Visualization Framework for Software Design Analysis

by

Likuo Lin

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Electrical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

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 Students

Visualization Framework for Software Design Analysis

by

Likuo Lin

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 Computer Science and Electrical Engineering

Abstract

Visualization has often been used to facilitate the processing, comprehension, analysis, and management of information. Graphs are a visualization mechanism that are frequently used in computer applications as a data structure to visualize objects and relationships between them. The goal of this thesis was to provide a tool that facilitates the comprehension and detailed analysis of small complex graphs. This tool, also called the Alloy Visualization Tool, used four main techniques to achieve its goal: focus by eliminating unnecessary parts of the graph, differentiating components of different types, enabling layout clarity, and providing higher-arity relation constructs. The tool can also save and load instances that represent objects and relationships between them and customizations on how to view the instances. This tool was applied to an object modeling tool, the Alloy Analyzer, but can also be used as a standalone tool.

Thesis Supervisor: Daniel Jackson
Title: Associate Professor

Acknowledgments

I would like to thank all the users of the Alloy Visualization Tool who reported bugs in the visualization tool and made various usability suggestions. I would also like to acknowledge everyone in the Software Design Group at the Lab for Computer Science for their ideas on different features and improvements for the tool. I would especially like to thank the people on the alloy development team, Daniel Jackson, Manu Sridharan, Ilya Shlyakhter, and Jesse Pavel for the many long discussions over the more challenging specification and design aspects of the tool.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Visualization

Visualization has often been used to facilitate the processing, comprehension, analysis, and management of information. Visualization facilitates the communication of mental models and information between users by allowing them communicate through pictures. Some examples of different visualization techniques are graphs, bar charts, pie charts, multi-dimensional visualizations, and time lines. For an overview of the different research in visualization see [10].

Graph visualization has been a major area of research in visualization as processing large amounts of information has become more important. Graphs are frequently used in computer applications as a general data structure to visualize objects and relationships between them. Graphs provide information in a much clearer and more concise manner than textual interfaces and can be used to represent hierarchies, dependency structures, networks, configurations, data flows, etc.

Most of the research in graph drawing has been mostly focused on the problem of drawing large graphs. Examples of systems that do this are aiSee[1],Walrus[13], daVinci[3], Tom Sawyer[12], and GraphViz[6]. The main problems these systems deal with are viewability and usability issues that arise from an information overload. These systems deal with these problems with layout techniques such as using spanning trees, 3D layouts, and hyperbolic layouts [7]. They also deal with large graphs by using

navigation and interaction strategies such as zooming and panning, focus and context (e.g. fish-eye distortions), incremental exploration and navigation, and clustering [7].

The focus of this research was not in dealing with the layout and viewability of large graphs but in facilitating the comprehension and detailed analysis of small graphs (less than 50 nodes). These small graphs are used to expose intricate relations and not display overall structure. They also deal with the viewing and representation of relations with arity greater which most previous graph visualizations do not deal with. The graph visualization will mainly be used for relational structures for which the objects and relations have types associated with them. With these type associations, higher order operations can be performed based on the types of the nodes and edges.

1.2 Alloy

Alloy is a lightweight modeling language. A model is any analyzable representation of a system. One can model algorithms, software designs, complex systems and virtually anything with some type of structure. Other modeling languages include OCL [9], Z [11], and UML [14]. They each have their own strengths and weaknesses. Alloy attempts to combine some of the strengths of these languages to create a more useful language that is expressive, simple, and analyzable.

There are four main aspects of the Alloy modeling language. First, it is *lightweight.* The language of Alloy is unusually small, but is still powerful and flexible enough for many complex applications. One can use Alloy to express the crucial parts of systems in order to explore the important intricacies of a system with much less effort than a full formalization. Second, Alloy is *declarative* and not operational. One can describe properties about the system instead of describing a sequence of operations as one does in conventional programs. This allows for incremental modeling. As more properties are described, the model's behavior is further constrained. To make a model fit a system more closely, one needs only to add more properties. Third, Alloy is *analyzable.* The Alloy tool can automatically simulate models and check their properties. Lastly,

Alloy is structural. The language focuses on the ability to analyze the structure of the state of systems but is also able to analyze algorithms using these structures.

There are two basic structures in Alloy: the *atom* and the *relation*. The atom is a primitive entity that is:

- **Indivisible:** it can't be broken down into smaller parts.
- **Immutable:** its properties do not change over time.
- **Uninterpreted:** there is no inherent theory or built in properties.

The atom is used as a simplified version of objects or concepts in the real world. Each atom has exactly one *type*. All types are disjoint.

The relation is a structure that relates atoms and contains a set of tuples. Each relation has a relation type, which is an ordered sequence of types. The number of types in a relation type is the arity of the relation. A relation of arity one is used to represent a set. The size of a relation is equal to the number of tuples that the relation contains. Each tuple of a relation is an ordered sequence of atoms that must match the relation type of the relation that contains it. The relation types match if the types of the atoms in the tuple correspond exactly to the types of the relation type.

The Alloy Analyzer takes a description of an Alloy model, a collection of types and relations between them followed by a series of constraints, and translates it into a boolean formula. This boolean formula is then passed to a boolean satisfiability solver that solves the constraints by finding a satisfying assignment for the boolean variables. This satisfying assignment is then converted back to a set of atoms, types they belong to, relations, their types, and their list of tuples. We can also refer to this solution as an instance. This instance either shows a valid example of the current model that satisfies all the constraints, or a counterexample that may prove that one or more constraints do not hold.

1.3 Tiny Example

Here is a tiny example of how the Alloy Visualization Tool can be used in conjunction with the Alloy Analyzer. We begin with an Alloy model:

```
sig Elevator{}
sig Level{}

sig State {
   at: Elevator -> !Level
}

fun Go (){
   two Elevator
   //there is some pair of states s, s' and an Elevator e,
   // such that the level that e is at in State s is different
   // from the level that e is at in State s'.
   some s,s': State, e:Elevator | s.at[e] != s'.at[e]
}

run Go for 2
```

In this simple elevator model we have three types: *Elevator*, *Level*, and *State*. The type State contains an *at* relation from Elevator to exactly one Level signifying the location of the elevator for a particular state. The function *Go* states that there is some pair of states *s*, *s'* and an Elevator *e*, such that the level that *e* is at in State *s* is different from the level that *e* is at in State *s'*.

After compiling and running this model in the Alloy Analyzer, a solution to the model is found. This solution is a list of relations, their types, and the tuples that they contain. Level, Elevator and State, which are relations of arity one, represent the sets that contain the atoms in the solution. The *State$at* relation holds the state information.

```
--------------------------
relation: elevator/Level (type: Level)
{ (Level_0) (Level_1)  }
--------------------------
relation: elevator/Elevator (type: Elevator)
{ (Elevator_0) (Elevator_1)  }
```

```
--------------------------
relation: elevator/State (type: State)
{ (State_0) (State_1)  }
--------------------------
relation: State$at (type: State -> Elevator -> Level)
{ (State_0 Elevator_0 Level_1) (State_0 Elevator_1 Level_1) (State_1 Elevator_0 Level_1)
```

Now, we can visualize this solution with the Alloy Visualization Tool. Several graphs of the solution with different customization settings are shown in Figures 1-1, 1-2, 1-3, 1-4, and 1-5. We can see from these figures that *Elevator_0* remains at *Level_1* in both states, and that *Elevator_1* is at *Level_1* in *State_0* and at *Level_0* in *State_1*. The rest of thesis will describe how these graphs are generated.

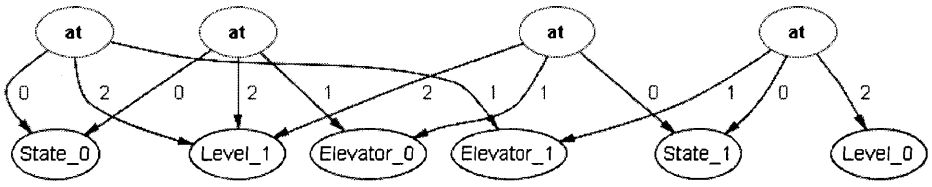

Figure 1-1: Diagram of solution to mini elevator model using hyper arcs
An *at* node represents a tuple of the *State$at* relation. For example, the upper leftmost *at* node says that *Elevator_1* is at *Level_1* in *State_0*.

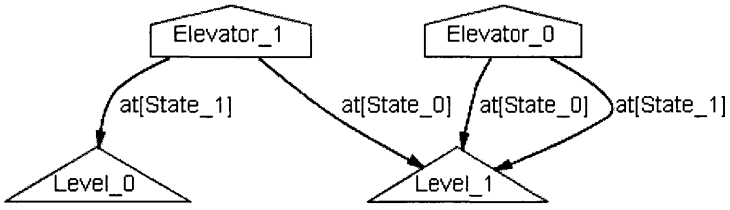

Figure 1-2: Diagram of solution to mini elevator model indexed on State
Edges are labeled with the State for which an elevator is at a Level. For example, the edge from *Elevator_1* labeled at[State_1] says that *Elevator_1* is at *Level_0* in *State_1*.

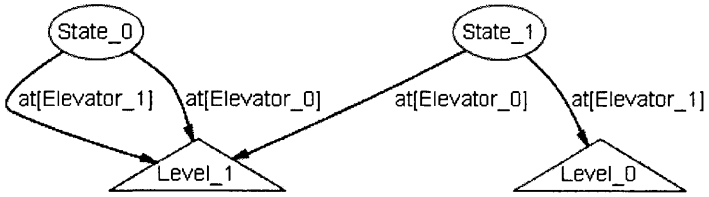

Figure 1-3: Diagram of solution to mini elevator model indexed on Elevator
Edges are labeled with the Elevator that is at a Level for a particular state. For example, the edge from *State_0* labeled at[Elevator_1] sasys that *Elevator_1* is at *Level_1* in *State_0.*

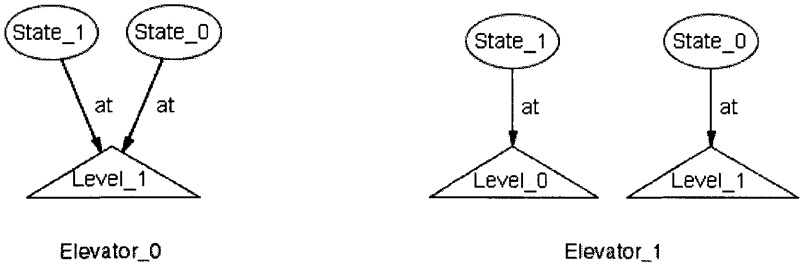

Figure 1-4: Diagram of solution to mini elevator model projecting on Elevator
The graph is projected on *Elevator_0* and *Elevator_1* . For example, the graph projected on *Elevator_0* shows the levels that the *Elevator_0* is at for the two states.

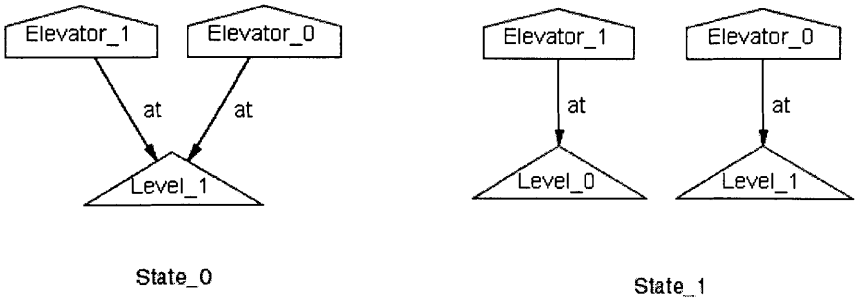

Figure 1-5: Diagram of solution to mini elevator model projecting on State
The graph is projected on *State_0* and *State_1* . For example, the graph projected on *State_0* shows the levels that the elevators are at for *State_0.*

1.4 Thesis Statement

Graph visualization is particularly applicable for Alloy because an Alloy model is essentially a set of objects with relationships between them. The Alloy Visualization Tool was intended to be used for the Alloy Analyzer but it can be used for any structured graph for which types of objects and relations are known.

1.4.1 Specification Goal

The goal of this project was to create a customizable graph visualization tool for viewing relational structures that facilitates comprehension and detailed analysis, and in addition facilitates communication between users.

In order to facilitate understanding and analysis of these relational structures, the Alloy Visualization Tool uses the following four techniques:

- **Differentiation** - Distinguishing between components in the graph.
- **Focus** - Allowing users to focus on the relevant information.
- **Layout and Clarity** - Providing layout customizability and visual clarity.
- **Higher-Arity** - Representing higher-arity relations in various ways that is more comprehensible to the user.

The Alloy Visualization Tool provides the user with the ability to customize a graph of relational structure using these techniques. The ability to generate a visual representation of an Alloy instance facilitates communication of models and concepts between users because users are provided with a clear visual representation of the relational structure. In addition to this, the tool allows the user to save and load both alloy instances and customizations on how to view these instances. Furthermore, the tool allows the user to save the generated graphs into common formats such as Postscript or GIF.

1.4.2 Design Goals

In achieving the goal of providing comprehensible and analyzable graphs, several design considerations were taken into account. I describe them below in order of importance.

- **The most important goal was the extensibility of the design.** This was particularly important during the design of the Alloy Visualization Tool because different features were requested, changed and removed. Extensibility would also allow future developers to more easily add new features without the refactoring or changing of code to be required.

- **Another design goal was to make the Alloy Visualization Tool modular.** The components should be exchanged for similar ones with ease. For example, if it is desirable to use a different third party layout program, it should be fairly easy to make this change. Another example would be to allow plugging in different user interfaces to view and control the underlying system.

- **Lastly, we should consider efficiency.** The graphs are generally small, so for the most part performance was not an issue. However, as more and more features are added, poorly implemented parts of the system will have a noticeable effect. Efficiency should be considered in so far as it does not hinder the other goals of this project.

Chapter 2

Specification

2.1 Overview

The specification of the Alloy Visualization Tool was constantly changing as users continued to make suggestions for features to be added, changed, or removed. It was only near the completion of the tool that most of the features stabilized.

Throughout the development of the tool, the features were intended to have several properties. First, they were meant to be problem independent. These features were meant to be useful whether you were modeling file systems or network algorithms. It was also desirable that the features by orthogonal so that they could be applied independently of one another. Another important aspect was that these features could be applied incrementally. When a user first opens the tool, a graph is initially generated for the user. The user can then incrementally make changes to the graph and see the intermediate results until the user is satisfied with what he sees. Finally, these features are made on a per model basis. This means that a set of customizations for a model can apply to a family of graphs that satisfy that model.

Some of the difficulties in the design of the features resulted from the fact that not all features were independent from one another. Many features that related to the visibility of nodes and edges were linked to each other. Some features affected the availability of other features. This not only complicated the semantics of these features but also complicated the design of the system itself. Through all this, it

was important to ensure that the feature semantics was such that the same graph would result irrespective of the order in which they were applied. Another major challenge was in dealing with the higher-arity relations and providing different ways to represent these relations in ways that the user thinks about these relations.

2.2 Features

2.2.1 Visualization Tool Features

Here, we give a brief overview of the available customization features in the Alloy Visualization Tool.

Differentiation

Differentiation is the simplest technique to use and implement. The features for differentiation are appearance features. The goal of this technique is simply to allow the user to easily distinguish between nodes of different types or nodes in different sets and between edges belonging to different relation types. Color, shape, and label name are used for differentiation in the tool.

Focus

Focus aids the user in focusing on the relevant information. Most of these features deal with the visibility of nodes and edges. The main idea behind this technique is to remove parts of the graph that are not of immediate interest to the user, and thus allow the user to focus on the more important and relevant parts for the current analysis.

Node visibility can be set by excluding types and by including or excluding relations. A node N appears in the graph if the type of N has not been excluded, there is some tuple of an included relation that contains N , and there is no tuple of an excluded relation that contains N. One can also choose to hide all unconnected nodes.

Edge visibility can be customized by selecting whether the edges of a relation

should be shown. The visibility of edges can also be affected by the visibility of nodes. An edge is only visible if all the nodes connected by the edge are visible.

Another feature can be used to reduce clutter in the graph is the merging of edges. Two unidirectional edges are merged into one bidirectional edge if both the unidirectional edges are part of the same relation and join the same two nodes in different directions.

Layout

The layout technique provides visual clarity with graph layout features. However, the challenging part of laying out the nodes and edges is handled by the Dot tool.

The Alloy Visualization Tool does provide the ability to set the ranking of particular nodes so that higher ranked nodes are placed above lower ranked nodes when the ranking is vertical, and to the left of lower ranked nodes when the ranking is horizontal.

Higher-Arity Relations

A major goal was to deal with relations that have an arity greater than two. The idea is to use different representations to display these higher-arity relations in a way that does not clutter the graph. The default representation in the Alloy Visualization Tool is hyper arcs. Hyper arcs represent a tuple as a new node, labeled with the relation name, with edges to other nodes that are part of the represented tuple. The edges to these other nodes would be labeled with the position in the tuple that the incident node was in. The other three representations of higher-arity relations that attempt to reduce the clutter are attributes, indexed edges, and projection on types. These are described below.

Representing higher-arity relations as attributes is fairly simple. Each tuple of the relation is simply described in text within the node representing the first atom of the tuple. One can think of this as containment. Note that this can be used for binary edges as well, and not just for ternary or higher.

Indexed edges represent a tuple by indexing on one or more of the atoms in the

tuple and labeling the edge with these indices. For example, suppose you had a relation R that contains the tuple a1->b1->c1. If you indexed on b1, then you would get an edge from a1 to c1 labeled R[b1]. If there were a relation that remained a ternary (or higher) relation after indexing, it is represented by hyper edges with the new indexed relation name.

One common application of using projection is when the solution represents a sequence of states. The relations representing state components each have a state atom signifying the state that the state component belongs to. Projecting on state would mean to split the graph into several subgraphs so that each subgraph represents the graph projected on a particular state. That is, each subgraph would only contain the relations and atoms that belong to that particular state or that belong to all states. During projection, we remove the atoms of the projected type from each relation. That information is now represented in the index of the subgraph. The resultant graph not only has smaller relations, but also gives users the ability to flip through a sequence of states.

For more details on how these representations work, please refer to Appendix A.3.

2.2.2 Dot Features

In order to implement some of these features, we used the layout capabilities and exploited some of the features of the Dot layout tool from the Graphviz graphing suite developed at AT&T Labs Research [6]. Dot takes a graph description and generates a graph layout for the nodes and edges of the graph [8]. The specification of the Alloy Visualization Tool was constrained by the features available in Dot. The repertoire of features available in Dot fall into the following main categories:

- Node Shape - These include preset shapes and user defined shapes.
- Labels - These include node labels, edge labels, and graph labels in various places.
- Graphics Styles - These include node and edge colors, border options, fonts, and edge arrow appearance.

- Drawing Orientation, Size and Space - These include resizing, orienting the graph and also setting fixed spacing between ranks.
- Node and Edge Placement - These include ranking the nodes and putting weights on the edges (higher weights usually make edges shorter and straighter).
- Advanced features - These include node ports that customizes where edges meet the node, clusters that group a set of nodes, records that split a node into various sections, and concentrators that merge arrows going to the same destination node.

For a more complete coverage of Dot's features, refer to the Dot User Manual [8].

A subset of the available Dot features was used in the Alloy Visualization Tool. Some features dealing with node shape, labels and graphics styles have been used to implement the differentiation technique in the tool. Also, the ranking and rank direction features in Dot were used to provide the ranking of types in the tool.

The rest of the Dot features are currently not used in the Alloy Visualization Tool. Setting the visibility of nodes and edges does not require any Dot features. Most of the higher-arity constructs also only use node and edge visibility and labels.

2.3 An Elevator System Example

We recall that the four main techniques used for enhancing the graph were: differentiation, focus, layout, and higher-arity. We now illustrate the use of these techniques with an example of an elevator system modeled in Alloy. The elevator model describes how an elevator should service a set of requests according to a set of physical constraints and some servicing policies. In the elevator model, the types of interest are *Elevator*, *Button*, *Level* (or floors), *Direction*, and *State*. There are two sets of buttons, *CallButtons* which belong to floors and have a direction, and *FloorButtons* which belong on elevators and have a destination floor. There are three singleton sets for directions: *Up*, *Down* and *Stopped*. Elevators must be at exactly one level at one time. Elevators also have a direction of movement as well as a direction of

service. The type State maps a given State atom to the properties of the system at that state. The state of the system at a particular time includes what buttons are pressed (representing a request), what level the elevator is at, the direction that the elevator is moving, and the direction that the elevator is servicing. Lists of the relevant sets and relations are given in Tables 2.1 and 2.2.

Table 2.1: A List of Relevant Sets

Button	all buttons
CallButtons	Buttons on floors
FloorButtons	Buttons on elevators
Elevator	all elevators
Level	all levels including Floors
Floor	all Floors
Direction	the set of all directions
Up	the up direction
Down	the down direction
Stopped	no direction
State	set of all states

Table 2.2: A List of Relevant Relations

buttons	Elevator -> Button	buttons that elevator contains
callButtons	Floors -> CallButtons	CallButtons Floor contains
above	Level -> Level	Level immediately above
below	Level -> Level	Level immediately below
direction	CallButton -> Direction	direction of request for CallButton
destination	FloorButton -> Floor	destination of request for FloorButton
pressed	State -> Button	set of pressed button at some state
at	State -> Elevator -> Level	location of elevator at some state
movement	State -> Elevator -> Direction	direction of elevator at some state
serviceDir	State -> Elevator -> Direction	service direction of elevator at some state

The solution that the Alloy Analyzer found represents a sequence of snapshots of this elevator system operating. We are mostly only interested in the solution and not in how it was modeled. If the reader is interested, however, the Alloy source code for the elevator model can be found in Appendix B.1.

The initial textual output from the tool shown in Section B.2 is almost incomprehensible. This solution contains a list of all the relations and the tuples in each

relation. Even after graphing this initial set of relations, the output is still incomprehensible because the graph is too large and complicated.

In order to make the graph small enough to fit on a page, the first thing we do is to use the focus technique to remove many of the unnecessary relations and types of properties that we are not interested in. This is one

The result of eliminating these types and relations is shown in Figure 2-1. This graph is still too large and complicated to understand. To eliminate some of the clutter, we reduce the higher relations that contain state by projecting on state. Projecting on state creates frames corresponding to the state of the system at different snapshots in time. We also hide the relation *Level$below* since it is redundant with the *Level$above* relation.

The result of applying the projection and removing the *Level$below* relation is shown in Figure 2-2. One can now begin to see a little of what is happening in the first state of the system. We see edges relating elevators to buttons and directions, floors to buttons, and buttons to directions or floors. But it is still hard to grasp what is happening. To further remove clutter, we are going to make some relations into attributes. These relation are *direction* , *buttons* , *movement* , and *serviceDir* . These relations are chosen because they seem to describe the properties of the components. Once these relations are made into attributes, the relations are described in the node instead of being represented with arrows from one node to another. We also remove unconnected nodes because they are not of use to us at the moment.

The result of applying these transformations is shown in Figure 2-3. We notice that we still have several problems. First, the direction nodes are simply labeled with "direction" followed by a number. Knowing the atom names is useless unless we know the sets (up, down, and stopped) to which these atoms belong. So, we must first replace the names of these atoms with the sets that they belong to. We also cannot see which buttons are in the set of pressed buttons. To distinguish pressed buttons from unpressed buttons, we set the color of pressed buttons to be black and leave unpressed buttons to be white. We also set levels to be light green, floors to be dark green, and elevators to be yellow; these may show up as shades of grey if not

printed in color. The shapes of the nodes are also currently all the same. To further aid the user distinguishing between elevators, levels and buttons, we will select the house shape for elevators and triangles for levels. We also shorten the Button label to B from Button for compactness. Finally, we line up all the levels in one vertical column to help provide a clearer picture of the structure of the floors.

The result of applying these transformations is shown in Figure 2-4. Now we are ready to examine in more detail the sequence of snapshots of this elevator system running. We see that in Figure 2-4 the elevator at *Level_1*is between two floors, moving in the up direction, and also servicing that direction. We also note that the elevator has buttons for all the floors and all of these buttons are pressed. There is also a call button on *Level_2*that is pressed requesting service to go up. The remainder of the diagrams are in Appendix B.3. In Figure B-1, the elevator has moved up one level and stopped at *Level_2*. After the elevator has stopped, in Figure B-2, the button for *Level_2*and the call button on *Level_2*are no longer pressed since these requests have been serviced. The elevator now resumes going up and is at *Level_3*in Figure B-3. In Figure B-4 the elevator has reached *Level_4*and is stopped there servicing requests on that floor. Finally, in Figure B-5 the button for *Level_4*is no longer pressed after being serviced, and the elevator has now changed its service direction and is ready to head down to service the last request.

In this example, we have used differentiation, focus, layout, and projection to make an initially incomprehensible graph into something easy to understand.

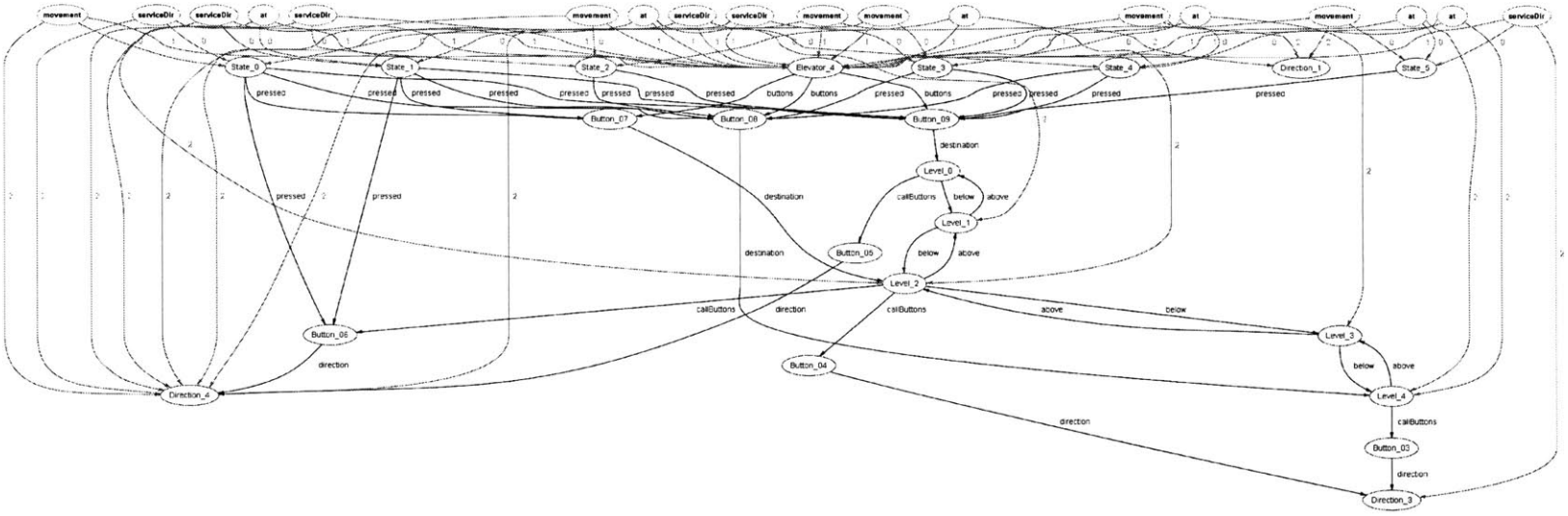

Figure 2-1: Elevator Example without unnecessary relations and types

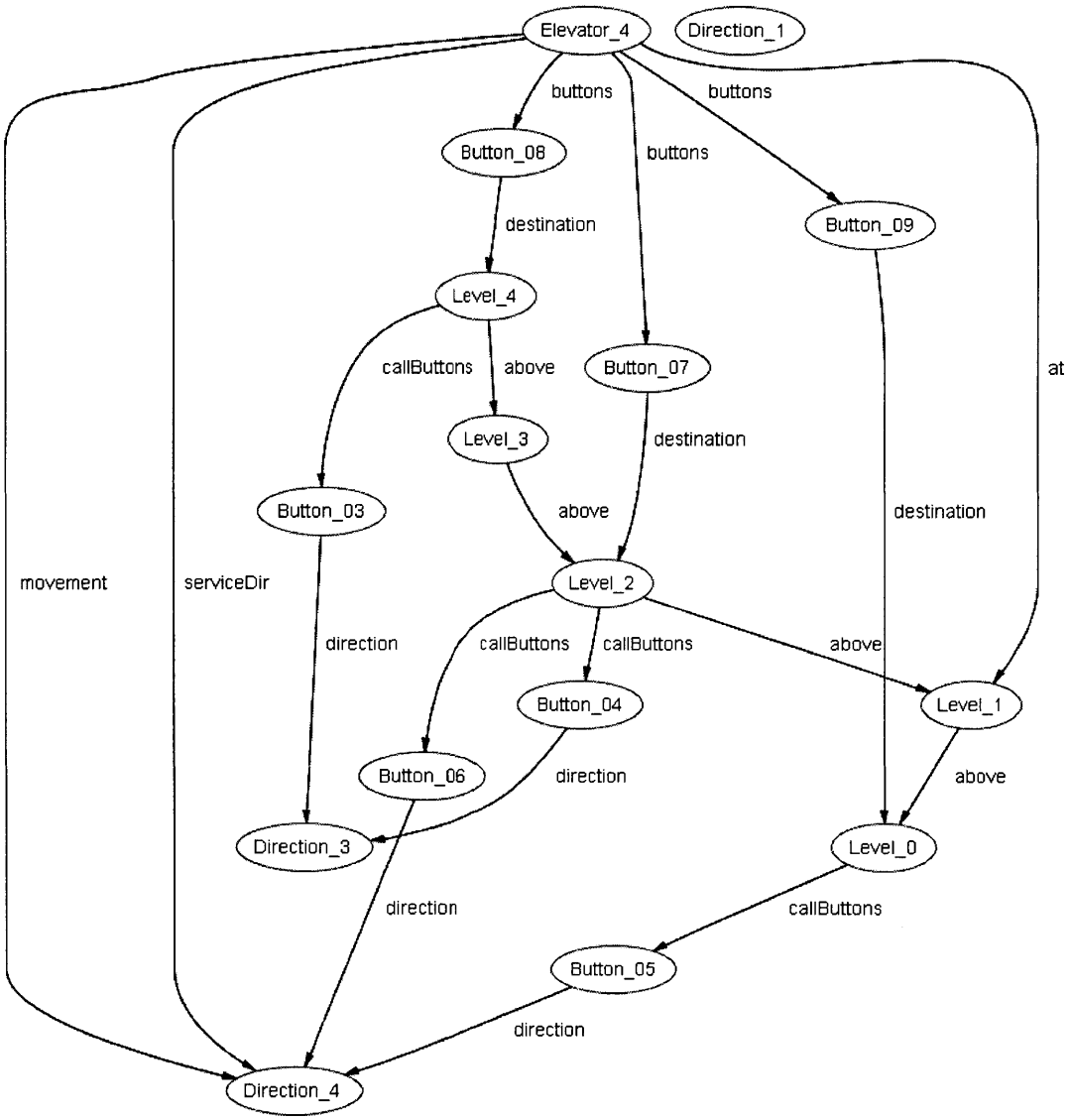

Figure 2-2: Elevator Example Projected on State_0

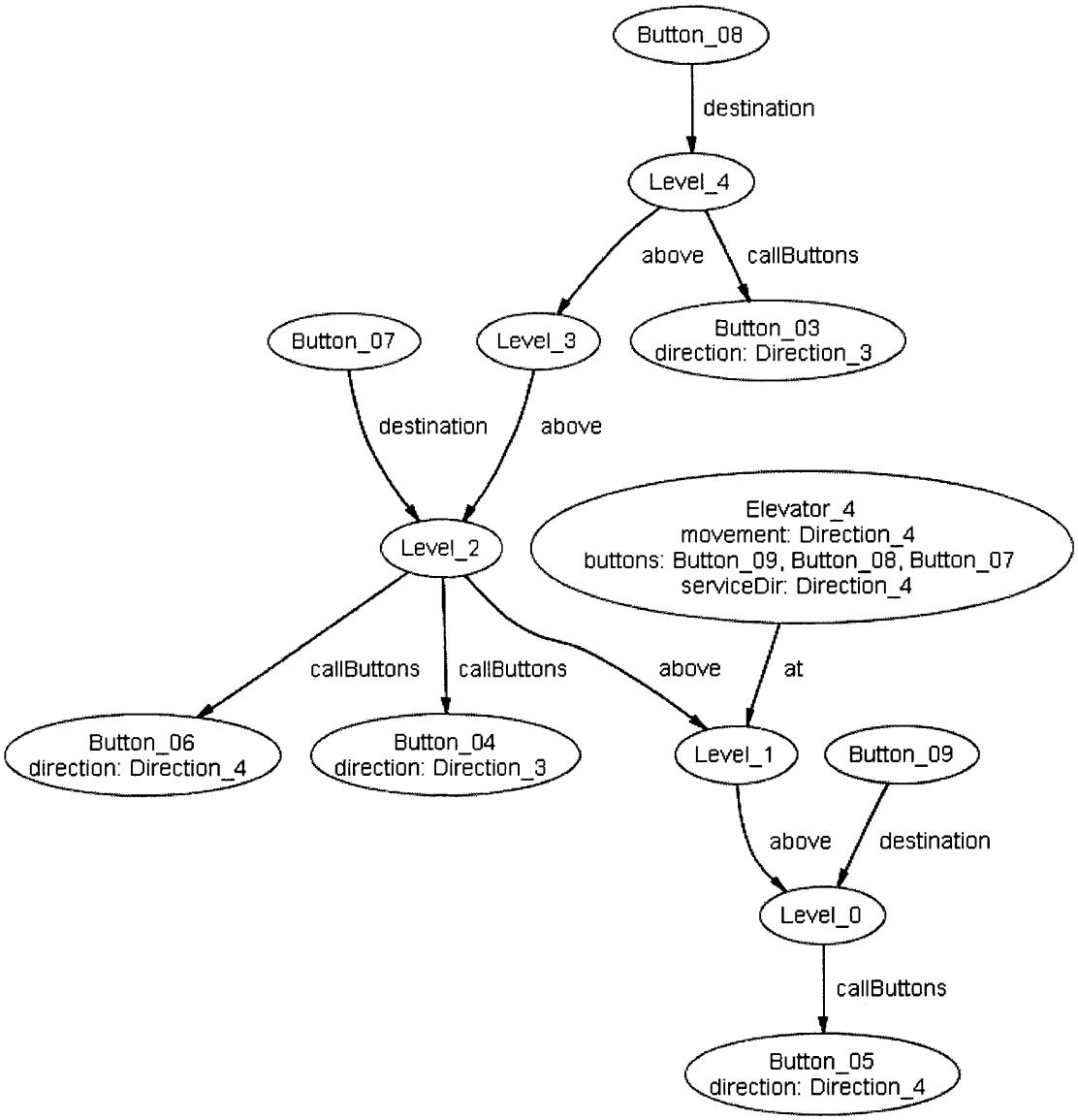

Figure 2-3: Elevator Example with relations turned into attribute

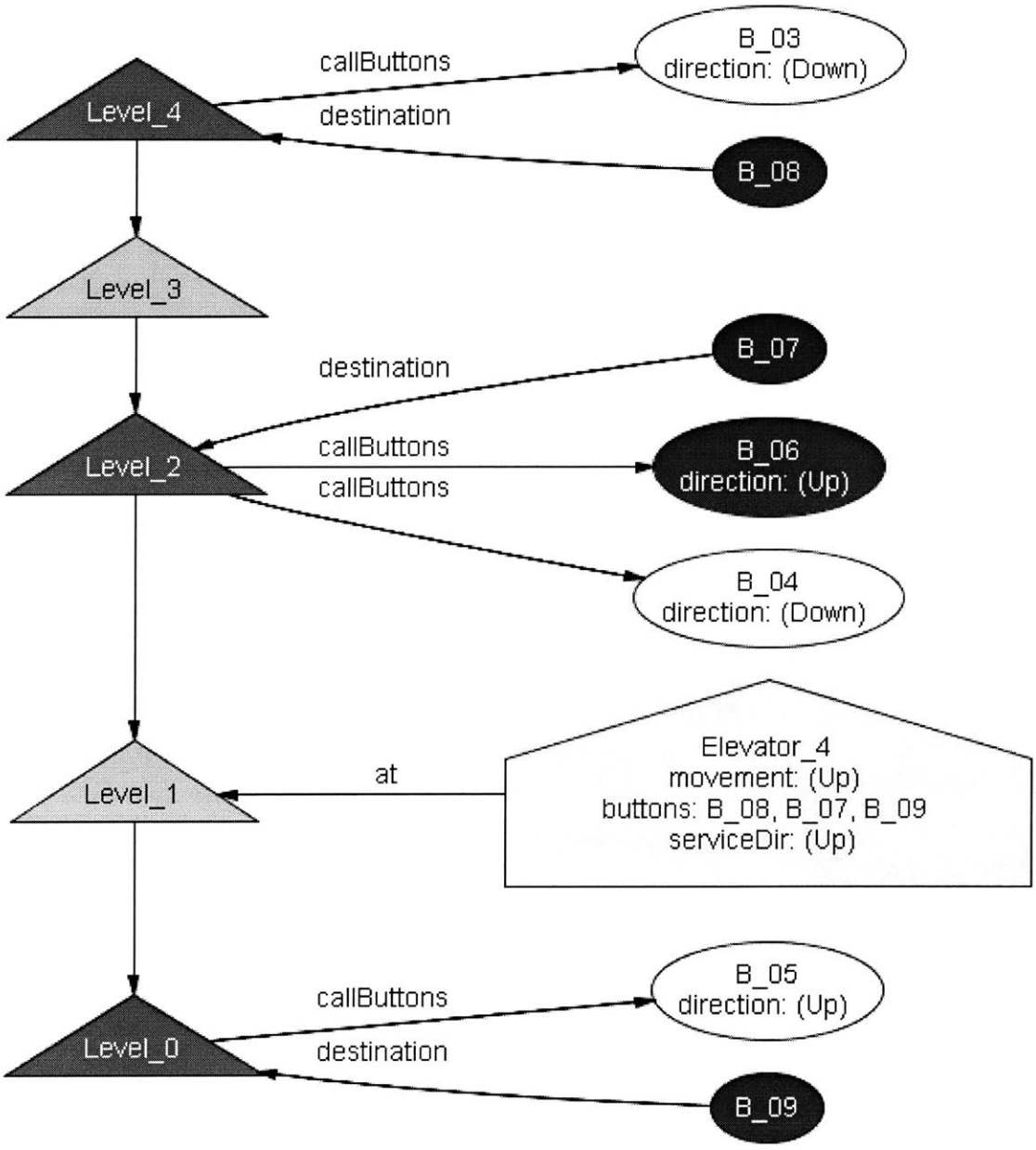

Figure 2-4: Final Elevator Diagram projected on state 0
Pressed buttons are in black. The elevator is about to service requests on Level_2.

Chapter 3

Design

3.1 Design Overview

The Alloy Visualization Tool was created to be both an integrated extension of the Alloy Analyzer and a stand alone tool. The visualization tool has a Graphical User Interface that has two main panels: a Graph Panel that displays the graph that is to be visualized and a Customization Panel that allows the user to set the customizations in the system.

The visualization tool, though written in Java, makes use of third party software. In addition to using the Dot layout tool, the Alloy Visualization Tool also uses a tool written in Java called Grappa that takes dot output, a graph layout description, and draws the graph[5]. Both Dot and Grappa are part of a graphing suite called Graphviz developed at AT&T Labs Research [6].

Many users found the tool to be useful early in its development. However, many changes and refactoring occurred as the tool was made to adapt to specification changes in response suggestions made by the users. Thus, there were many iterations of this design. The entire code base was scrapped and rewritten several times as the new requirements on the tool gave insight to how to make the system more flexible, modular and elegant.

Much of the design challenge for the Alloy Visualization Tool comes from dependencies between the components of the system and the need for it to be easily

extensible. The design of the Alloy Visualizations Tool used a few design patterns to deal with these dependencies. An important feature of the final design is that the dependency graph is acyclic. This made it a lot easier not only to think about how to adapt the tool to changes but also to make the changes themselves.

The general design space is split up into several areas:

- **Representing the three main data components:** the instance, the customization, and the graph.
- **Implementing the graph generation algorithms and format transformations.** These include generating the graph from the instance and customizations, converting the graph into Dot format, and converting Instance and Customization data to and from XML.
- **Designing a user interface that must be placed on top of the data components and tools that allows the user to control interactions between them.** And in doing this not requiring any of the data components or tools to depend on this user interface.
- **Designing the part of the system that would act as a mediator for the communication necessary because of the complex dependencies between the data components, user interface, and tools.**

3.2 Design Patterns Used

In order to achieve the design goals of being extensible and modular, several main design patterns were used in the Alloy Visualization Tool.

Central Registry

A simplified version of the Central Registry [15] was used to act as a mediator in the system. The central registry is a design pattern in which there is a Singleton Class, the Central Registry, that handles events in the system. Components can register

EventHandler, EventFilter pairs with the Central Registry. Components notify the Central Registry of events that have occurred. The Central Registry then processes each of these events by applying each event filter to the event. For each event filter, event handler pair, if the event filter accepts the event, the event handler is executed.

There are several benefits to the Central Registry. First there are no dependencies from the data objects to the observing objects. This is done by triggering an event whenever changes to a data object require the observers to update themselves. Any observer interested in this event should have registered an event filter and handler pair with the central registry. When the data object changes and triggers an event, a properly registered event handler and event filter will respond to the event by updating the observer's view.

The Central Registry design pattern also allows any number of observers to observe the data, without the data objects knowing how many observers are observing them, or that any observers exist. It is the responsibility of the observers to register themselves to the central registry and the responsibility of the central registry to notify them. This is an advantage over the standard observer pattern where the subjects are required to know of all their observers. Observers, therefore, must implement a certain interface. However, in order to pass these observers to the subjects, they may have to go through intermediate modules. This may get tedious and force modules to depend on the existence of these observers and the different interfaces when they do not need to. There could also be multiple interfaces that the observers have to implement and which the underlying modules must depend on - adding to the complexity. Furthermore, it is tedious not only to write the code that passes each of these observers around, but also to make changes if the structure of system changes. In the central registry pattern, the observers depend on the Central Registry and the events signaled by data objects.

Finally, for events, there can be finer control of what happens in response to a change in one part of the system. Each event can be very specific so that only the necessary work is performed.

The lack of dependence from the data objects on the observers in the Alloy Visu-

alization Tool allows us to implement and insert different user interfaces (UI) without any changes in the rest of the code. Each UI component can choose which events it wants to listen to depending on what it wants to display. The data objects must simply notify the central registry of the important events that may be relevant to anyone observing the data.

Abstract Factory

The second design pattern that was used was the Abstract Factory [4]. The Abstract Factory defines an interface for creating families of related or dependent objects without specifying their concrete classes.

This pattern was used in the user interface generation part of the tool. Only the main top level frame depends on the abstract PanelFactory, and only on the abstract PanelFactory's interface. Concrete factories can then extend the abstract factory to implement multiple user interfaces. This makes it very easy to implement multiple interfaces and to switch between them.

Typesafe Enumeration

The third design pattern used was the Typesafe Enumeration [2]. This pattern implements a typesafe enumeration by defining a class that represents a single element of the enumerated type and does not provide any public constructors. The enumerated type that was used in the visualization tool is the attribute type (AttrType).

The main benefits of the typesafe enumeration are type safety and added functionality. Because it is typesafe, it is hard to misuse the enumeration in a way for which it was not intended. Also, since there are no public constructors, all instances of the AttrType are valid. Since the enumeration is a class, methods and properties can easily be added such as default values, valid value types, and valid values. In addition, the Typesafe Enumeration is not less efficient than integer enumerations since all equality tests can be done with a '=='.

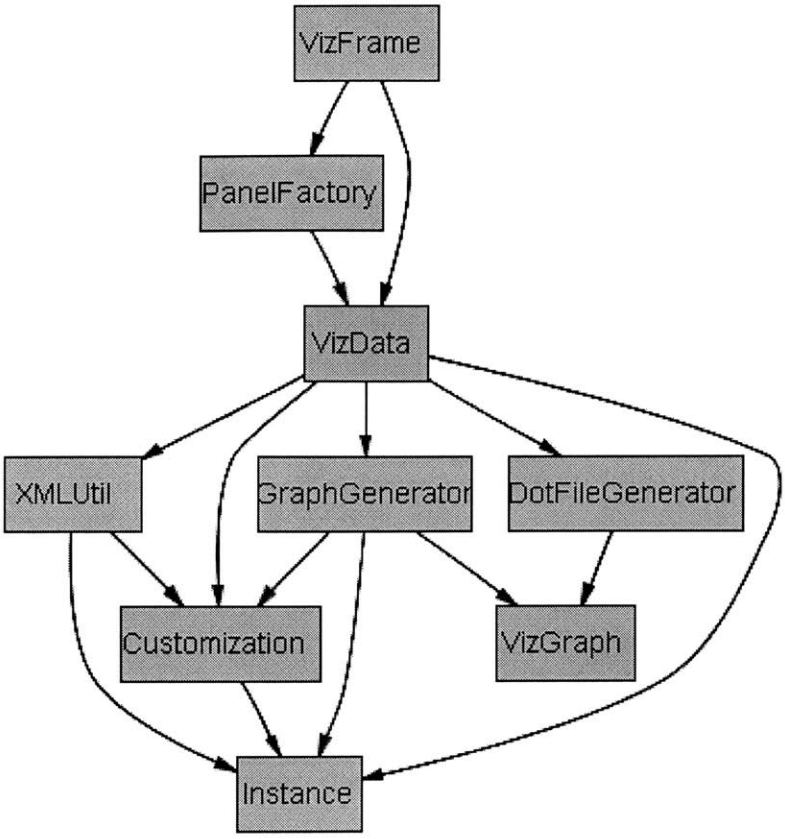

Figure 3-1: Top Level alloy.viz Module Dependency Diagram
An arrow in the graph from a component A to a component B shows that A depends on the specification of B.

3.3 Structure

3.3.1 Overall Structure

We will begin this section by describing the overall structure of Alloy Visualization Tool from the diagram in Figure 3-1 and fill in some of the details later. VizFrame, at the top, is the main frame that handles requests from the command line and from the Alloy Analyzer. The VizFrame depends only on the PanelFactory and the VizData to perform all its functions. It uses the PanelFactory to create the UI components that it contains and it uses VizData to hold the data and to perform requests on it.

The PanelFactory is an abstract class and provides an interface to create all the viewable components in the tool. It depends on VizData and most of the data com-

ponents to create the viewable components.

The state of the system is contained in VizData. VizData depends on two data components which are Instance and Customization and three tool components which are XMLUtil, GraphGenerator, and DotFileGenerator. For the most part, VizData only needs to make high level requests on these components, and pass them as arguments to each other; thus VizData changes very little over time.

The Instance and Customization components correspond to a model that is a set of Basic Types and Relations with their Types. Customization depends on the Instance because it sometimes needs to match its model to the current instance.

XMLUtil depends on the Customization and Instance in order to generate XML from them and to reconstruct them from parsed XML. The GraphGenerator takes a Customization and an Instance and creates a VizGraph object and thus depends heavily on all three data components. The DotFileGenerator, which has only static methods, takes a VizGraph and generates a Dot file that represents the graph. Dot can then use this file to generate a layout file which can be passed back into the Alloy Visualization Tool to be made into an image by Grappa.

3.3.2 UI Components

PanelFactory

The module dependency diagram in 3-2 captures the important dependencies in the Alloy Visualization Tool. At the top, GTRPanelFactory and VALPanelFactory both extend the PanelFactory. They are both implementations of different UI's that the user can select from. They also depend heavily on the VizData and the other data components in order to create the viewable components. The benefit of this design is that new UI's can be quickly written by extending the PanelFactory and only overwriting methods where change is necessary. Furthermore, adding a new factory that implements a new UI requires no change to the rest of the system.

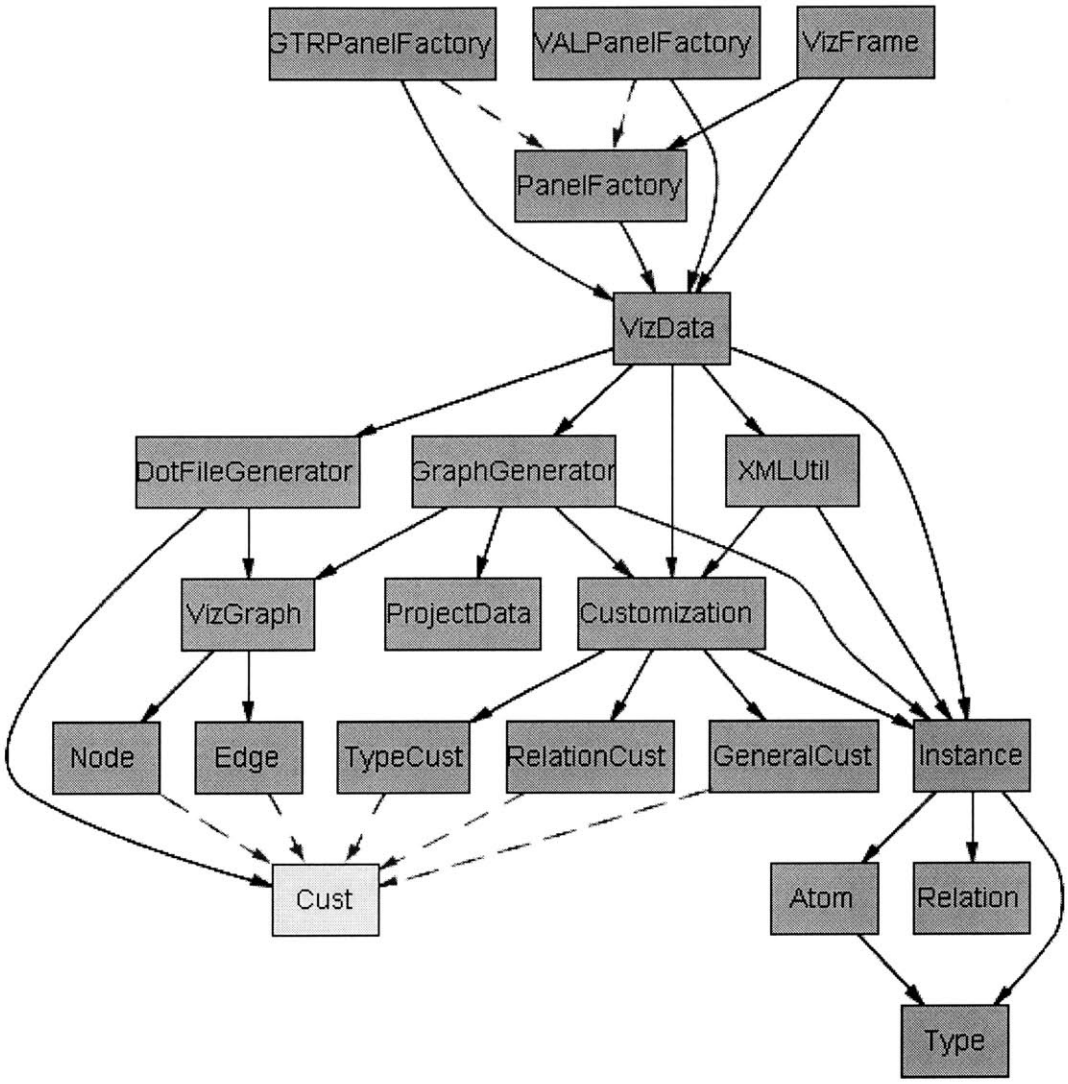

Figure 3-2: alloy.viz Module Dependency Diagram Subset
An solid arrow in the graph from a component A to a component B shows that A depends on the specification of B. Dotted arrows denote subclassing.

3.3.3 Data Components

Instance

The Instance represents the relational structure that we are interested in visualizing. It is similar to a solution of the Alloy Constraint Analyzer. This component contains Atoms, Types and Relations corresponding to their identically named counterparts in Alloy.

The instance component was rarely modified. This was convenient because many components had dependencies on the instance while it had no dependencies on other components.

Customization

The Customization represents all the customizations that should be applied to an instance. Customization contains one GeneralCust, a set of RelationCusts and a set of TypeCusts which all extend the Cust abstract class (see Figure 3-2). Customization depends on all these components in addition to its dependence on Instance.

The Cust abstract class contains a table mapping attribute types to attribute values. It also has all the methods to get and set these attributes. The General-Cust contains all the customizations that apply to the whole graph. RelationCust and TypeCusts contains all the customizations that apply to a relation or a type respectively.

The benefit of this design is that most of the functionality is in the Cust abstract class. The RelationCust, GeneralCust, and TypeCust are very small classes which only have a list of their attributes and a few methods. Also, the customization only depends, for the most part, on the Cust interface. This allows the available customizations to be changed very rapidly and with ease. Most of the time, adding a new customization feature only requires adding a single line to one of the classes that extend Cust.

VizGraph

VizGraph is the representation of the graph resulting from the application of customizations to an instance. VizGraph contains a set of nodes and edges and thus has dependencies on these classes. VizGraph, Node, and Edge, all extend the Cust abstract class and thus each of these components have a set of attributes. These attributes describe how the graph looks.

Note that both the DotFileGenerator and GraphGenerator only depend on VizGraph and not on Nodes or Edges. This allows the representation of the VizGraph to change over time without modifying either the GraphGenerator or the DotFileGenerator as long as VizGraph maintains its interface.

3.3.4 Tools

ProjectData is only class that remains unmentioned in Figure 3-2. The GraphGenerator uses ProjectData to keep some intermediate state while generating multiple projected graphs.

Each tool component has a very specific and clearly defined purpose and only a few high level calls are made on each tool. These properties of tool components allows them to be replaced with different tool components with great ease. Thus, it will be easy to create different semantics for applying the customization by replacing the GraphGenerator or to write graph output for another layout program by replacing DotFileGenerator.

3.3.5 Other Components and Dependencies

In Figure 3-3, a complete module dependency diagram is provided for the alloy.viz package.

VizFrame contains a CustomizationPanel and a VizPanel which were not shown in Figure 3-1 and Figure 3-2. However, because VizFrame uses the PanelFactory interface, it needs to know nothing about these classes; it simply gets them from the PanelFactory and treats them as JComponents. The CustomizationPanel is a view

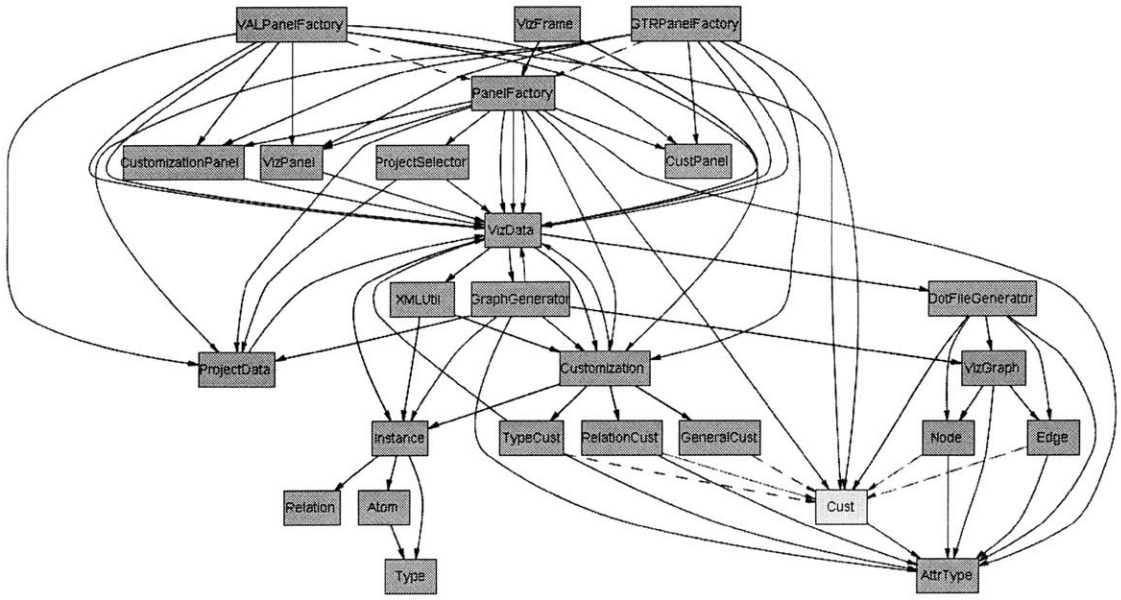

Figure 3-3: alloy.viz complete Module Dependency Diagram
An solid arrow in the graph from a component A to a component B shows that A depends on the specification of B. Dotted arrows denote subclassing.

on the customization that allows the user to view and modify customization settings. The VizPanel contains the final graph that is displayed to the user.

We will now talk about some of the remaining dependencies in the system. Most of the dependence edges that clutter the module dependence diagram fall into three categories:

- **Dependencies on AttrType.** Any class that needs to contain or use any attribute types will have to depend on AttrType. Some components depended only on the existence of AttrType since they never make any method calls to it, but many depended on it to get its string representation, description, or default value. It was important to put all of the properties of an attribute in the AttrType and not in the containing component since this made it easy for attributes to be easily added to other components and passed around.

- **Weak dependencies on VizData.** In order to allow multiple visualizations at the same time, events from different independent visualizations had to be distinguished. To do this, we filter most events by at least the VizData that

the event belongs to. VizData is a reasonable object to filter on since we expect that there is exactly one VizData for every independent visualization. This also allows multiple views on the same VizData while preserving functionality.

- **dependencies from the observers to user interfaces classes.** In order to properly display these components, the observers must depend on every data object that they display and on the observer classes that they must create.

There is also a set of dependencies on the CentralRegistry. But these are not shown in Figure 3-3 since they are part of the alloy.viz.event package and adding them would only serve to clutter the graph without adding much information. One can just assume that nearly every major component depends on the central registry system.

3.4 Program Flow - Algorithms

The user must first supply an instance as the input to the Alloy Visualization Tool. This can be done in one of two ways. One way is to provide the instance from the Alloy Analyzer in the form of a SolutionData object, either from the Alloy Graphical User Interface or the Alloy Command Line Interface. The SolutionData object is part of the Alloy API and represents a solution given by the Alloy Analyzer. The second way to supply an instance is to pass it an instance in XML format.

Once the Alloy Visualization Tool has started, the user can make three main types of requests to the tool: setting customizations, generating a graph for the current instance and customization, and updating, loading or saving the current instance or customization.

3.4.1 Setting Customizations

The user is able to set customizations through the Customization Panel. Each time the user clicks on a button or checks a check box, the action immediately causes an update of the customization in the tool. The Alloy Visualization Tool also propagates

any effects that should immediately take place because of this change. For example, if the user chooses to project on a type, the UI must display a different set of available customizations to the user.

3.4.2 Generating the Graph

There are several ways in which the user could initiate the graph generation process of the tool. One way is through the standard call to generate the graph and display it in the Graph Panel. Another way to do it is to request the Alloy Visualization Tool to write the graph to different formats such as GIF and PostScript. Each time a new customization and instance is to be used to generate a graph, a GraphGenerator object is created for that customization and instance pair. After a GraphGenerator is created, the Alloy Visualization Tool can make successive requests to get a graph in the form of a VizGraph. The GraphGenerator is free to cache any results it feels necessary. When a type is projected, there could be several subgraphs of a single instance customization pair. These subgraphs are generated on demand by the GraphGenerator. These subgraphs can also be cached by the GraphGenerator but caching is not currently implemented.

After a VizGraph is obtained, the DotFileGenerator can generate a layout file using the VizGraph object. The layout file can then be converted into a java image in a JPanel using Grappa and displayed to the Graph panel, or be saved in a particular file format.

3.4.3 Updating and Saving Customizations and Instances

The user can update the Instance by loading an XML file or by updating with a Solution Data object from the alloy analyzer tool. The user can update the customizations by creating a new customization or by loading a previously saved customization file. Each time any of these updates are done, the model of the customization is matched to the model of the instance.

The user can also choose to save the current instance or customization to an

XMLfile that can be loaded into the tool at a later time. The user can also change the UI for viewing the customization.

3.5 Discussion

The coupling and dependency challenges in the design of the system fall into two main categories: Dependencies between components of the Alloy Visualization Tool, and between customization features. Dependencies between components of the Alloy Visibility Tool was for the most part dealt with using events. Dependencies between customization features was mostly resolved by ordering when features were applied and having certain features take precedence over others to get clear feature semantics.

3.5.1 Component Dependencies

We first talk about the dependencies in the interactive state of the visualization tool.

Many component dependencies came from certain customizations affecting the availability of other customizations. For example, when a projection occurs, the arity of a relation may change as a result of this. Different options are available for different arity relations. Therefore, the GUI should update the available options to reflect those options that still have meaning given the new arity of the relation. For instance, if the arity of a relation goes from two to one, all the edge options are no longer valid while the set color and set label options now have meaning.

It was useful to use events to deal with these dependencies because it was important for each of the subcomponents to not have to depend on the components it may effect. For example, the project setting should not need to know about every relation that may contain a projected type, or about the observers on these settings. The project component simply signals to the Central Registry that a projection has occurred. Other components in the system that need to react to this projection then register themselves to be notified of that event and to act accordingly.

Another portion of these dependencies comes from data objects needing to notify their observers to update themselves. Some events that require observers to update

themselves are when there is a new customization, or a customization is loaded from a file, or the name of the customization file is changed. This is very similar to the observer pattern. Using the Central Registry design pattern, we simply have an event signifying that a subject has been modified. The observers are notified through the central registry and update themselves accordingly.

Another dependency comes from the fact that subcomponents have the option to request the current graph to be visualized. These subcomponents do not need to know what component is creating the visualization. For example, suppose a user is flipping through the different graphs of a projection. The button that the user clicks on does two things. It updates the state of the GraphGenerator so that it knows what atoms are currently projected. It also makes a request to visualize. This is nice because it does not even need to know how many panels are interested in visualizing the current instance. Nor does it need to depend on the interface of the component that is visualizing the graph.

3.5.2 Customization Feature Dependencies

The customization feature dependencies in the graph generation phase is not extremely complex. One just has to think carefully about how each setting affects the parts of the graph and how the settings affect each other.

One way to deal with some of this complexity is to put the features in order such that all information is available when the feature is applied. One also has to be careful about which settings can be set before knowing the other settings. This is simpler in one sense because if the order of the settings is correct, then we know exactly when each setting is applied and need to only consider the settings that were applied before the current one.

To illustrate the complexity in graph generation, we look at the process of setting which nodes and edges are visible. Visibility of nodes depend on projection, set inclusion/exclusion, and connectedness of visible edges. Visibility of edges depend on projection, relation on/off, visible nodes, relation indexing, and merging of arrows. Some of the complexity comes from the fact visibility of edges and nodes depend

on the visibility of each other. Since there is a clear precedence on these settings, if the visibility settings were applied in this order, then the correct behavior would result. Projection and type inclusion/exclusion is done first. Then relation inclusion/exclusion for nodes would then only be applied to nodes that were not excluded during projection and type exclusion. Then edge visibility can be set, followed by indexing of edges. Then we only allow edges to appear for which all nodes of the edge are visible (Note that there may be fewer nodes required as a result of an indexed edge). The unconnected nodes can be hidden if desired. We see that even though the order determines the semantics of node and edge visibility, the interdependence of these visibility features still add to the complexity.

Chapter 4

Conclusions

4.1 Lessons Learned

There are several lessons that I have learned through creating this tool.

First, refactoring takes a lot of time and can create new bugs, but many times hard to avoid because all the requirements are not known ahead of time and one cannot possibly account for all possible changes to the system. Many times the idea building many prototypes is not so bad. It would have been better, however, if there was a testing suite that checked the system to see if anything has broken after a refactoring was done.

The final design of the system seemed to be a lot simpler and easier to implement than previous designs. However, the simplicity was actually the result of hard work and design thought rather than something that was obvious at the time. Hard problems always seem very easy after a simple but elegant solution is given.

I also learned that most times you have to create some dependencies to remove others. And although using some constructs that split up functionality into several classes creates more dependencies, it may make the design better. Some types of dependencies are certainly better than others. For example, the dependencies on the Central Registry and the events made the system a lot simpler to reason about and implement even though there were seemingly many more dependencies that were added.

4.2 Future directions

The tool is undergoing constant improvement even at the time of the writing of this thesis. Some improvements that could be done in the near future include the following:

- Exposing some of the dependencies of customization settings to reflect the available customizations as the user makes different settings. For example, hiding the edge color option when the edge has been made made invisible.

- Currently, the graph display to the user is not interactive. Grappa actually provides some features that are not supported by the visualization tool. One improvement to the tool would be to have better integration with Grappa and exploit some of the extra features of Grappa.

- There are some improvements that could be made in terms of efficiency. The UIComponents are rather complex objects that take a long time to create and take up a lot of memory. It would be much more efficient if there was an UIComponent Pool that was the source of all the UI components. The pool could reuse objects that are no longer used by some other component instead of creating one each time.

- Although, the functionality is there, the Alloy Analyzer GUI does not currently take advantage of the ability to have multiple visualization windows open. That would be an easy and useful addition to the tool.

- Beyond the user interfaces that have already been created for the customization panel, more user interfaces could be added to make the user experience more pleasant. One particularly useful UI would be a simple mode for novice users who do not want to deal with the more complex features. Having this simple mode will become increasingly important as more and more features are added that could overwhelm the novice user.

- Ideally one should be able to look back at old visualizations and also check the model and constraints from which it came. If this could be made to work in conjunction with a versioning system for the models, it could be very useful for users.

- One flaw in the layout of the tool is the stability of nodes across frames of a projection. It becomes confusing to the user when the same node moves around the graph through different frames. This is actually a very complex problem since it is difficult to satisfy user requirements of the location of nodes and still run the complex algorithms that are necessary to layout the graph. This is also a problem because the tool uses Dot as its layout program and therefore has only control limited to the user features available in Dot.

- In the future, one may consider using more of the available Dot features. Weights of edges could give the user more control of the graph layout. One may also consider using more of the graphic style and label features to further enhance the graph appearance. Some other features that have not been exploited yet but may show some promise in the future are the advanced features. Clustering of nodes or representing them as records could aid in reducing the complexity in a graph. Node ports might also be useful in affecting the layout of a graph. Finally, edge concentrators may be an easy way to reduce unnecessary clutter in the graph.

4.3 Accomplished Goals - Contributions

The Alloy Visualization Tool provides the ability to visualize and customize small graphs for relational structures by integrating graph layout with a set of customization features. It also provides a solution for viewing and representing higher-arity relations. The final design achieves the design goals of being extensible, modular and efficient.

The Alloy visualization tool is currently very useful in providing graph representations of Alloy models to users of Alloy. The future of this research could go both

towards improving usability and extending the available customizations not only for Alloy instances but for relational data structures in general.

Appendix A

User Manual

A.1 How to Start the Tool

The Alloy Visualization Tool was implemented both as an extension to the Alloy Analyzer and as a stand-alone tool. There are three ways to start the Alloy Visualization Tool:

- Through the Alloy analyzer graphical interface. Once the Alloy analyzer has found a solution, the user can type Ctrl-U or select 'visualize' from the 'tools' drop down menu.

- Through the Alloy command line interface by using the -V option. The user can also load a customization setting with the -f option followed by the name of the customization file.

  e.g. *java alloy.cli.AlloyCLI model.als -V -f model.cst*

- Through the top level visualization class, VizFrame, by passing it an instance file followed by an optional customization file.

  e.g. *java alloy.viz.VizFrame model.ins model.cst*

From here the user can begin to view the graph and set customization settings. There are two main panels that the user will see: the graph panel which displays the

graph, and the customization panel which displays the customization settings that the user can set. There are also three drop down menus: File, Graph and UI. The File menu includes the commands to save and load instances and customizations. The Graph menu contains the commands to generate the graph and display it to the graph panel, and has the command to export GIF or postscript files of the graph. Finally, the UI menu allows the user to select different user interfaces.

A.2 Menu Items

This list describes how each of the menu items work.

A.2.1 File Menu - Loading and Saving of Customizations

Save Instance

This saves the current instance to a file in XML format.

Load Instance

This loads an instance from an XML file. After this instance loads, if the filename is different than the filename of the previous instance, then a new customization is generated for the new instance. Otherwise, old customizations are kept but the model of the customization is forced to match that of the instance. Some settings may be lost when the model of the customization is made to match the model of the instance.

Save Customization

Saves a customization to file in XML. The format of the XML is in the appendix. The default extension of these files is ".cst" since the standard file filter filters out .cst files. The program however would read the file with any extension as long as it has the correct format.

Load Customization

This loads saved settings into the current customization. The current settings are cleared. Invalid settings or missing settings of the customization file will be set to default values.

New Customization

Creates a completely new customization whose model matches that of the current instance. All settings are set to default values.

A.2.2 Graph Menu

Graph

This writes the graph to the graph panel. If the graph has been projected, then there will be a set of buttons that allows the user to flip through the different subgraphs.

Write to Postscript

Writes the graph in the graph panel to a Postscript file that the user specifies.

Write to GIF

Writes the graph in the graph panel to a GIF file that the user specifies.

A.2.3 UI Menu

General Type Relation

Selects a user interface that separates the customization features by General Settings, that apply to the entire graph, Type Settings, that apply to specific types, and Relation Settings that apply to specific relations. Figures A-1, A-2, and A-3 show some screen shots of this user interface on a Windows machine .

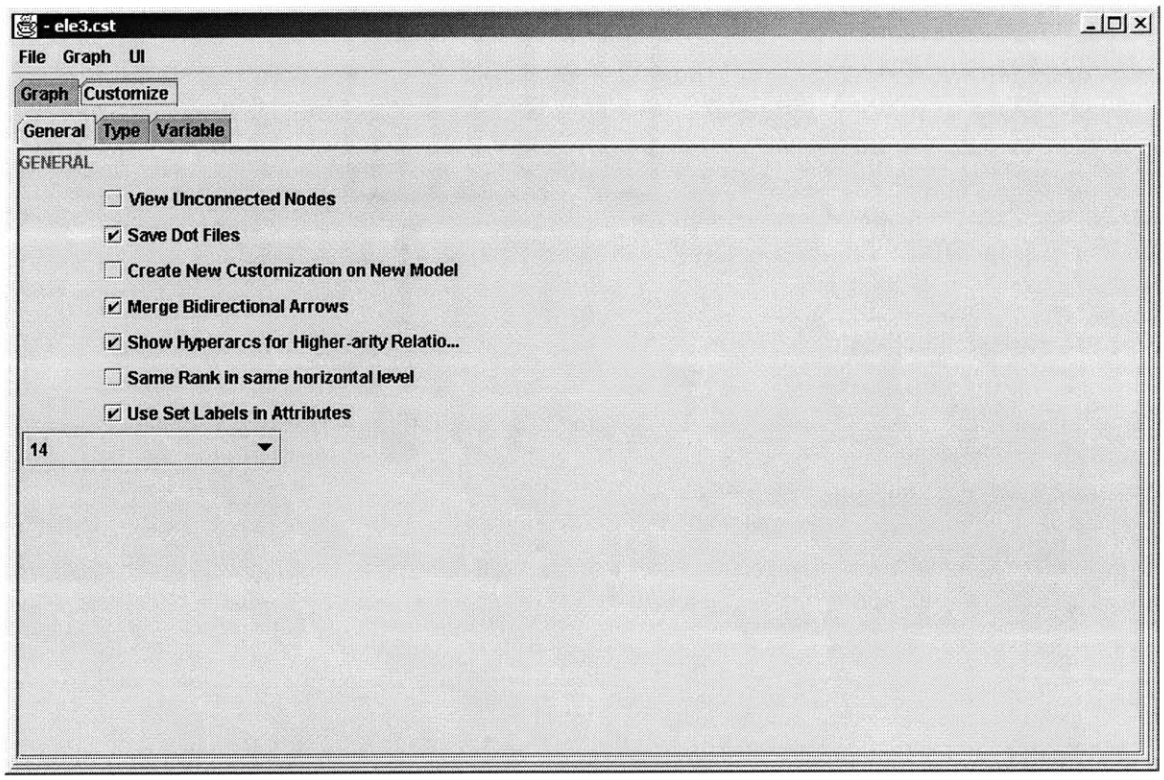

Figure A-1: General Customization Panel

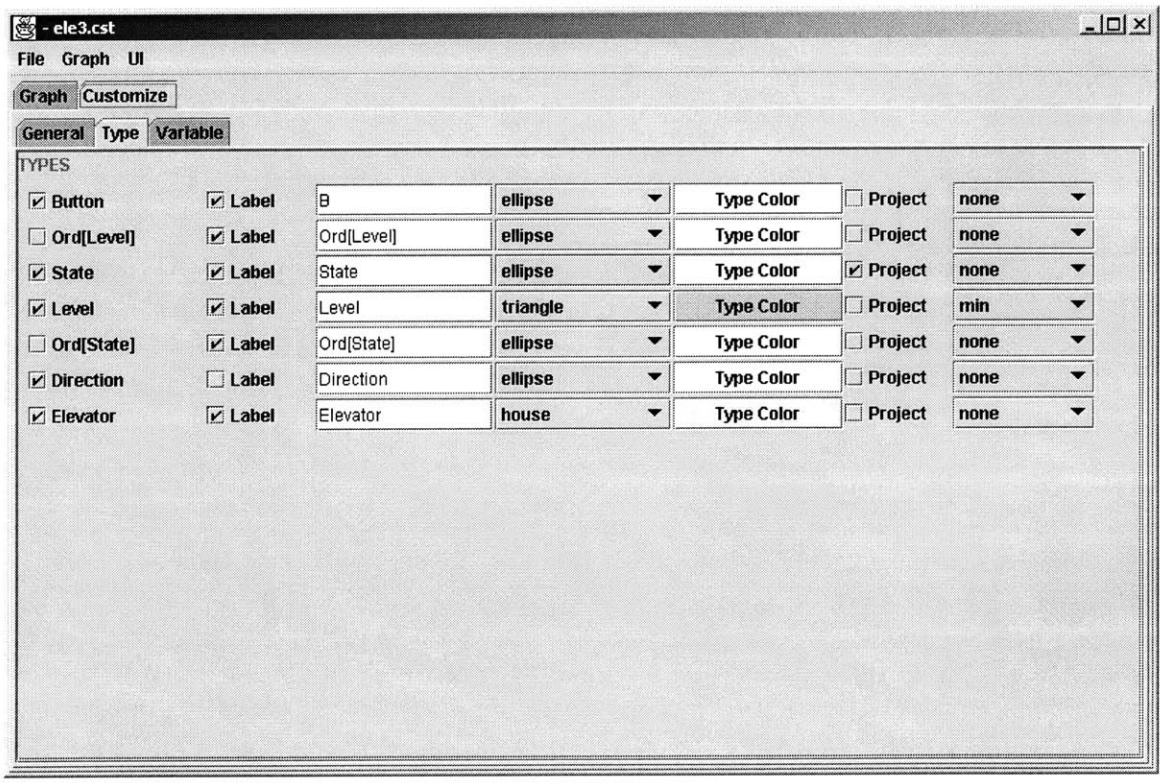

Figure A-2: Type Customization Panel

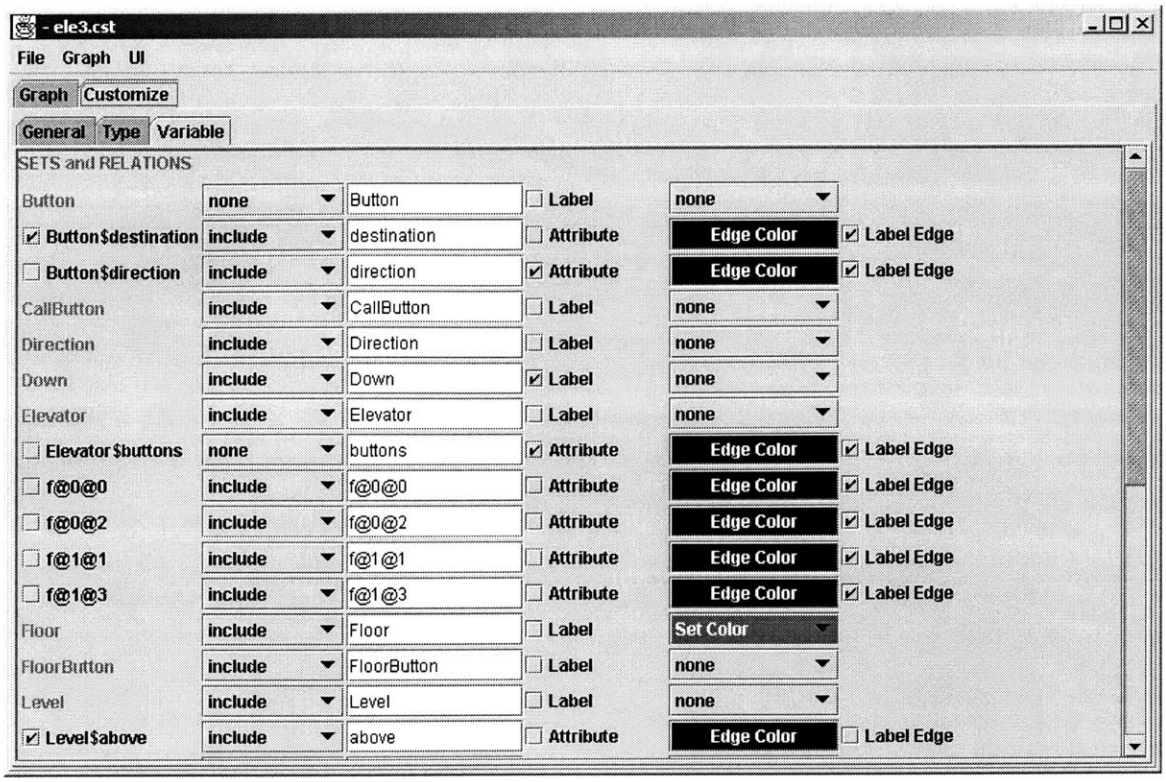

Figure A-3: Relation Customization Panel

Visibility Appearance Layout

Selects a user interface that separates the customization features by Node Visibility, Node Appearance, Edge Visibility, Edge Appearance and Layout.

A.3 Customization Features

This is a more detailed description and behavior of the features in the system.

A.3.1 General

These customizations apply to the entire graph.

View Unconnected Nodes (Focus)

This is a post filter on the nodes of the graph. All nodes without edges going in or out of it are hidden. When this is on, all visible nodes are displayed.

Save Dot Files

When this is on, intermediate dot files are saved in the alloy temporary directory. When this is off, intermediate files are not saved.

Create New Customization on New Model

When this is on and a new model is loaded and visualized a new customization will be made. A new model is defined by a model with a different name. The visualization tool will prompt the user to save the current customization if it has been modified since last saved. When this is off, the updating behavior of the customization from loading a new model will be the same as rebuilding the same model. The old customization will remain and are extended by new variables. Non relevant customizations to the current model will be removed.

Merge Bidirectional Arrows (Focus)

When this is on and there are two edges from the same relation between the same two nodes but going in opposite directions, a new single edge replaces the two edges. The new edge will have bidirectional arrows. When this is off, both arrows are displayed as normal.

Use Hyper Arcs for Hyper Edges (Higher-Arity)

When this is on, hyper edges are represented by creating a new node labeled by the relation Name. Then an arrow is drawn from the relation node to each node of the hyper edge. Each of these arrows is labeled with the index of the node in the edge (i.e. if atom was first in the tuple, it would be labeled with a 0). When this is off, for each edge with more than two nodes, an indexed edge is used instead. For an edge with nodes (n1, n2, ... nm), an indexed edge is drawn from n(m-1) to nm. The edge will be labeled with relationName[n1][n2]... [n(m-2)].

Rank Direction is Top Bottom (Layout)

This is also labeled as Same Rank in Same Horizontal Level since types of the same rank are in the same horizontal plane. The the ranking goes from top to bottom. Thus, minimum rank is at the top, maximum rank is at the bottom. When this is off, all types of the same rank are in the same vertical plane and the ranking goes from left to right.

Use Set Labels in Attributes (Layout)

When this is on, set labels also appear in node attributes. When this is off, they do not appear.

Font size (Layout)

Changes the font size of the components.

A.3.2 Type

Visibility (Focus)

This sets the visibility of a type. When the visibility of a type is turned off, all atoms of that type are no longer visible. We call this exclusion. All customizations regarding that type are also ignored. This affects not only the atoms themselves but each excluded type of a relation is removed entirely. Thus, ternary relations may become binary relations and binary relations may become sets. The available customization options for a relation is changed to reflect this.

Color (Differentiation)

This sets the color of a type. Each atom appearing on the graph of this type will be shaded with the selected color. The text of each of these nodes will automatically be changed to a color that makes the text more readable.

Shape (Differentiation)

This sets of shape of a type. Each atom appearing on the graph of this type will have the selected shape. There is also a special shape called record where the node appears as a box. If the node has attributes, the box will be split up into two sections, the name and labels will be displayed in the upper section, attributes will be displayed in the lower section. This is merely to provide a clearer visual separation between the nodes properties and what it "contains."

Name (Differentiation)

The label name of the type can be changed from the default value. Changing the type should change all instances where atoms of that type are mentioned and should replace where the type text appears in the original atom name. For example, if the type "file" has the name label "f," then the atoms "file_1" and "file_2" should be changed to "f_1" and "f_2" respectively. If the original atom name does not contain a "_" followed by a number then the original atom name is used and this setting does not affect it.

Label (Differentiation)

The name label for each node of a type can be turned on and off. By default this is on.

Projection (Higher-Arity)

The graph can be projected on any number of types. The user then selects the projected atom for each projected type. This splits the graph into many subgraphs, one for each combination of atoms. In each subgraph, all relations containing the projected type becomes relations without the projected type. Thus, no nodes of that type would ever appear in the graph. For example, suppose there are types A, B and C in the instance, with relations of types A->B, A->C->B, C->B->B, and B. Projecting on B would result in the relations of types A, A->C, and C. The relation

of type B would completely disappear. After the types to be projected have been chosen, the user can then choose the atom of each type that the instance is to be projected on. A different graph will be created for each combination of atoms.

To determine if a relation belongs in a certain graph, the program follows these rules. If a relation originally (before any projection) did not have any of the projected types, the relation is displayed. If a relation has at least one projected type, then we consider each tuple of the relation separately. A tuple is in the graph if for each projected type that the tuple has, it contains at least once the atom that was selected for the projected type. This means that if the tuple contains a projected type twice, if either of the atoms of that type in the tuple is the same as the projected atom, the tuple would be displayed in the current graph.

Example: Suppose you had the relation A->A->B and B->C projected on A, and atom A1 selected. Then tuple A1->A2->B1 and tuple A1->A1->B1 would be in the relation. Tuple A2->A3->B1 would not be in the relation. All relations of type B->C would be in the relation.

Rank (Layout)

Sets the rank of a type (i.e. all atoms of a certain type will have the selected rank). The available options are *same*, *min*, *max*, *source*, *sink*. Same rank nodes are visualized on the graph on the same horizontal level or vertical level depending on rank direction. Minimum rank nodes are placed on the same rank at the leftmost or topmost of the graph depending on rank direction. Source nodes are the same as min rank nodes except that only source or min rank nodes can appear on the same rank as a source node. Max and sink have same behavior except on the maximum rank.

A.3.3 Relation

Options change depending on the arity of the relation.

Node Visibility (Focus)

For each relation, one can select *include*, *none*, or *exclude*. A node A appears in the graph if the type of A has not been excluded and there is some tuple of an included relation that contains A, and there is no tuple of an excluded relation that contains A. Hiding unconnected nodes could further hide nodes that were visible after this setting has been applied.

Edge Visibility (Focus)

The visibility of an edge for a relation can be toggled on or off. When an edge is turned off, the edge is not displayed in any way, as an indexed edge, hyper arcs, or as a normal binary edges.

Node Color (Differentiation)

Sets the color of a unary relation. All nodes of atoms in that unary relation will be shaded with the selected color. Atoms that are part of two or more unary relations with conflicting colors will have undetermined color.

Edge Color (Differentiation)

Sets the color of the edges for each binary or higher-arity relation. The edges will have the selected color.

Attribute (Higher-Arity)

Sets a binary or higher-arity relation to be an attribute, displayed as text, of the first component of the relation. The text contains the name of the relation followed by each tuple of that relation without the first component. The components of each tuple will be separated by "->". For example, the tuple, (A,B,C) will be of the form B->C in node A. If the type name labels are turned off for a particular type, the atom names do not appear in the attribute either. Set labels are not repeated if two or more nodes have the same set label and the individual atoms are not labeled.

Index On (Higher-Arity)

With this feature, the user can choose to index the edge on any of the types in the relation. This is only available for ternary edges and above. Each time a type is selected to be indexed on, the arity of the relation goes down by one. Thus a ternary edge would now be represented as a binary edge. For each tuple, all edge label names are appended with the name of the atom of the type that was indexed. For example, suppose there is a relation REL: A->B->C->D and we index on C and c1 was an atom of type C. A hyper arc would be created (if global indexing for edges was turned off) and the node representing the edge would be labeled REL[c1] if c1 was an atom in this edge. If we further indexed on D and d1 was an atom of type D, we would get binary edges of the sort, REL[c1][d1].

Edge Label (Differentiation)

Turns on and off the label for the edges of relation. The color of the text is the same as the color of the edge. If this relation is also an attribute, there will be no edges to be labeled, and thus the setting is ignored.

Label (Differentiation)

For unary relations, the nodes in the unary relation can be labeled with the name of the unary relation. This feature turns on and off that label.

Name (Differentiation)

Sets the name of the relation for when it is used for set labels, edge labels or attributes.

A.4 Strategies and Hints on Using the Visualization Tool

In customizing a graph, there are some common strategies that might be helpful in more quickly arriving at a good visualization.

- The user should start by eliminating as much unnecessary clutter as possible. This can be done by first eliminating any relations or types that are not relevant. One can eliminating relations most easily by turning off the edges for these relations. Irrelevant types can be removed by excluding the type. Removed types are completely removed from the model. Not only are they no longer visible, but they could lower the arity of a relation that contains that type. An example of an irrelevant type is the Ord parameterized type that is used to order elements. Finally, if there is a suitable type to be projected on, the projection should be done next.

- After some of the clutter has been removed and the graph is beginning to take shape, one should probably next consider to make some relations into attributes and further removes the clutter. Good candidates for this are relations that can be thought of as containment such as names and properties of an object. For example, names of files or entries of a routing table are good candidates to set as an attribute. Relations whose incident nodes connect many other nodes are probably not good candidates.

- If there are still some higher-arity relations, one may consider indexing on them instead of just using the default hyper-arcs. Again the purpose of this is to reduce clutter and to represent the relation in a clearer way since indexed edges are represented by fewer edges than hyper arcs.

- After this main reduction has been done, the next thing that the user might consider is to remove nodes that is not currently relevant to the analysis. This can be done using the relation inclusion/exclusion. Normally, one would just toggle between inclusion and none of a relation. Basically, one would use include if one is interested in displaying the nodes in a relation, and none if one is not. But since a node exists if any relation includes it, sometimes it may be faster if there is a predefined set with all the nodes that a user may want to exclude. For example, if in a file system you want to see directories and not files, one could choose to exclude the set of files. Another quick way to do hide undesirable

nodes is to simply hide all unconnected nodes.

- Most of the hard work has been done up to this point. After the visibility of nodes and edges have been set, the next thing to do is probably to try to differentiate between the nodes and edges of different types by changing the color, shape, or labels. This part is mostly just getting the graph to look nice. One may also consider changing the font size at this point if one so desires.

- Finally, one should consider modifying the layout after one has fixed the size and shape of nodes and the edges that exist. This is done last because changing the existence or sizes of nodes greatly affects the layout.

Appendix B

Elevator Example

B.1 Alloy Model of Elevator System

```
open std/ord

sig Direction{}

static part sig Up, Down, Stopped extends Direction{}

sig Elevator{
   buttons: set FloorButton,
}
{
   //doesn't have two buttons going to same place
   no disj b1, b2: buttons | b1.destination = b2.destination
   //has buttons for all floors
   buttons.destination = Floor
}

sig Level{
   above, below: option Level
}

fact{
   Ord[Level].prev = Level$above
   Ord[Level].next = Level$below
   //the top and bottom levels are floors
   Ord[Level].last in Floor
   Ord[Level].first in Floor
}
```

```
//movement
sig Floor extends Level{
   callButtons: set CallButton

}
{
   //there is always some levels in between floors that are not floors
   some above => above !in Floor
   some below => below !in Floor

   this != Ord[Level].first <=> one b:callButtons | b.direction = Down
   this != Ord[Level].last  <=> one b:callButtons | b.direction = Up
   no disj b1, b2: callButtons | b1.direction = b2.direction
}

sig Button{}

disj sig CallButton extends Button{
   direction: Up + Down
}

disj sig FloorButton extends Button{
   destination: Floor
}

sig State {
   pressed: set Button,
   at: Elevator -> !Level,
   movement: Elevator -> !Direction,
   serviceDir: Elevator -> ?(Up + Down)
}

fact{
   //can't share buttons
   all disj f1, f2: Floor |
      no f1.callButtons & f2.callButtons
   all disj e1, e2: Elevator |
      no e1.buttons & e2.buttons
   //all buttons in some floor or elevator
   Button in Floor.callButtons + Elevator.buttons
}

fun init(s: State){
   //make sure initial state is valid
   all e: Elevator{
      s.serviceDir[e] in Up =>
          some f: s.at[e].^below | floorRequested(s, f, e)
```

```
      s.serviceDir[e] in Down =>
          some f: s.at[e].^above | floorRequested(s, f, e)
   }
}

fun floorRequested(s:State, f:Floor, e:Elevator){

   some (s.pressed & e.buttons).destination & f ||
   s.serviceDir[e] in (s.pressed & f.callButtons).direction
}

fun policy (s:State){
   all e: Elevator{
      //if you're moving then you must be servicing in that direction
      s.movement[e] in (Up + Down) => s.movement[e] = s.serviceDir[e]
      //can't stop in between floors
      s.at[e] !in Floor => s.movement[e] in (Up + Down)
   }
}

fun physics(s, s':State){
   all e :Elevator{
      //can't skip levels
      s'.at[e] in (s.at[e] + s.at[e].above + s.at[e].below)
      //if you're moving up, the new level must be above the old level
      s.movement[e] = Up => s.at[e] = s'.at[e].above
      //if you're moving down, the new level must be below the old level
      s.movement[e] = Down => s.at[e] = s'.at[e].below
      //if stopped, don't move next state
      s.movement[e] = Stopped => s.at[e] = s'.at[e]
      //can't change direction between floors
      s'.at[e] !in Floor => s.movement[e] = s'.movement[e]
   }

}



fun policies(s, s':State){
   all e: Elevator{
      //outstanding request iff elevator is servicing in some direction
      (some (s.pressed & e.buttons) || some (s.pressed & CallButton)) <=>
          some s.serviceDir[e]
               //change direction iff there are no more floors requested
               //in direction of service
      s.serviceDir[e] in Up && s'.serviceDir[e] !in Up <=>
          no f: s.at[e].^below | floorRequested(s',f,e)
```

```
      s.serviceDir[e] in Down && s'.serviceDir[e] !in Down <=>
          no f: s.at[e].^above | floorRequested(s',f,e)

      //no skipping request at floor
      //if elevator is at floor of one of requests, it must stop
      floorRequested(s, s.at[e], e) => s.movement[e] = Stopped

      //if servicing and buttons in current floor not pressed, must move
      !floorRequested(s, s.at[e], e) && some s.serviceDir[e] =>
          s.movement[e] in (Up + Down)
   }
}

fun buttonUpdate(s, s': State){


//cancel buttons if an elevator has stopped on that floor for floor buttons
//or if it has stopped and is servicing that direction for callButtons
   s.pressed -
   {b:FloorButton | some e:Elevator{
      s.at[e] = b.destination  &&
      s.movement[e] = Stopped }
   } -
   {b:CallButton | some e:Elevator{
      s.movement[e] = Stopped &&
      b in s.at[e].callButtons &&
      b.direction = s.serviceDir[e]}
   }
   = s'.pressed

   //can be modified to allow new buttons to be pressed
}

fun Go (){
   init(Ord[State].first)
   all s, s':State {
      s in OrdPrev(s') => physics(s, s')
      s in OrdPrev(s') => policies(s, s')
      s in OrdPrev(s') => buttonUpdate(s, s')
   }
   all s: State {
      policy(s)
   }
}


fact stuff{
```

```
   Go()
   one Elevator
   #Ord[State].first.pressed > 3
   #(State.at[Elevator]) > 2
}

run Go for 5 but 10 Button, 6 State
```

B.2 Solution output

This is an output from the Alloy Analyzer describing a solution. It is a list of relations with their types and their tuples.

```
============================================================
----------------------------
relation: elevator/Level (type: Level)
{ (Level_0) (Level_1) (Level_2) (Level_3) (Level_4)  }
----------------------------
relation: Level$above (type: Level -> Level)
{ (Level_1 Level_0) (Level_2 Level_1) (Level_3 Level_2) (Level_4 Level_3)  }
----------------------------
relation: Level$below (type: Level -> Level)
{ (Level_0 Level_1) (Level_1 Level_2) (Level_2 Level_3) (Level_3 Level_4)  }
----------------------------
relation: elevator/Floor (type: Level)
{ (Level_0) (Level_2) (Level_4)  }
----------------------------
relation: Level$callButtons (type: Level -> Button)
{ (Level_0 Button_05) (Level_2 Button_04) (Level_2 Button_06)
  (Level_4 Button_03)  }
----------------------------
relation: std/ord/Ord[elevator/State] (type: Ord[State])
{ (Ord[State]_0)  }
----------------------------
relation: Ord[State]$next (type: Ord[State] -> State -> State)
{ (Ord[State]_0 State_0 State_1) (Ord[State]_0 State_1 State_2)
  (Ord[State]_0 State_2 State_3) (Ord[State]_0 State_3 State_4)
  (Ord[State]_0 State_4 State_5)  }
----------------------------
relation: Ord[State]$prev (type: Ord[State] -> State -> State)
{ (Ord[State]_0 State_1 State_0) (Ord[State]_0 State_2 State_1)
  (Ord[State]_0 State_3 State_2) (Ord[State]_0 State_4 State_3)
  (Ord[State]_0 State_5 State_4)  }
----------------------------
relation: Ord[State]$last (type: Ord[State] -> State)
{ (Ord[State]_0 State_5)  }
----------------------------
```

```
relation: Ord[State]$first (type: Ord[State] -> State)
{ (Ord[State]_0 State_0)  }
---------------------------
relation: elevator/Elevator (type: Elevator)
{ (Elevator_4)  }
---------------------------
relation: Elevator$buttons (type: Elevator -> Button)
{ (Elevator_4 Button_07) (Elevator_4 Button_08) (Elevator_4 Button_09)  }
---------------------------
relation: std/ord/Ord[elevator/Level] (type: Ord[Level])
{ (Ord[Level]_0)  }
---------------------------
relation: Ord[Level]$next (type: Ord[Level] -> Level -> Level)
{ (Ord[Level]_0 Level_0 Level_1) (Ord[Level]_0 Level_1 Level_2)
  (Ord[Level]_0 Level_2 Level_3) (Ord[Level]_0 Level_3 Level_4)  }
---------------------------
relation: Ord[Level]$prev (type: Ord[Level] -> Level -> Level)
{ (Ord[Level]_0 Level_1 Level_0) (Ord[Level]_0 Level_2 Level_1)
  (Ord[Level]_0 Level_3 Level_2) (Ord[Level]_0 Level_4 Level_3)  }
---------------------------
relation: Ord[Level]$last (type: Ord[Level] -> Level)
{ (Ord[Level]_0 Level_4)  }
---------------------------
relation: Ord[Level]$first (type: Ord[Level] -> Level)
{ (Ord[Level]_0 Level_0)  }
---------------------------
relation: elevator/Direction (type: Direction)
{ (Direction_1) (Direction_3) (Direction_4)  }
---------------------------
relation: elevator/Stopped (type: Direction)
{ (Direction_1)  }
---------------------------
relation: elevator/Down (type: Direction)
{ (Direction_3)  }
---------------------------
relation: elevator/Up (type: Direction)
{ (Direction_4)  }
---------------------------
relation: elevator/Button (type: Button)
{ (Button_03) (Button_04) (Button_05) (Button_06) (Button_07) (Button_08)
  (Button_09)  }
---------------------------
relation: elevator/FloorButton (type: Button)
{ (Button_07) (Button_08) (Button_09)  }
---------------------------
relation: Button$destination (type: Button -> Level)
{ (Button_07 Level_2) (Button_08 Level_4) (Button_09 Level_0)  }
---------------------------
relation: elevator/CallButton (type: Button)
{ (Button_03) (Button_04) (Button_05) (Button_06)  }
---------------------------
relation: Button$direction (type: Button -> Direction)
{ (Button_03 Direction_3) (Button_04 Direction_3) (Button_05 Direction_4)
  (Button_06 Direction_4)  }
```

```
---------------------------
relation: elevator/State (type: State)
{ (State_0) (State_1) (State_2) (State_3) (State_4) (State_5)  }
---------------------------
relation: State$at (type: State -> Elevator -> Level)
{ (State_0 Elevator_4 Level_1) (State_1 Elevator_4 Level_2)
  (State_2 Elevator_4 Level_2) (State_3 Elevator_4 Level_3)
  (State_4 Elevator_4 Level_4) (State_5 Elevator_4 Level_4)  }
---------------------------
relation: State$movement (type: State -> Elevator -> Direction)
{ (State_0 Elevator_4 Direction_4) (State_1 Elevator_4 Direction_1)
  (State_2 Elevator_4 Direction_4) (State_3 Elevator_4 Direction_4)
  (State_4 Elevator_4 Direction_1) (State_5 Elevator_4 Direction_1) }
---------------------------
relation: State$pressed (type: State -> Button)
{ (State_0 Button_06) (State_0 Button_07) (State_0 Button_08)
  (State_0 Button_09) (State_1 Button_06) (State_1 Button_07)
  (State_1 Button_08) (State_1 Button_09) (State_2 Button_08)
  (State_2 Button_09) (State_3 Button_08) (State_3 Button_09)
  (State_4 Button_08) (State_4 Button_09) (State_5 Button_09) }
---------------------------
relation: State$serviceDir (type: State -> Elevator -> Direction)
{ (State_0 Elevator_4 Direction_4) (State_1 Elevator_4 Direction_4)
  (State_2 Elevator_4 Direction_4) (State_3 Elevator_4 Direction_4)
  (State_4 Elevator_4 Direction_4) (State_5 Elevator_4 Direction_3)  }
---------------------------
relation: Elevator$elevator/init@f@0@0 (type: Elevator -> Level)
{ (Elevator_4 Level_2)  }
---------------------------
relation: Elevator$elevator/init@f@1@1 (type: Elevator -> Level)
{ (Elevator_4 Level_0)  }
---------------------------
relation: Elevator$elevator/init@f@0@2 (type: Elevator -> Level)
{ (Elevator_4 Level_2)  }
---------------------------
relation: Elevator$elevator/init@f@1@3 (type: Elevator -> Level)
{ (Elevator_4 Level_0)  }
============================================================
```

B.3 Elevator Visualizations

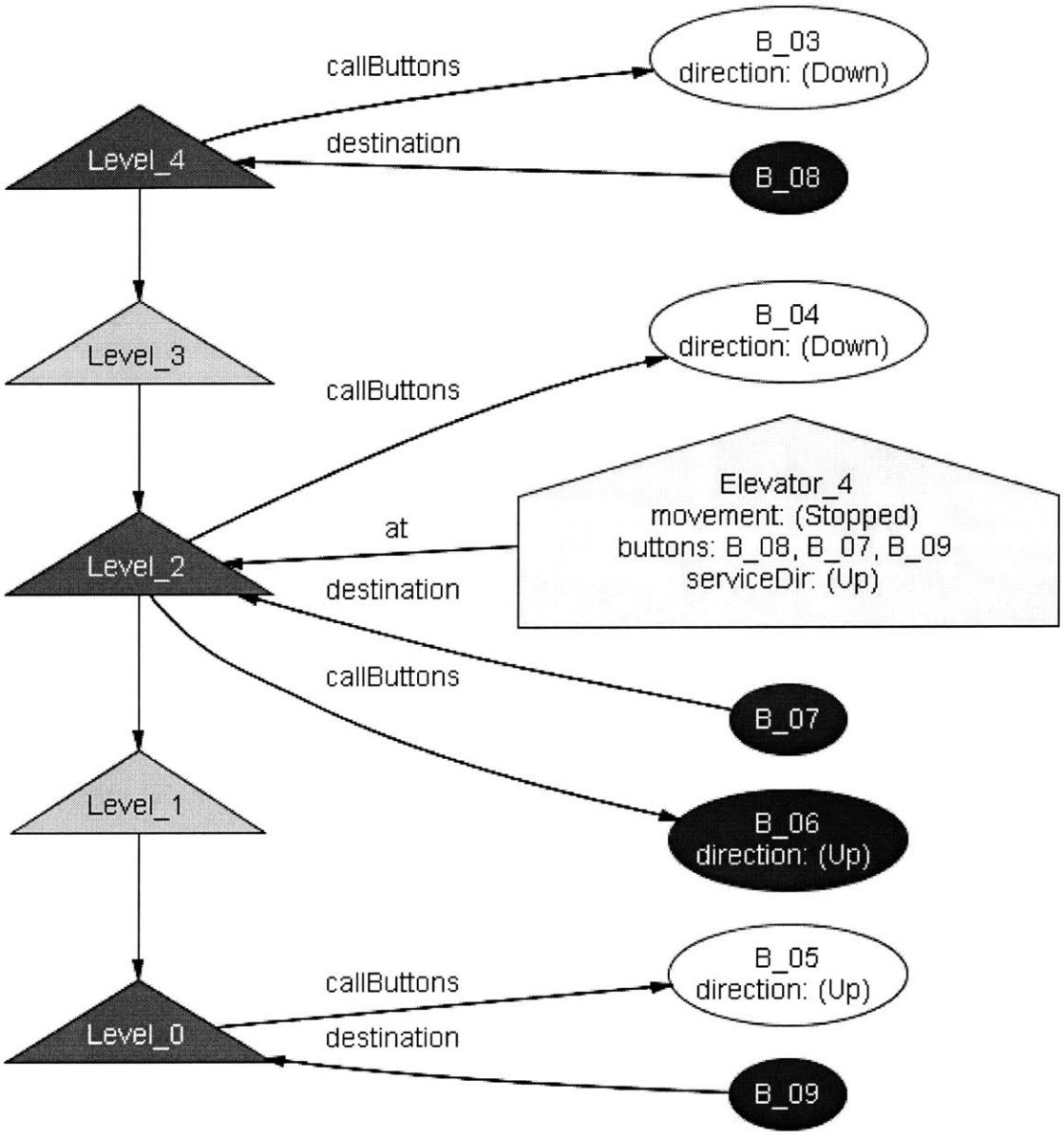

Figure B-1: Final Elevator Diagram projected on state 1
Pressed buttons are in black.
The elevator has moved up one level and stopped at Level_2.

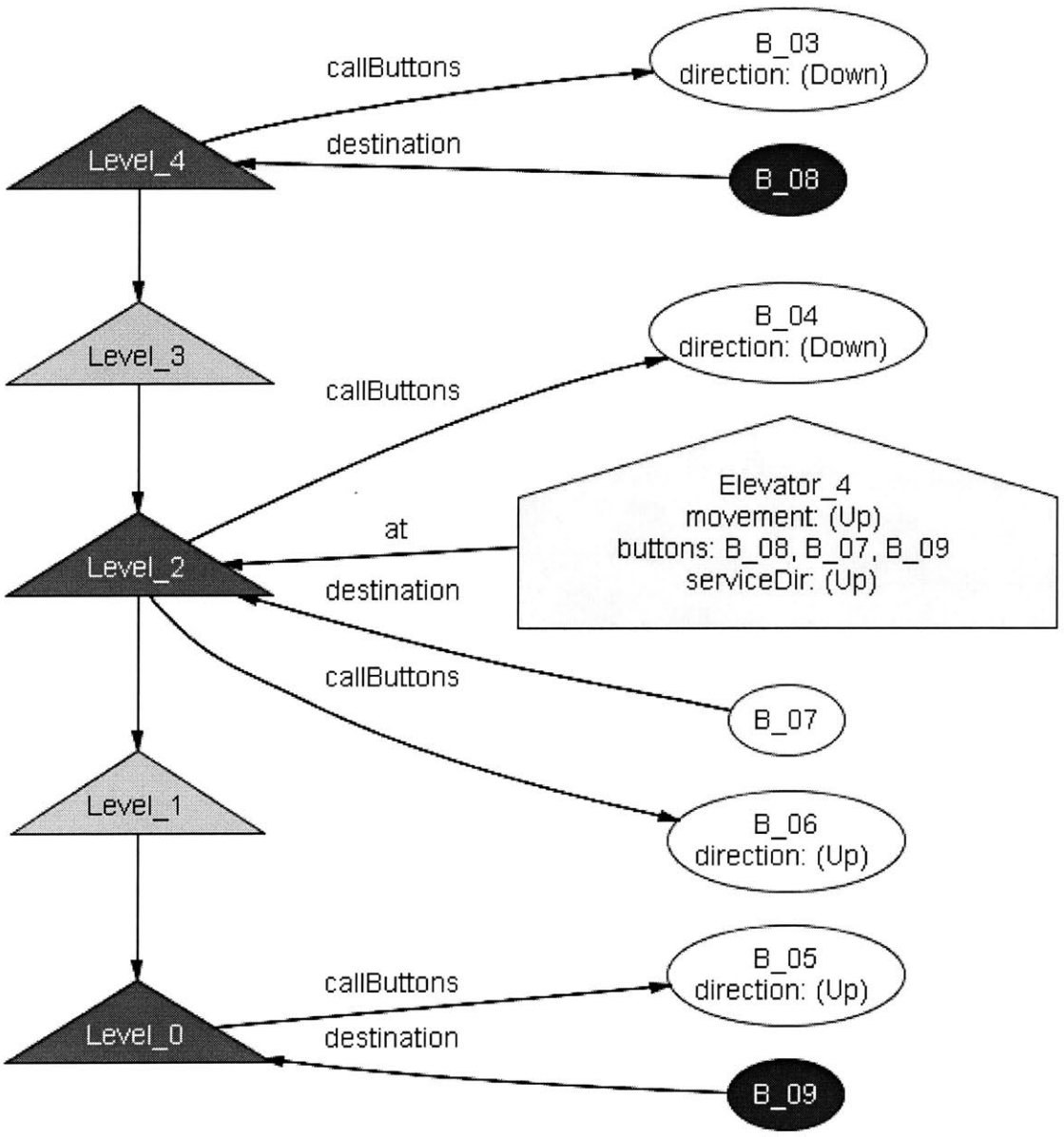

Figure B-2: Final Elevator Diagram projected on state 2
Pressed buttons are in black.
The button for Level_2 and the call button on Level_2 are no longer pressed since these requests have been serviced.

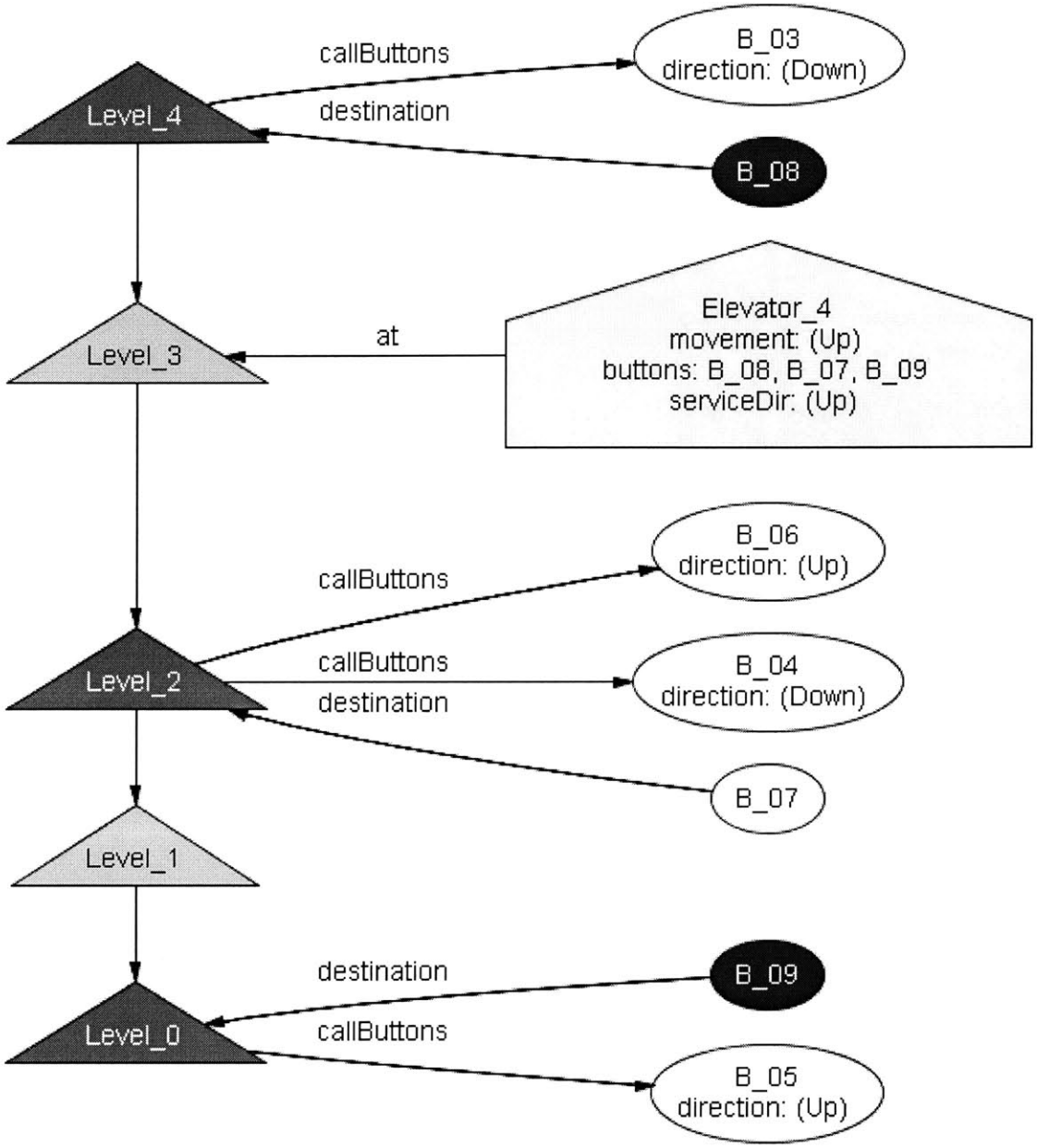

Figure B-3: Final Elevator Diagram projected on state 3
Pressed buttons are in black.
The elevator now resumes going up and is at Level_3.

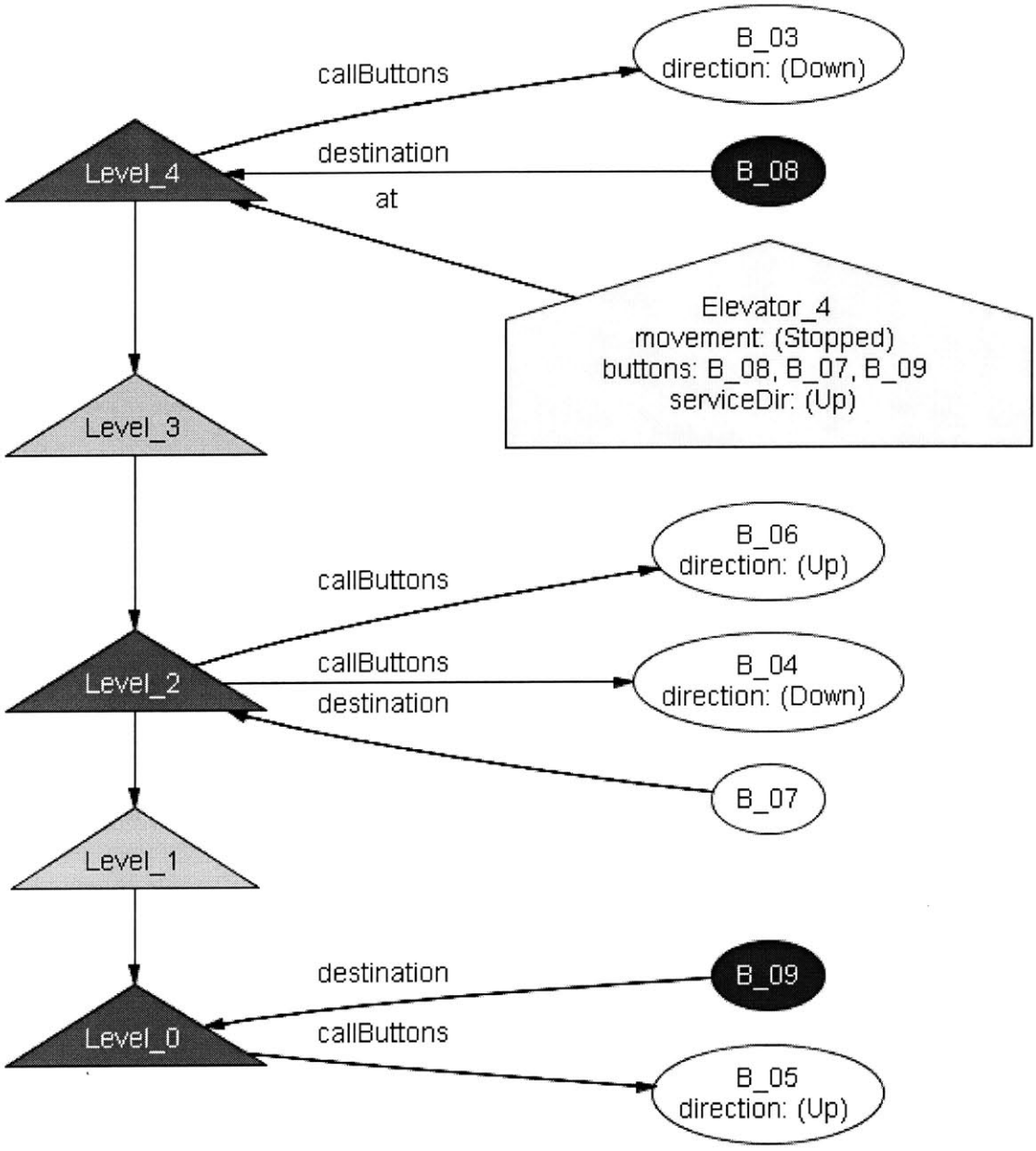

Figure B-4: Final Elevator Diagram projected on state 4
Pressed buttons are in black.
The elevator has reached Level_4 and is stopped there servicing requests on that floor.

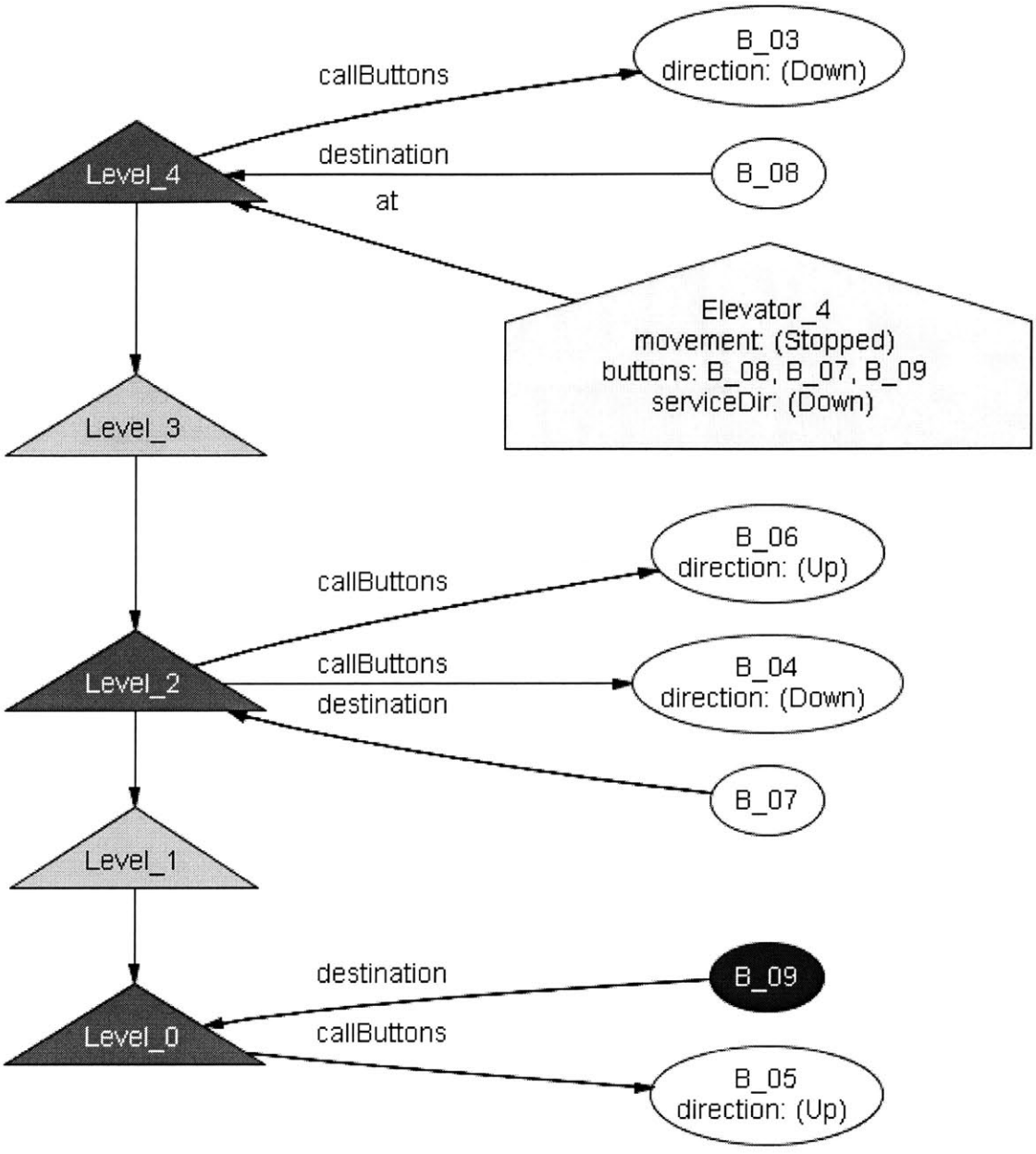

Figure B-5: Final Elevator Diagram projected on state 5

Pressed buttons are in black.

The button for Level_4 is no longer pressed after being serviced, and the elevator has now changed its service direction and is ready to head down to service the last request.

Bibliography

[1] AiSee: A Tool that Automatically Calculates a Customizable Layout of Graphs Specified in GDL (graph description language). http://www.absint.com/aisee/.

[2] Joshua Bloch. *Effective Java Programming Language Guide*, chapter 5. Addison-Wesley, 2001.

[3] daVinci: X-Window Visualization Tool for Drawing Directed Graphs Automatically in High Quality. http://www.informatik.uni-bremen.de/daVinci/.

[4] Erich Gamma, Richard Helm, Ralph Johson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, chapter 3. Addison-Wesley, 2000.

[5] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software - Practice and Experience*, 1999.

[6] GraphViz: Open Source Graph Drawing Software Developed at AT&T Laboratories. http://www.research.att.com/sw/tools/graphviz/.

[7] Ivan Herman, Guy Melancon, and Scott Marshall. Graph visualization and navigation in information visualization: a survey. *IEEE Transactions on Visualization and Computer Graphics*, 2000.

[8] Eleftherios Koutsofios and Stephen North. *Drawing graphs with dot.* AT&T Bell Laboratories, Murray Hill, NJ, February 2002. Dot User's Manual.

[9] Object Constraint Language (OCL): The Expression Language for UML. http://www-3.ibm.com/software/ad/library/standards/ocl.html.

[10] Michael Reed and Dan Heller. Online library of visualization environments. http://otal.umd.edu/Olive/.

[11] J. Michael Spivey. An introduction to z and formal specifications. *Software Engineering Journal*, 4(1):40–50, 1989.

[12] Tom Sawyer: High-quality Scalable and Incremental Graph Management, Layout, and Visualization. http://www.tomsawyer.com/.

[13] Walrus: A Tool for Interactively Visualizing Large Directed Graphs in Three-dimensional Space. http://www.caida.org/tools/visualization/walrus/.

[14] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.

[15] Jon Whitney. Description and analysis of central registry, a pattern for modular implicit invocation. Master's thesis, Massachusetts Institute of Technology, 2002.