Checking Semantic Usage of Frameworks

Ciera Jaspan
Institute for Software Research
Carnegie Mellon University
Pittsburgh, PA 15213
cchristo@cs.cmu.edu

Jonathan Aldrich
Institute for Software Research
Carnegie Mellon University
Pittsburgh, PA 15213
jonathan.aldrich@cs.cmu.edu

ABSTRACT
Software frameworks are difficult for plugin developers to use, even when they are well designed and documented. Some of these difficulties stem from the many constraints that frameworks impose on plugin code. These constraints might restrict operations from being called on certain objects, or they might restrict how long an object is available. Additionally, the constraints are relative to the current context of the plugin, and they can involve multiple, interacting framework objects. This paper proposes a lightweight specification system and analysis to check plugins from a semantic perspective, rather than a purely structural view.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: Software/Program Verification—*Class invariants*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Frameworks*

General Terms
Design,Verification

Keywords
framework constraint, object relationship

1. INTRODUCTION
Software frameworks allow developers to reuse not just an implementation, but a complete architecture and design. By abstracting and reusing the architecture, the programmer also receives the benefit of architectural solutions to difficult issues such as scalability, concurrency, security, and performance. However, there is a cost to using a framework; software frameworks are complex and difficult to learn [1]. *Plugins* that use frameworks must adhere to many underlying constraints of the framework. These constraints typically concern multiple objects that interact together, and they may change based upon the state of the plugin or framework.

It is easy for an application developer to unknowingly break a constraint, and it is difficult for application developers to determine what occurred when a constraint breaks. The framework documentation may explain the constraint, but the plugin developer would only discover the documentation if he knew which constraint he broke. Unfortunately, many framework constraints cause errors which either do not occur within the plugin code, or they cause unusual runtime behavior that occurs long after the constraint was first broken. For this reason, even experienced developers have difficulty keeping track of all the constraints that they must comply with.

We propose to ease this burden by creating a specification and analysis to discover mismatches between the plugin code and the declared constraints of the framework. Our proposal is guided by the following principles:

1. *No up-front effort for the plugin developer* The plugin developers should not have to make any additional effort to use the framework. In particular, they do not add any specifications to their plugin code.

2. *Minimal effort for the framework developer* The framework developer will have to specify how to use the framework and what constraints exist. This should not require a complete specification of the framework's internals.

3. *Localized errors* Frameworks require developers to use many objects from different places in the framework. The constraints are inherently distributed across all of these objects. In order for this solution to be adoptable, errors from incorrect usage cannot simply state that there was an error in a constraint; errors should be specific to the plugin code and should give a localized description.

4. *Modularity of constraints* Framework constraints work across many objects, and some objects are governed by many constraints. It should be possible to specify each constraint separately. Framework developers should not be forced to specify the entire framework, or even an entire class. Since these constraints must be modular, it also follows that they cannot interfere with each other during enforcement. That is, if the framework developer adds or removes a constraint specification, it should only add or remove plugin errors caused

by that constraint. The results from other constraints should not be affected.

This work contains three contributions. First, we propose a way to capture the semantic knowledge of the framework without the plugin developer adding any specifications to his code. Second, we propose a mechanism for the framework developer to specify the framework constraints in a modular and incremental manner. Finally, we have proposed a local analysis that will use the relationship and constraint specifications to discover defects in the plugin code. This analysis will also provide localized, understandable errors to the plugin developer.

2. EXAMPLE CONSTRAINTS

To better understand framework constraints, we will explore two examples from the ASP.NET framework. We will use these examples to motivate a specification language for semantic constraints and an analysis of plugin code. The first example, DropDownList Selection, is from the author's own experiences with ASP.NET. The other example, Login Status, was mined from the ASP.NET help forums [2].

ASP.NET is a web application framework. Developers can create web pages that work together to form a complete web application. There are two parts to every web page in ASP.NET, an ASPX file and a code-behind file. The ASPX file represents the user interface for the web page and uses an XML-based language. Developers may use tags specific to ASP.NET web controls, or they may use HTML tags. The code-behind file stores the event handling code for the page. This file is typically written in either C# or VB.NET, and it interfaces directly with the ASP.NET framework. A .NET server will read the ASPX file and run the code-behind file to produce HTML to transmit back to the client.

2.1 DropDownList Selection

The ASP.NET framework allows developers to create web pages with user interface controls on them. These controls can be manipulated programatically through the callbacks provided by the framework. Developers can respond to control events, add and remove controls, and change the state of controls.

One task that a developer might want to do is programmatically change the selection of a drop down list. The ASP.NET framework provides us with the relevant pieces as shown in Figure 1. [1] Notice that if we want to change the selection of a `DropDownList` (or any other derived `ListControl`), we have to access the individual `ListItems` through the `ListItemCollection` and change the selection using `SetSelected`. Based on this information, a developer might naively change the selection as shown in Program 1. Our expectation is that the framework will see that we have selected a new item, and it will change the selection accordingly.

This code breaks an important framework constraint. A `DropDownList` must have one and only one item selected.

[1] To make this code more accessible to those unfamiliar with C#, I am using traditional getter/setter syntax rather than properties.

Program 1 Incorrect selection for a DropDownList

```
DropDownList list;

private void Page_Load(object sender, EventArgs e)
{
   string searchVal = ...
   ListItemCollection items;
   ListItem newSel;

   items = list.GetItems();
   newSel = items.FindByValue(searchVal);
   newSel.SetSelected(true);
}
```

If this code is run, an interesting error occurs, as shown in Figure 2. The error message clearly describes the problem; a DropDownList had more than one item selected. An experienced developer will realize that setting the selected item did not deselect the existing one, though an inexperienced developer might be confused because he did not select multiple items.

The stack trace is more interesting though; it does not point to the code where we made the selection. In fact, the entire stack trace is from framework code; there is no plugin code referenced at all. The program control flowed through the plugin developer's code and then returned to the framework before the error was discovered. The program flow could go back and forth several times before finally reaching the check that triggered the error. Since we don't know exactly where the problem occurred or even what object it occurred on, the developer must search his code by hand to determine where the erroneous selection occurred.

Program 2 Correctly selecting an item using the API

```
DropDownList list;

private void Page_Load(object sender, EventArgs e)
{
   string searchVal = ...
   ListItemCollection items;
   ListItem newSel, oldSel;

   oldSel = list.GetSelectedItem();
   oldSel.SetSelected(false);

   items = list.GetItems();
   newSel = items.FindByValue(searchVal);
   newSel.SetSelected(true);
}
```

Program 2 shows correct code for this task. We must deselect the current selection before making a new selection. We must also perform the tasks in this order, otherwise, `GetSelectedItem()` may return the newly selected item, rather than the old selected item.

There is also unusual behavior when no item is selected. If there is no item selected by the time the `DropDownList` is rendered, the framework will be forced to choose an item to

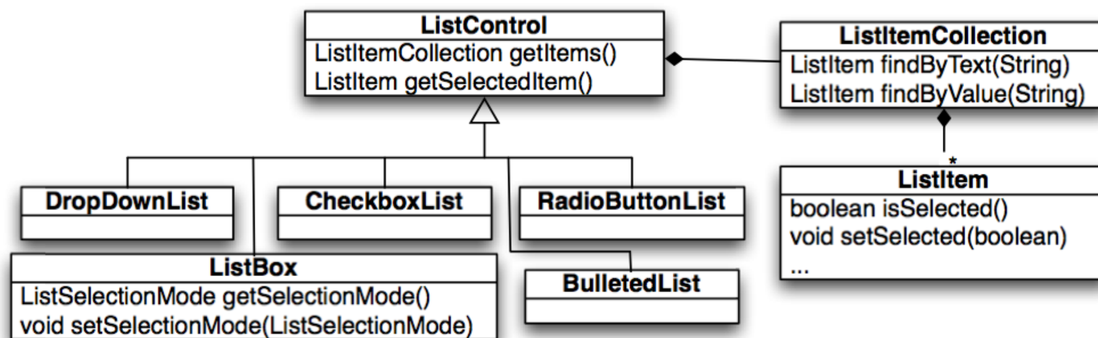

Figure 1: ListControl Class Diagram

Cannot have multiple items selected in a DropDownList.

Stack Trace:

```
[HttpException (0x80004005): Cannot have multiple items selected in a DropDownList.]
   System.Web.UI.WebControls.DropDownList.VerifyMultiSelect() +133
   System.Web.UI.WebControls.ListControl.RenderContents(HtmlTextWriter writer) +206
   System.Web.UI.WebControls.WebControl.Render(HtmlTextWriter writer) +43
   System.Web.UI.Control.RenderControlInternal(HtmlTextWriter writer, ControlAdapter adapter) +74
   System.Web.UI.Control.RenderControl(HtmlTextWriter writer, ControlAdapter adapter) +291
```

Figure 2: Error with partial stack trace from ASP.NET

display. By default, this is the first item in the list. In many cases, this is the correct behavior, but it can also signal a mistake by the developer to explicitly select an item. To be safe, a developer should explicitly select after deselecting.

Program 3 Selecting on the wrong DropDownList

```
DropDownList listA;
DropDownList listB;

private void Page_Load(object sender, EventArgs e)
{
   string searchVal = ...
   ListItemCollection items;
   ListItem newSel, oldSel;

   oldSel = listA.GetSelectedItem();
   oldSel.SetSelected(false);

   items = listB.GetItems();
   newSel = items.FindByValue(searchVal);
   newSel.SetSelected(true);
}
```

A user might also make the correct sequence of calls, but on the wrong objects. It is not unusual for there to be multiple, related `DropDownLists`. Program 3 looks very similar to the correct program, but it makes the calls on the wrong objects.

Why didn't the `SetSelected` call just assert if something was already selected? The problem is that `ListItem` does not know about its parent control, so it cannot be responsible for enforcing this constraint. Even if it had a reference to the parent, it would have to change the selection based upon the type of the control. A `DropDownList` required one and only one item to be selected, but a `CheckboxList` allows zero, one, or more items to be selected. Other lists, such as the `ListBox`, might even change their constraint according to how it is being used. The selection constraint is eventually checked by the derived `ListControl`, but only in a much later callback, just before the `DropDownList` is rendered on the web page.

There are alternate designs that avoid this problem. For example, the selection could be controlled by the `ListControl` and the mechanism could change for each derived class, or a separate class could define the selection mechanism for a `ListItemCollection`. Each of these designs have tradeoffs in other places though; they either limit extensibility or add complexity to the design. Without speaking to the designers, it is difficult to guess why they chose this particular set of tradeoffs. Finally, it is possible that the designers simply overlooked this problem. This type of issue is frequently overlooked, but it is difficult to rectify the problem after the framework has become an industry standard.

2.2 Login Status

On the ASP.NET forums, a developer reported that he was attempting to retrieve a `DropDownList` from his page, but his code was throwing a `NullReferenceException` [3]. This page was meant to display some controls if the user is not logged in, but other controls if the user is logged in. The developer's intention was to set up the controls with all their data, and then for the `DropDownList` to display only when a user was logged in. The `LoginView` control provides developers with this functionality by having two states. One state displays when the user is logged in, and one state displays when the user is not logged in.

The developer properly set up a `LoginView` and then attempted to set up a subcontrol with data. The typical way to get a subcontrol is to call `Control.FindControl` with the appropriate name; `FindControl` will return null only if there is no subcontrol with that name. His ASPX file declared the `DropDownList` properly, and it had the correct name. However, he could not access his list in order to fill it with data. An abbreviated version of the ASPX file is in Program 4, and the code snippet causing the error is in Program 5.

Another developer responded to the post and explained this unusual error. The `LoggedInTemplate` allows the developer to declare which controls are shown when the user is logged in. However, until the user is logged in, these controls do not even exist. If the developer wishes to set up data in these controls, he must do so before the control is displayed, but after the user has logged in. This constraint make more sense from a security perspective; we do not want any chance of the control leaking out of the system. The solution proposed was to first check the login status from `Request.IsAuthenticated`, as shown in the corrected Program 6.

The `LoginView` also allows us to show different controls to different users by creating many `LoginTemplate`s and associating each with a group of users. If we also want this functionality, we must check the properties of the logged-in user to determine whether a control is accessible. This adds a great deal of complexity to the plugin, and it is compounded if a user is specified in more than one `LoginTemplate`.

Program 4 ASPX with a `LoginView`

```
   <asp:LoginView ID="LoginScreen" runat="server">
      <AnonymousTemplate>
         You can only set up your account
         when you are logged in.
      </AnonymousTemplate>
      <LoggedInTemplate>
         <h4>Location</h4>
         <asp:DropDownList ID="LocationList"
             runat="server"/>
         <asp:Button ID="ContinueButton"
            runat="server" Text="Continue"/>
      </LoggedInTemplate>
   </asp:LoginView>
```

Program 5 Incorrect way of retrieving controls in a `LoginView`

```
LoginView LoginScreen;

private void Page_Load(object sender, EventArgs e)
{
   DropDownList list = (DropDownList)
     LoginScreen.FindControl("LocationList");
   list.DataSource = ...;
   list.DataBind();
}
```

2.3 Properties of Framework Constraints

In both of these examples, the plugin developers broke an unknown framework constraint. They had used the framework in a way which seemed intuitive because they had a slightly misshapen view of what the framework was doing. Additionally, the problem was not always clear from the runtime error. In the first example, the message might have helped, but the stack trace did not come from the plugin code. In the second example, the stack trace pointed to the correct location, but the error message was a side effect of the real issue. We cannot realistically expect that a plugin developer will understand all of the internals of a framework, so we propose a framework specification that will discover these issues *without* specifications from the plugin developer. The framework developer would specify these constraints in framework code, and an analysis will look for inconsistencies between the specifications and the plugin code.

Program 6 Correct way of retrieving controls in a `LoginView`

```
LoginView LoginScreen;
Request request;

private void Page_Load(object sender, EventArgs e)
{
   if (request.IsAuthenticated()) {
   DropDownList list = (DropDownList)
     LoginScreen.FindControl("LocationList");
      list.DataSource = ...;
      list.DataBind();
   }
}
```

Based on these examples and several others mined from the ASP.NET developer forum, we have identified four interesting properties of framework constraints. These properties are specific to framework constraints and imply a set of challenges that a solution must overcome. Previous work, as discussed in Section 5, does not cover all four of these properties.

Framework constraints involve multiple classes and objects. Unlike library and protocol constraints, which typically involve only one object, framework constraints frequently span across several objects. These objects may know about their fellow objects, but in some cases, they do not know about other objects which they share a constraint with. The DropDownList Selection example had four relevant runtime objects, and the code was split across four classes. In this case, the `DropDownList` was at least indirectly aware of the other objects, but the `ListItem`s had no knowledge about the `DropDownList`, the other `ListItem`s, or even the `ListItemCollection` that they belonged to. In the Login Status example, the `RadioButtonList` was completely unaware of the constraint surrounding it, which template it was in, or even that it was inside of a `LoginView` in the first place.

Framework constraints have semantic properties. They are not only about method naming conventions, which methods to override, or the structural nature of the plugin code. In each of these examples, the developer had to be aware of the order of operations and the *relationships* objects had with each other. The DropDownList Selection example required the developer to deselect an item before selecting a new one.

Additionally, the items had a specific set of relationships: they were both children of the same control, the one being deselected was already selected, and they were items within a `DropDownList`. The Login Status example certainly had structural aspects; the call to `Control.FindControl` had to exist within a particular if-statement. However, there were also many semantic properties. The `Request` object had to be the same `Request` that governed the `LoginView`, and a control with the requested name had to exist within the `LoggedInTemplate`.

Framework constraints are non-local. In both of these examples, the method that triggers the error is defined in a class that has no knowledge of the framework constraint. In the DropDownList Selection, the constraint is specific to the `DropDownList`, but the operations that the constraint covers are in `ListItem`. The Login Status example had a similar problem, though it was across more objects. The constraint is in the `LoginView`, but the `Request` class affected whether the `RadioButtonList` was accessible, and the `getControl()` method is defined in the base class `Control`.

Framework constraints are independent of other framework constraints. The same class may be involved in multiple constraints; for example, the `DropDownList` also constrains which `ListItem`s can be enabled and disabled, and it could even be inside of a `LoginView` itself. However, the constraints operate on the same set of objects and operations. For example, the constraints for selection and enabling both need to know which `ListItem`s are children of the `DropDownList`. Constraints must be allowed to operate on the same objects and operations, but they must be enforced separately. By enforcing them separately, we ensure that adding or removing a framework constraint does not affect what defects other constraints are finding against the plugin code.

3. RELATIONSHIPS AND CONSTRAINTS

We propose two constructs to specify and enforce framework constraints. These constructs, along with a planned static analysis, will allow us to discover defects within plugin code. In addition to handling the specific properties of framework constraints, our proposed constraint specifications are modular, the plugin developer does not add any specifications, and the errors produced from the analysis will point to a specific error in the plugin code. We will use the DropDownList Selection example to motivate this section, though the specifications and analysis have also been completed for the other example.

The first construct we create specifies the *relationships* among objects. Relationships statically describe the dynamic associations between objects. We will statically track these relationships through plugin code to produce a semantic context for every expression. This context will represent the relationships of framework objects before the expression is evaluated.

The second construct, *constraint*, will represent a single framework constraint. This specification combines the semantic information from relationships with syntactic specifications. We define a constraint as a path of code through the plugin, with a specified start and end point. Within these paths, new operations may be available, or old operations may be disabled. Additionally, some objects may not be read outside of this path. This is a handy feature for frameworks that reclaim objects or do not guarantee a returned object to be valid after a certain point.

3.1 Relationships

Relationships are facts that we learn after calling a framework method. We annotate the framework methods with information about how the calling object, parameters, and return value are related after the method call. These relationships can be thought of as a side-effect of the method. The attribute `[Add("Child", item, ctrl)]` creates a "Child" relationship between `item` and `ctrl`, while `[Remove("Child", item, ctrl)]` removes this relationship. Object parameters can be wild-carded, so `[Remove("Child", _, ctrl)]` removes all the "Child" relationships between `ctrl` and any other object. Relationships may refer to the parameters, primitive values, the receiver object, and the return value of a method.[2] These relationships are user-defined; the relationship "Child" has no meaning other than what the framework developer intends for it to mean. Program 7 shows some sample relationships on the `ListControl` API.

Program 7 Partial API of the ListControl

```
public class ListControl {
   [Add("Items", ret, this)]
   ListItemCollection GetItems();

   [Add("Child", ret, this)]
   [Add("Selected", ret, true)]
   ListItem GetSelectedItem();
}

public class ListItem {
   [Add("Selected", this, true)]
   boolean IsSelected();

   [Remove("Selected", this, _)]
   [Add("Selected", this, select)]
   void SetSelected(boolean select);
}
```

We propose to track relationships through the plugin code using a dataflow analysis. After calling a method, we acquire or kill a set of relationships, according to the relationships defined on that method. As an example, we have tracked the relationships (with predicates in curly braces) on the correct code from the DropDownList Selection example in Program 8. Notice that even the absence of a relationship is tracked with '!'; it may be important to know that two objects are not associated with a relationship. Relationship predicates can also be in a third state, "unknown". This is the default state for relationship predicates when we start a method, and it is also used after a branch.

By tracking the relationships through the code, we gain additional knowledge about the framework on every line of

[2] C# attributes and Java annotations do not fully support our notation currently, but we can use longer notation to capture the same information.

Program 8 Tracking relationships on the correct example

```
DropDownList list;

private void Page_Load(object sender, EventArgs e) {
   string searchVal = ...
   ListItemCollection items;
   ListItem newSel, oldSel;

   {}
   oldSel = list.GetSelectedItem();
   {Child(oldSel, list), Selected(oldSel, true)}
   oldSel.SetSelected(false);
   {Child(oldSel, list), !Selected(oldSel, true), Selected(oldSel, false)}

   items = list.GetItems();
   {Child(oldSel, list), !Selected(oldSel, true), Selected(oldSel, false), Items(items, list)}
   newSel = items.FindByValue(searchVal);
   {..., Item(newSel, items), Child(newSel, list)}
   newSel.SetSelected(true);
   {..., Child(newSel, list), Selected(newSel, true)}
}
```

code. Each line now has a context for the current state of the framework.

3.2 Constraints

Once we can track relationships, we can describe the framework constraints and use the relationship predicates to describe semantic aspects of the constraint. We will define a constraint as a path through the plugin code in which some operations are allowed or disallowed. A path can start and end on any expression or language construct. As we will see, the constraint for the DropDownList Selection defines a path that exists between two method calls. On the other hand, the Login Status defines a constraint where the path exists within an if statement. We call these places where constraint paths can start and end *program points*.

As we noted earlier, constraints also have semantic parts. We must be able to declare not just a syntactic expression where the path starts, but also a context that the framework is in at that expression. Therefore, a program point must depend on set of relationships which we must have for the program point to apply.

The constraint specification for the DropDownList Selection constraint appears in Program 9. We will use this example to describe the parts of a constraint specification

- *declared objects* These objects are used in the program points and relationships. Each declared object is bound to a runtime object. A constraint instance is unique if it is bound to a unique set of objects. In our example, we only declare three objects: the `DropDownList`, the `ListItem` we are deselecting, and the `ListItem` we are selecting. The plugin code might reference other objects, such as the `ListItemCollection` or other `ListItem`s, but they are not relevant here.
- `start` A list of program points and relationships that start the path. The program points are syntactic expressions while the relationships provide the semantic context that is required. Both parts are required to trigger a new constraint instance. In this example, the constraint starts when we deselect a `ListItem` and that `ListItem` is a child of a `DropDownList` and it is currently selected. The Selection example only declares one program point for `start`, but we can list several program points and relationship sets.
- `end` A list of program points and relationships that end the path. Like `start`, `end` requires relationships; the program point by itself does not end the path. The Selection constraint will only end if we call `ListItem.SetSelected(false)` on a `ListItem` which is in a Child relationship with the `DropDownList`. This enforces that both `ListItem`s are children of the same `DropDownList`.
- `enable` A list of program points and relationships that are allowed to occur along the path. If the plugin has these relationships and attempts to use the plugin point without a constraint instance, the analysis will trigger an error. In the Selected constraint, we only allow `ListItem.SetSelected()` to be called with a value of `true` while we are on the constraint path. If this is called off the constraint path, we expect an error.
- `forbid` A list of program points and relationships that are not allowed to occur along the path. These program points are not allowed if there is a matching constraint instance; they may only be used off of the path. The Selected constraint forbids the ability to end the method while a constraint path is still active. By forbiding the end of the method, a developer cannot only deselect and leave the `DropDownList` without any item selected.
- `scoped` A subset of the declared objects that are temporary inside this path. They are not valid off the path

Program 9 The Selected constraint specification

```
constraint Selected {
   DropDownList ctrl
   ListItem oldSel
   ListItem newSel

   start:  oldSel.SetSelected(false) with {Child(oldSel, ctrl), Selected(oldSel, true)}
   end:  newSel.SetSelected(true) with {Child(newSel, ctrl)}
   enable: newSel.SetSelected(true) with {Child(newSel, ctrl)}
   forbid:  eom
   scoped: -
}
```

and cannot be stored for later use. This feature is not used in the Selection constraint, but it is used in the Login example.

We will not create the constraint for our other example here, but the path would travel through the if block. Within this block, the constraint will allow the call to `Control.FindControl("LocationList")`. Additionally, the objects within the `LoggedInTemplate` would only be accessible within that block of code; they cannot be stored for later use.

3.3 Proposed Analysis

We propose a static analysis to check for mismatches between plugin code and the framework specifications. The proposed analysis is a simple flow analysis which depends on several other analyses, including an alias analysis and a constant analysis. We have chosen to make the constraint analysis unsound, though we strive for as much soundness as possible if there are only minor tradeoffs for annotation cost and false positives. We should be able to use any alias or constant analysis, though we prefer analyses which have tradeoffs that are consistent with our own.

In Section 3.1, we described how relationships track through the code. We build upon that concept by treating constraints as predicates which flow through the code. Like the relationship predicates, constraint predicates can be in three states: true, false, or unknown. A constraint predicate is true on paths between the declared start and end points of the constraint.

Constraint predicates are parameterized by the variables that are used in the `start` program point and relationships. The Selected constraint defined in Section 3.2 uses a single `Listitem` and single `DropDownList` in the `start` program point and relationships, so the constraint predicate appears as SELECTED(`oldSel, ctrl`).[3] While the constraint specification declares a second `ListItem`, it is not used until later and will not be part of the constraint predicate.

The analysis works by assuming the presence of two logical formulas for each expression signature.[4] The first logical formula is a validation check; the validation check is evaluated for truth to determine whether an expression is valid in the current context. The second formula is an analysis transfer function. It can change the state of the relationship and constraint predicates if certain conditions hold.

These two logical formulae are generated from the constraint and relationship specifications declared by the framework developer. Each piece of a constraint specification is translated into part of a logical formula. The `enable`, `disable`, and `scoped` parts of the specification will generate the validation check, while the `start` and `end` parts will generate the transfer function.

We will examine the validation check first. In the Selected constraint, the enabled specification states:

```
enable:  newSel.SetSelected(true) with
{Child(newSel, ctrl)}
```

This will map `ListItem.SetSelected(boolean)` to the logical formula below.

$$\begin{aligned}&\lambda\, \mathit{newSel} : \mathit{ListItem}.\ \lambda\, b : \mathit{boolean}.\\ &\quad \forall\, \mathit{ctrl} : \mathit{DropDownList}.\ \exists\, \mathit{oldSel} : \mathit{ListItem}.\\ &\qquad b = \mathit{true}\ \wedge\ \mathit{Child}(\mathit{newSel}, \mathit{ctrl}) \implies\\ &\qquad \textsc{Selected}(\mathit{oldSel}, \mathit{ctrl})\end{aligned}$$

This logical formula states that, given a `ListItem` and a `boolean`, if `b` is true and `newSel` has a Child relationship with a `DropDownList`, then we must be currently in a Selected constraint. This constraint must be bound to our `DropDownList` and a `ListItem`.

The logical formula for the `forbid` specification is much shorter, as there are no relationship requirements and the expression takes no parameters. It must simply check that there are no open constraint paths by the end of the method. Therefore, the `forbid` specification maps the validation check for the end of method to:

$\forall\, \mathit{ctrl} : \mathit{DropDownList}.\ \forall\, \mathit{oldSel} : \mathit{ListItem}.$

[3] We will use small caps to represent constraint predicates and regular capitalization for relationship predicates.

[4] By expression signature, we mean a kind of operation with a unique typing signature. That is, a method call to `Listitem.SetSelected(boolean)` is distinct from a method call to `DropDownList.GetItems()`. Likewise, an assignment of the form `ListItem = ListItem` is different from an assignment of `DropDownLists`.

$\neg$SELECTED$(oldSel, ctrl)$

We will now examine the specification parts that contribute to the transfer functions. The `start` piece of the Selected constraint will cause a constraint predicate to change state to true. The start specification

```
start:  oldSel.SetSelected(false) with
{Child(oldSel, ctrl), Selected(oldSel, true)}
```

will map `ListItem.SetSelected(boolean)` to a logical formula that uses a new operator, `generates`. This operator will evaluate the left hand side first, and it will only generate the predicates on the right if the left side is true.[5] The formula below will bind a constraint predicate to the objects `oldSel` and `ctrl` and will set this predicate to be true, but only if we have the correct relationship predicates.

$$\begin{array}{l} \lambda\ oldSel : ListItem.\ \lambda\ b : boolean. \\ \quad \forall\ ctrl : DropDownList. \\ \quad\quad b = false \wedge Child(oldSel, ctrl) \wedge Selected(oldSel, true) \\ \quad\quad \texttt{generates}\ \textsc{Selected}(oldSel, ctrl) \end{array}$$

The `end` specification will set a SELECTED predicate to false if it was already true and we have the right relationships. The translation of the `end` specification will map `ListItem.SetSelected(boolean)` to the transfer function below.

$$\begin{array}{l} \lambda\ item : ListItem.\ \lambda\ b : boolean. \\ \quad \forall\ ctrl : DropDownList.\ \forall\ oldSel : ListItem. \\ \quad\quad b = true \wedge Child(item, ctrl) \wedge \textsc{Selected}(oldSel, ctrl) \\ \quad\quad \texttt{generates}\ \neg\textsc{Selected}(oldSel, ctrl) \end{array}$$

Of course, now we have multiple logical formulae for the operator `ListItem.SetSelected(boolean)`. We combine these formulae together so that the transfer function for `Listitem.SetSelected()` is:

$$\begin{array}{l} \lambda\ item : ListItem.\ \lambda\ b : boolean. \\ \quad (\forall\ ctrl : DropDownList. \\ \quad\quad b = false \wedge Child(item, ctrl) \wedge Selected(item, true) \\ \quad\quad \texttt{generates}\ \textsc{Selected}(item, ctrl)) \\ \quad \wedge \\ \quad (\forall\ ctrl : DropDownList. \forall\ oldSel : ListItem. \\ \quad\quad b = true \wedge Child(item, ctrl) \wedge \textsc{Selected}(oldSel, ctrl) \\ \quad\quad \texttt{generates}\ \neg\textsc{Selected}(oldSel, ctrl)) \end{array}$$

We use a similar methodology to combine formula from different constraint specifications. Our rules for translating constraint specifications into logical formula guarantee that the constraints do not interfere.

We also translate the relationship specifications into logical formulae, though they are much simpler to translate. The specification

```
[Add("Child", ret, this)]
[Add("Selected", ret, true)]
ListItem GetSelectedItem();
```

will map the operation `DropDownList.GetSelectedItem()` to the transfer function

$$\begin{array}{l} \lambda\ ret : ListItem.\ \lambda\ this : DropDownList. \\ \quad \texttt{true generates}\ Child(ret, this) \wedge Selected(ret, true) \end{array}$$

These are combined into the same transfer function generated from the constraint specifications. The only difference is that the relationship predicates are allowed to interfere, as changing a relationship specification is expected to change the analysis results for many constraints.

By translating constraint specifications into logical formulae, we have produced validation checks and transfer functions for a simple flow analysis. The analysis must depend on a constant analysis to determine the truth of expressions such as `b == true`. It must also depend upon an alias analysis so that it can track objects as labels rather than variable names. Labels allow the analysis to keep track of relationship predicates after the variables go out of scope, and they allow the analysis to track objects which are only indirectly referenced, such as elements within arrays.

Program 10 shows how the DropDownList Selection constraint properly allows the correct plugin code. At each expression, we first use the validation check to make sure that the expression is allowed in the current context. Once we know the expression is allowed, we use the transfer function to propagate any changes produced from the expression. Upon finding the call to `oldSel.SetSelected(false)`, we will run the transfer function for `ListItem.SetSelected(boolean)` and the constraint predicate SELECTED`(oldSel, ctrl)` will be set to true. Later, the analysis will find the call to `newSel.SetSelected(true)`. The validation check for this expression will pass since we meet the relationship requirements and we currently are on the constraint path. The transfer function for this expression will also end the constraint, thus allowing us to end the method properly.

When we revisit our incorrect plugin code, the analysis will find the defect and produce a local error. Program 11 is our incorrect code that shows the results of the relationship analysis. As in the previous example, the constraint analysis will check whether `newSel.SetSelected(true)` is accessible. However, the logical formula generated by `enable` will fail, so the analysis will produce an error on the last line.

Program 12 showcases how constraints can create instances to separately check many objects. In this example, we have the same problem as the previous example, except now we have two `DropDownLists`. We deselect correctly from `listA`, but instead of then selecting a `listA`, we select a `listB` item. Had we specified constraints in a purely syntactic manner, the analysis would not have caught this error. However, the analysis recognizes that these are different objects. When

[5]The right hand side of this operator is restricted to only conjuncted and negated predicates. Disjunction and implication are not allowed on the right hand side of a `generates` operator.

Program 10 The Selected scope on correct code

```
DropDownList list;
private void Page_Load(object sender, EventArgs e) {
   string searchVal = ...
   ListItemCollection items;
   ListItem newSel, oldSel;

   oldSel = list.GetSelectedItem();
   {Child(oldSel, list), Selected(oldSel, true)}
   oldSel.SetSelected(false);
//selected constraint instance begins
   {Child(oldSel, list), !Selected(oldSel, true), Selected(oldSel, false), SELECTED(oldSel, list)}

   items = list.GetItems();
   {Child(oldSel, list), ..., SELECTED(oldSel, list), Items(items, list)}
   newSel = items.FindByValue(searchVal);
   {Child(oldSel, list), ..., SELECTED(oldSel, list), ..., Item(newSel, items), Child(newSel, list)}
//calling this operation with the current relationships is allowed by the selected constraint
   newSel.SetSelected(true);
//selected constraint ends
   {Child(oldSel, list), ..., !SELECTED(oldSel, list), Child(newSel, list), Selected(newSel, true)}
//the end-of-method can be used because we exited the selected scope
}
```

Program 11 Detecting a selection error

```
DropDownList list;
private void Page_Load(object sender, EventArgs e) {
   string searchVal = ...
   ListItemCollection items;
   ListItem newSel, oldSel;

   items = list.GetItems();
   {Items(items, list)}
   newSel = items.FindByValue(searchVal);
   {Items(items, list), Item(newSel, items), Child(newSel, list)}
//ERROR
   newSel.SetSelected(true);
}
```

Program 12 Detecting an error with multiple DropDownLists

```
DropDownList listA, listB;
private void Page_Load(object sender, EventArgs e) {
   string searchVal = ...
   ListItemCollection items;
   ListItem newSel, oldSel;

   oldSel = listA.GetSelectedItem();
   {Child(oldSel, listA), Selected(oldSel, true)}
   oldSel.SetSelected(false);
//selected constraint instance begins with listA and oldSel bound
   {Child(oldSel, listA), !Selected(oldSel, true), Selected(oldSel, false), SELECTED(oldSel, listA)}

   items = listB.GetItems();
   {Child(oldSel, listA), ..., SELECTED(oldSel, listA), Items(items, listB)}
   newSel = items.FindByValue(searchVal);
   {Child(oldSel, listA), ..., SELECTED(oldSel, listA), ..., Item(newSel, items), Child(newSel, listB)}
//ERROR
    newSel.SetSelected(true);
}
```

```
Suggestion: Insert before newSel.SetSelected(true)
   ListItem listItem1 = listB.GetSelectedItem();
   listItem1.SetSelected(false);
```

Figure 3: Suggestion based upon the current context

the constraint region starts at `oldSel.SetSelected(false)`, it will bind `listA` to the constraint variable `list`. Later on, the analysis checks the line `newSel.SetSelected(true)`, but it also fails on the logical formula generated by `enable`. Since we do not have a constraint predicate SELECTED`(oldSel, listB)`, the analysis generates an error.

It is possible for the analysis to produce false positives. Our preliminary investigations show that the false positives typically come from code which is extremely difficult to read; refactoring the code into a cleaner state appears to remove the false positives. We will be investigating the extent of false positives in real codebases in later work, along with how to minimize them further.

4. FUTURE WORK

The work on formalizing the analysis is ongoing, though we described the preliminary direction in Section 3.3. Once the formalization is complete, we will implement the analysis and use it to check several more examples from frameworks such as ASP.NET, EJB, and Eclipse.

Currently, the specifications for constraints are not as concise as we would like, and we are exploring ways to condense the constraint abstraction. We are also investigating alternate abstractions, such as pre-/post- conditions and multi-object typestates.

If possible, we would also like the analysis to suggest steps to fix the plugin. For example, if an operation cannot be called because it is controlled by a constraint, then the analysis will suggest operations to trigger the constraint. The suggestion would be based upon the plugin context right before the error and the specification of the constraint that caused the error. A suggestion for Program 11 might read as shown in Figure 3. Notice that the suggestion refers to `listB`, which it retrieved from the relationship context at the error site.

We will also explore how this abstraction enhances existing abstractions for library constraints. Libraries and frameworks are very similar, and many industrial codebases, such as Swing and ASP.NET, are actually both. The difference is merely in how the developer uses the codebase, and developers must frequently switch back and forth. The ability to describe both framework and library constraints, in one system, would be extremely beneficial. In future work, we will look at how to integrate this specification system with specifications meant primarly for libraries. Typestates[4] and tracematches[5] are both likely candidates for future interaction with library constraints.

5. RELATED WORK

Some of the original work in frameworks [6] discussed using design patterns as a way of describing frameworks. Later research has looked at formalizing design patterns and extracting design patterns from code[7, 8, 9]. Patterns alone cannot completely specify a framework. While they provide information about high-level interaction mechanisms, they do not describe the temporal framework constraints shown in our examples.

SCL [10, 11] allows framework developers to create a specification for the structural constraints for using the framework. The specifications we propose focus on semantic constraints rather than structural constraints. Some of the key ideas from SCL could be used to drive the more structural focused parts of the specifications.

Object typestates [4, 12] provide a mechanism for specifying a protocol between a library and a client. The client sees the library to be in a particular state, and calling methods on the library transitions it into a new state. This general concept can also be applied to frameworks and plugins. However, due to inversion of control, the protocol is now on the plugin; in a framework setting, we call this a *lifecycle*. If we continue to use typestates to represent lifecycles, then the plugin methods are the state transitions. This is not how a plugin developer thinks of the code; we would prefer to think of the framework as transitioning the state and the plugin doing specialized code within the current framework state. Additionally, framework states involve multiple interacting objects; this is awkward to model with typestates. While the proposed work may be inherently different from a typestate-based protocol, it might be possible to reuse some of the underlying theory.

Some typestate work has explored inter-object typestate. This work still considers each object to have an individual typestate, though it can be affected by other objects [13] or manipulated through participation in data structures [14]. The proposed specifications differ in that they view multiple heterogeneous objects as having a shared state.

Scoped Methods [15] are another mechanism for enforcing protocol. They create a specialized language construct that requires a protocol to be followed within it. Like SCL, this is structural and does not take context into account.

Tracematches have also been used to enforce protocol, and they take semantic knowledge into account. Tracematches provide a user-friendly way to specify a temporal sequence of events that involve multiple objects. They have been used to change the program execution with a dynamic analysis [5], and they have been used to check protocols with a static analysis [16]. Tracematches also allow global checking when paired with a dynamic analysis. Unlike the proposed work, tracematches have no restrictions about interactions between tracematches. Additionally, a tracematch does not separate the general knowledge about the framework from the constraint itself. In cases where multiple execution traces lead to the same constraint, a tracematch would have to specify each possibility. Since the proposed solution depends on relationships, the constraint only needs to specify which relationships it requires for each program point. This separation of the constraints from the relationship definitions allows the specifications to be more flexible to future changes.

Like the proposed framework language, Contracts [17] also view the relationships between objects as a key factor in specifying systems. A contract also declares the objects involved in the contract, an invariant, and a lifetime where the invariant is guaranteed to hold. Contracts allow all the power of first-order predicate logic and can express very complex invariants. Contracts differ from the proposed specifications because they do not make the tie directly back to the plugin code and have a higher complexity for the writer of the contract.

Other research projects [18, 19, 20, 21] help plugin developers by finding or encoding known good patterns for using frameworks. The proposed work differs significantly in that it does not suggest a way to complete the task, but it finds defects once a task has been started. We see the two bodies of research as complimentary.

This work also has some overlap with formal methods, particularly in describing the relationships and invariants of code [22, 23]. These formal methods verify that the specified code is correct with respect to the specification. Instead, we are checking the unspecified plugin code against the framework's specification. Other formal methods [24, 25] focus on a detailed description of the entire system. These systems also allow developers to model the invariants between objects, and some of the notation could be reused for the translated logical formulae. However, the checkers for these systems are meant to stand on their own, without any ties to executable code. Additionally, the checkers work at a more global level and expect to verify global properties and invariants of the system. The proposed analysis intends to check framework constraints in a local manner, and it plans in the ability for constraints to hold true only for a period of time.

The analysis itself is similar to shape analysis, with the closest working being TVLA [26]. This work allows custom shape analyses to create new predicates between objects. The custom analysis writer must provide logic that states whether the predicate is true or false for each type of expression; this logic is similar to a transfer function. It is possible to translate the constraint specifications into logic that can be used by TVLA. However, TVLA needs to reevaluate every predicate at every expression; this makes it difficult to automatically generate a logic which holds true even when unchanged by the expression. The proposed analysis only changes specific predicates which are mentioned in the constraint specification, so it is easier to automatically generate a logical formula from it.

6. CONCLUSION

Frameworks place constraints on plugins that are relative to a semantic context of the plugin code. These constraints can be dependent upon many interacting objects, and they can affect the operations and objects which the plugin code can access.

We have proposed a lightweight and modular way to specify framework constraints and check plugins for broken framework constraints. The plugin developers will not have to do anything other than run a tool, and the proposed specifications are written entirely by framework developers. The framework developer uses relationships to define associations between framework objects at runtime. An analysis uses these relationships to keep track of the semantic knowledge a plugin has based upon its previous framework interactions. This semantic knowledge is also used by the framework developer to add semantic aspects to the framework constraints. A framework developer can specify the constraints as valid paths through plugin code, and she can specify operations and objects which are allowed or disallowed on the path. The constraints are specified and checked separately from each other, so that adding or removing constraints will not affect how the other constraints are enforced. In future work, we plan to implement the analysis which reads these specifications to discover defects in plugins, and we will apply the specifications and analysis to several industry frameworks.

7. ACKNOWLEDGMENTS

This work was supported in part by NSF grant CCF-0546550, DARPA contract HR00110710019, the Department of Defense, and the Software Industry Center at CMU and its sponsors, especially the Alfred P. Sloan Foundation.

8. REFERENCES

[1] Johnson, R.E.: Frameworks = (components + patterns). Commun. ACM **40**(10) (1997)

[2] (none): (The ASP.NET forums) http://forums.asp.net.

[3] (none): Binding to a DropDownList membership roles (2006) http://forums.asp.net/thread/1415249.aspx.

[4] DeLine, R., Fahndrich, M.: Typestates for objects. In: Proceedings of the European Conference on Object Oriented Programming. (2004)

[5] Walker, R.J., Viggers, K.: Implementing protocols via declarative event patterns. In: Proceedings of the 12th International symposium on Foundations of Software Engineering, New York, NY, USA, ACM Press (2004) 159–169

[6] Johnson, R.E.: Documenting frameworks using patterns. In: conference proceedings on Object-oriented programming systems, languages, and applications. (1992)

[7] G. Florijn, M. Meijers, P.v.W.: Tool support for object-oriented patterns. In: Proceedings of the European Conference on Object Oriented Programmming. (1997)

[8] D. Heuzeroth, S. Mandel, W.L.: Generating design pattern detectors from pattern specifications. In: 18th IEEE International Conference on Automated Software Engineering. (2003)

[9] Soundarajan, N., Hallstrom, J.O.: Responsibilities and rewards: Specifying design patterns. In: Proceedings of the 26th International Conference on Software Engineering. (2004)

[10] Hou, D., Hoover, H.J.: Towards specifying constraints for object-oriented frameworks. In: Proceedings of the 2001 conference of the Centre for Advanced Studies on Collaborative research. (2001)

[11] Hou, D., Hoover, H.J.: Using SCL to specify and check design intent in source code. IEEE Trans. Softw. Eng. **32**(6) (2006)

[12] Bierhoff, K., Aldrich, J.: Modular typestate checking of aliased objects. In: To appear at OOPSLA '07, Montreal, Canada (2007)
[13] Nanda, M.G., Grothoff, C., Chandra, S.: Deriving object typestates in the presence of inter-object references. In: OOPSLA. (2005)
[14] Lam, P., Kuncak, V., Rinard, M.: Generalized typestate checking for data structure consistency. In: Verification, Model Checking, and Abstract Interpretation. (2005)
[15] Tan, G., Ou, X., Walker, D.: Enforcing resource usage protocols via scoped methods (2003) Appeared in the 10th International Workshops on Foundations of Object-Oriented Languages.
[16] Martin, M., Livshits, B., Lam, M.S.: Finding application errors and security flaws using PQL: a program query language. In: Proceedings of the 20th Conference on Object oriented Programming, Systems, Languages, and Applications. (2005)
[17] Helm, R., Holland, I.M., Gangopadhyay, D.: Contracts: specifying behavioral compositions in object-oriented systems. In: Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications. (1990)
[18] Froehlich, G., Hoover, H.J., Liu, L., Sorenson, P.: Hooking into object-oriented application frameworks. In: Proceedings of the 19th International Conference on Software engineering. (1997)
[19] Riehle, D.: Framework Design: A Role Modeling Approach. PhD thesis, Zurich (2000)
[20] Mandelin, D., Xu, L., Bod, R., Kimelman, D.: Jungloid mining: helping to navigate the API jungle. In: Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation. (2005)
[21] Fairbanks, G., Garlan, D., Scherlis, W.: Design fragments make using frameworks easier. In: Proceedings of the 21st Conference on Object-oriented programming systems, languages, and applications, New York, NY, USA, ACM Press (2006) 762–763
[22] Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: Proceedings of the Conference on Programming Language Design and Implementation. (2002)
[23] Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for Java. SIGSOFT Softw. Eng. Notes **31**(3) (2006)
[24] Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. **11**(2) (2002) 256–290
[25] Spivey, J.: The Z Notation: A Reference Manual. Prentice Hall (1992)
[26] Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. **24**(3) (2002) 217–298