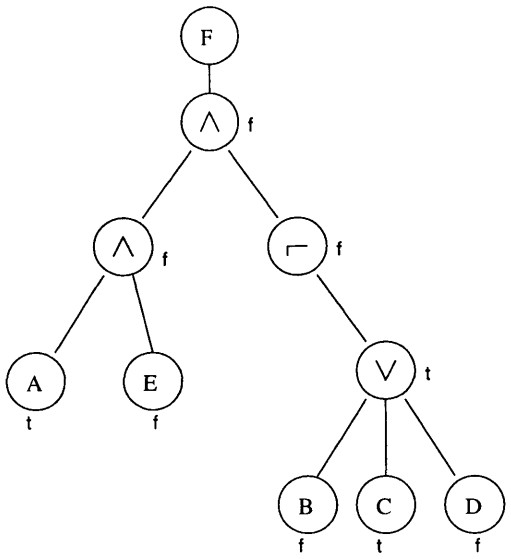

Figure 3-1: Sample Boolean AST

3.1.3 Source Synchronization

Many of the global constraints now synchronize, thanks to Manu Sridharan, to sections of the source which seem reasonable in context, such as the signature definition for a clause that reports that a `static` (singleton) signature contains multiple atoms.

3.2 Usability Improvements

3.2.1 Name Shortening

Previously, the AST would display all signature expressions, variable names, and skolem constants using fully qualified names; for instance, if one had a model in the `puzzles/hunter` module which contained the signatures `State` and `Object`, they would be displayed as `puzzles/hunter/State` and `puzzles/hunter/Object`. Having many names with identical prefixes made the tree hard to read, and made expression lines longer than necessary. Our name shortening procedures first sort through all of the names in a solution, finding those that can be shortened without creating name collisions, and then when the AST is displayed, expressions are parsed to replace qualified names with their shortened equivalents.

3.2.2 Quantified Variable Tree

For AST nodes whose value depends upon one or more quantified variables, the tool generates a tree that shows the truth value of the node for each setting of variables. Formerly, all of these trees were generated before the AST was displayed, and were embedded in the AST at the level of the node whose value they represented. It was hard to distinguish at first glance these quantified variable trees for other, normal child nodes, and it took time to generate all the trees even if most were never viewed. We changed the presentation so that the variable trees appeared in a separate pane from the AST, and were generated and cached only when a user viewed the corresponding node.

Chapter 4

Scenario

Here I present an example model, albeit one with a small bug that prevents the desired solution from being found, and the steps that a user might take using the debugging features to find the section of the model that needs to be fixed.

4.1 Model Description

The model is designed to generate a solution to the puzzle:

> A hunter is one one shore of a river, and has with him a fox, a goat, and cabbages. He has a boat that fits one object besides the hunter himself. In the presense of the hunter nobody eats anything, but if left without the hunter, the fox will eat the goat, and the goat will eat the cabbages. How can the hunter get all three possessions across the river safely?

The solution to this model will be a sequence of states that represent the hunter's trips across the river. We specify constraints on how the hunter can carry things on the boat, and what things he cannot leave on the shore together unattended; the solver will then figure out a sequence of states (trips) that will lead to him having transferred all of his belongings to the other side safely.

4.2 Example Model

```
1. /*
    Model by Jesse Pavel <jpavel@mit.edu>
   */
```

```
module hunter

open std/ord

// The hunter and all his possessions will be represented as Objects.
sig Object {}

// The static keyword specifies that each subsignature will
// have only one member, and disj forces each of these
// to be disjoint from the others. Thus, we create four
// unique objects here.
static disj sig Hunter extends Object {}
static disj sig Fox extends Object {}
static disj sig Goat extends Object {}
static disj sig Cabbages extends Object {}

// Each state represents one trip by the hunter across the river,
// and the near and far relations contain the objects held on each
// side of the river, respectively.
sig State {
    near: set Object,
    far: set Object
}

// In the initial state, all objects are on the near side.
fun Initial (s: State) {
    s.far = none[Object]
    s.near = Fox + Goat + Cabbages + Hunter
}

// TransferSomething() constrains the movement of objects from a
// pre-state (from, to) to its post-state (from', to').
fun TransferSomething (from, to, from', to': set Object) {
   // There are three ways in which the Hunter can do things.

   // The hunter can take exactly one object with him across the river.
   (one item: from - Hunter |
      (to' = to + Hunter + item) &&
      (from' = from - Hunter - item))
   ||
   // Or he can just wait where he is.
   (to' = to  &&  from' = from)
}

// For each pair of subsequent states, the hunter can move
// something only from the side he is currently on.
fun MoveSomething (s, s': State) {
    Hunter in s.near => {
       TransferSomething (s.near, s.far, s'.near, s'.far)
    }
    else {
       TransferSomething (s.far, s.near, s'.far, s'.near)
    }
}

// This function represents the constraints posed
// by the problem itself, that the hunter cannot leave
// the fox and goat by themselves, or the goat and cabbages.
fun NothingGetsEaten (objs: set Object) {
    (Hunter !in objs) => {
   ((Fox + Goat) !in objs) &&
        ((Goat + Cabbages) !in objs)
    } else {}
}

// We must constrain our simulation so that objects do
```

```
    // not spontaneously duplicate or disappear from the world.
    fun OneOfEverything (s: State) {
       all o: Object |
          (o in (s.near + s.far)) &&
          (o in s.near => o !in s.far)
    }


80. // Here we tie things together:
    // The initial conditions must hold, and then for all
    // pairs of subsequent states, the hunter can move something,
    // making sure that nothing gets eaten on either shore, and
    // that nothing violates a law of existence.
    fun ValidTransitions () {
        Initial (Ord[State].first) &&
        (
        all pre: State - Ord[State].last | let post = OrdNext(pre) |
       MoveSomething (pre, post) &&
90.    NothingGetsEaten (post.near) && NothingGetsEaten (post.far) &&
       OneOfEverything (post)
        )
    }

    // The puzzle is solved if the hunter can move everything to the
    // far side of the river.
    fun HunterSucceeds () {
        Ord[State].last.far = (Goat + Cabbages + Fox + Hunter)
    }
100.
    // Let us check how we can do this.
    fun FindSolution () {
        ValidTransitions () && HunterSucceeds()
    }

    // It turned out to take 8 trips across the river.
    generate: run FindSolution for 8 but 4 Object
```

4.3 Debugging

Running this model through Alloy will yeild no solutions, and the user finds that changing the scope does not have an effect; he concludes it is not a greater number of trips that are required, but some change in how we define the system's behavior. Acting on the notion that the flaw in his system will manifest itself at a small scope, he decides to use the instance editor to input a few trips across the river manually, and then he'll check to see if those trips—which he feels should be correct—lead to false clauses.

He inputs this sample instance, where G = goat, H = hunter, C = cabbages, and F = fox.

State #	Near Side	Far Side
1	GHFC	
2	CF	HG
3	FCH	G
4	F	HCG
5	F	HCG
6	F	HCG
7	F	HCG
8	F	HCG

He knows that this is not a satisfying solution, because not all of the objects are on the far side in the last state, and thus the `HunterSucceeds` function will be false, but he is primarily interested in finding if any of the transitions he has given are flagged as invalid.

The user's plan at this point is to enter the instance, which will yield him an AST that has false clauses, and then use the tunneling feature to find the general portions of the model which are unsatisfied; then, using the quantified variable value tree, he will focus in on those portions which are false for the first few transitions that he feels should not cause problems.

After editing the instance, he sees an AST with many false clauses, which are marked in red; however, after selecting "Tunnel to False Leaf Nodes" he sees that only two leaf nodes are false, one of which corresponding to the `HunterSucceeds` function on line 98, as he had expected, and the other corresponding to the `ValidTransitions` function, on lines 85–90. When he selects that node in the AST, the quantified variable value tree is displayed with the values for the `pre` variable, which spans over all eight states. He clicks on the BinaryFormula (which highlights lines 89–91) node, and notices that it is false for the transition from state 2 to state 3, one which he thought should be fine. He descends through the AST tree, the quantified variable value tree maintaining its modality so that he can easily keep an eye on the suspect state, opening only those branches which have a false value for state 2. This takes him to the `MoveSomething` and then the `TransferSomething` functions, but that is the smallest level of granularity of the source that he can glean from the AST at this point.

So, he realizes that there is something wrong with the `TransferSomething` function, but here he has to solve it by examining his instance and his function definition. He sees that the unique characteristic of the transition between states 2 and 3 is that the hunter moves without taking anything with him, and in his model he has made allowances only for the hunter carrying one item (lines 42–44) or to stay where he is (line 47), but not for

moving by himself. So the user changes the `TransferSomething` function to allow for this situation.

4.3.1 Corrected Model

```
fun TransferSomething (from, to, from', to': set Object) {
   // There are three ways in which the Hunter can do things.

   // The hunter can take exactly one object with him across the river.
   (one item: from - Hunter |
      (to' = to + Hunter + item) &&
      (from' = from - Hunter - item))
   ||
   // Or he can make the trip by himself.
   ((to' = to + Hunter) &&
    (from' = from - Hunter))
   ||
   // Or he can just wait where he is.
   (to' = to  &&  from' = from)
}
```

Now he compiles the model and executes the `generate` command again, but now he is yielded a solution.

State #	Near Side	Far Side
1	GHFC	
2	CF	HG
3	FCH	G
4	F	HCG
5	FHG	C
6	G	HCF
7	GH	CF
8		HCFG

Chapter 5

The Alloy GUI

This chapter presents an architectural overview of the Alloy tool's graphic user interface, along with information and tips that would be useful to someone who wanted to modify the code.

5.1 Components

The `alloy.gui` module comprises a number of class, some of which are closely tied to the GUI, while others exist relatively independently. In figure 5-1 is shown a selection of the classes I wrote expressly for the GUI, as well as some of the more user-visible classes written by other developers; this class structure is the result of significant refactoring, after we saw how difficult it was to deal with the large AlloyGUI class that resulted from feature accretion during the development cycle. The *Interfaces* section following each class description lists the interfaces the class provides through which others can listen for appropriate events.

AlloyGUI

This class is the basis for the Alloy tool's interface: it contains the `main` method, and is responsible for constructing the layout of the GUI, connecting user interface elements to handlers exported by the various components, and listening to events that those components send which could affect the state of the GUI, or which must be dispatched to other components.

Interfaces:

Figure 5-1: Object Model of GUI Module

`ProgressInterface`

SwingASTVisitor

This class was written by Ilya Shlyakhter to traverse the formula generated by the solver; it operates as a visitor, and builds a Swing representation of each node, gluing them together into what the user sees as the AST. This class uses the name shortening code in `SolutionData` to pare down long names before displaying the tree.

ListSelector

The `ListSelector` is a convenience class used by the AlloyGUIASTHandler to display the various lists of nodes constructed, and to monitor user interaction with the list.

Interfaces:
`ListSelectionListener`

FontSelector

I was surprised that Swing doesn't provide a prefabricated font selection dialog, which is the essence of this class. Additionally, it uses a preview callback interface to allow the user to preview font changes in the trees and text before making a decision.

Interfaces:
`PreviewCallback`

InstanceEditor

The instance editor, also written by Ilya, provides a dialog by which the user can manually set which tuples are contained in each relation. I modified it so that a map between each widgets and the boolean variable it represents is stored, so that we can load an instance (through the `SolutionData` interface) and update the checkboxes correctly.

LocDisplayer

This class acts along with the `LocatableTextArea` to highlight regions of the text which correspond to a node in the AST.

LocatableTextArea

This class is a standard text area supplemented with facilities to highlight regions of characters—when combined with the `LocDisplayer` class—and also a few miscellaneous functions, such as undo'ing and redo'ing text operations.

Interfaces:

`TextChangedListener FileChangedListener`

AlloySolver

The `AlloySolver` presents a simple interface for the rest of the GUI package that allows it to access the features of AlloyRunner without worrying about threading issues. `AlloySolver` presents a set of callback interfaces through which interested parties can listen for events related to compilation and generating solutions. Additionally, this class prevents events which should not overlap, such as building and solving, from doing so. This class exports a menu of `Commands` that the `AlloyGUI` class can place where it will, and from which the user can select the command he wishes to execute.

Interfaces:

`BuildFinishedListener SolveFinishedListener`

GUITree

This class encapsulates certain functionality that is common to both `ASTTree` and `SolutionTree`, such as changing the font and clearing its nodes.

ASTTree

The `ASTTree` is responsible for displaying both the AST itself and the quantified variable value tree, both of which are exported as separate widgets, for `AlloyGUI` to arrange as it wishes. This class also provides a popup menu through which the user can tunnel to false clauses and bookmark AST nodes.

Interfaces:

`ASTSelectionListener`

SolutionTree

The Alloy Tool presents a solution to a model as a tree that shows which atoms populate the signatures, and the tuples of atoms that constitute relations. Because the expansion of the tree is potentially unbounded, we must generate children nodes lazily, as the user clicks through the tree. `SolutionTree` exports a widget that `AlloyGUI` can arrange in the interface.

5.1.1 Structure of AlloyGUI

Though each section of `AlloyGUI` is relatively independent, the class is still large, and it may be helpful to have an overview of the function that each section performs.

- Start-up and Utility functions (~150 lines)

  A few hundred lines of code are spent on simple routines that perform tasks such as listing and setting Java's look-and-feel, parsing the command line, and saving and loading user preferences.

- `AlloyGUI` constructor (~200 lines)

  The constructor is responsible for instantiating the various components and setting them up in a Swing frame; if one wishes to change the layout of the GUI, it is here that modifications should be made.

- Menu bar constructor (~200 lines)

  This function creates all of the menu components and links them to the `Actions` that define their behavior. The methods for creating the `Action` objects are described below.

- GUI State (~100 lines)

  These are methods which allow the state of the GUI, that is, the enabled state of menu items, whether a solve command is in progress, and messages displayed, to be saved and restored, in event of a crash.

- `Actions` for Menu Items (~500 lines)

  This set of methods define the `Action` objects to be used when selecting menu items. Each method returns a closure to be run as the result of a menu selection, and in addition wraps this closure in error-handling code that causes the tool to write a dump file in the event of an error.

- Methods for Handling `Commands` (~100 lines)

  When a model is compiled, the GUI populates a menu with the commands defined therein, and this code interacts with the `AlloyRunner` to handle the interface between the GUI menu and the underlying `Command` objects.

- Build and Solve Listeners (~100 lines)

  The GUI registers callbacks with `AlloySolver` to be run when an appropriate event is finished; these callback functions take care to set the state of GUI components depending on the outcome of a build or solve attempt.

- GUI Message Listener (~100 lines)

  This inner class listens for error messages emitted by the compiler, displays them in a pane of the main frame, and correlates error messages with the source text when selected.

5.2 Threading in the GUI

Multi-threading is responsible for some of the more subtle errors we have encountered, and I have tried to localize the use of threads, and make their interaction as simple as possible.

AlloySolver uses separate threads when compiling models and executing commands; when finished, it fires `BuildFinishedEvents` and `SolveFinishedEvents`, respecively, in the Swing event thread so that the callback listeners do not have to worry about thread safety when modifying the GUI.

ASTTree uses a separate thread when it builds a Swing representation of the abstract syntax tree, before it realizes any of the components. Upon completion, it displays the AST in one of its managed components.

Thus, outside of these two classes, we need not worry about threading issues, simplifying our reasoning about control flow.

5.3 Design Patterns

Throughout the GUI, we make consistent use of a number of patterns that create a comprehensible design logic, and which let us change parts of the system more easily, so long as the replacement conforms to the same pattern.

SwingWorker

We use the `SwingWorker` pattern given by Sun^{TM}, in which a `construct` method is run in a separate thread, and is intended to perform time-intensive calculations which do not have GUI interactions, and which returns an object to the `finished` method, which the SwingWorker runs safely in the Swing event thread.

Action

It is useful to be able to separate a program's response to an event from the GUI widget that generated the event, and we use Java's `Action` pattern to effect this. Rather than having a module attach a handler to the activation of a particular widget, it can just export an `Action` closure which the `AlloyGUI` can associate with any widget, a menu item, toolbar, or button, as it sees fit; additionally, the same `Action` object can be attached to multiple widgets, and can control certain aspects of their appearance and enabled/disabled state, allowing the layout component to delegate that responsibility to whichever other component defines the behavior. We started using this pattern after our old method, having a large if-else block to deal with all menu events, became too unwieldy and centralized to manage easily.

Observer

This basic pattern is used extensively, both for GUI events, such as the user opening a branch of the solution tree, and for non-GUI events, for example to signal the completion of a compile. By using Observers, we were able to significantly reduce coupling between parts of our system: as an example, rather than having the `AlloySolver` call all the routines

which must be executed when a solve is finished, it just notifies all of its registered listeners, and they can take appropriate action. Thus, if a new module that depends on solve events is added to the system, it can just attach an observer for the event, instead of having to change `AlloySolver` to accomodate it.

ContextListener

Certain widgets, such as the AST display or the list of bookmarked nodes, may contain items upon which other components want to act, and rather than having each of these widgets subsume all the necessarily functionality, they provide an interface by which interested parties can register menu items that the user can activate through a context menu. For instance, the `ASTTree` doesn't know anything about tunneling, but it allows the `ASTHandler` to register a popup menu on the AST from which one can select various tunneling or bookmarking options. This pattern allows a developer to add such functionality without disturbing the class that manages the widget itself.

Chapter 6

Conclusion

My work on the Alloy tool involved the development of features that alleviate some of the problems that users face when debugging models; such features include searching through the AST to find interesting false nodes, bookmarking tree nodes, and shortening long names. To illustrate these enhancements, I showed how someone might use the debugging aids to find a flaw in a sample model.

Much of my work consisted of writing a user interface for the tool, and so I presented an architectural overview of the GUI, in its end result after many iterations, accretions, revisions, and refactorings, in a way that I hope will be helpful to developers who maintain or enhance the code.

Finally, during my work on Alloy, I learned a lot about language design and implementation, software modelling, and program architecture; and found that working on the tool debugged, as it were, many of the misconceived programming faults I had not yet winnowed out of my software design practices.

Bibliography

[1] Jackson, D. (2001). *Micromodels of Software: Modelling & Analysis with Alloy* Cambridge, MA: MIT LCS. `http://sdg.lcs.mit.edu/alloy/book.pdf`

[2] Jackson, D., Schechter, I., & Shlyakhter, I. (2000) *Alcoa: the Alloy Constraint Analyzer* Proc. International Conference on Software Engineering, Limerick, Ireland, June 2000

[3] Jackson, D. (2000) *Automating First-Order Relational Logic* Proc. ACM SIGSOFT Conf. Foundations of Software Engineering. San Diego, November 2000.

[4] Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D. (1999) *Refactoring: Improving the Design of Existing Code* Addison-Wesley Pub Co., 1st Ed.