# Checking Framework Interactions with Relationships

Ciera Jaspan and Jonathan Aldrich

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

**Abstract.** Software frameworks impose constraints on how plugins may interact with them. Many of these constraints involve multiple objects, are temporal, and depend on runtime values. Additionally, they are difficult to specify because they are often extrinsic and may break behavioral subtyping. This work introduces *relationships* as an abstraction for specifying framework constraints in FUSION (Framework Usage SpecificatIONs), and it presents a formal description and implementation of a static analysis to find constraint violations in plugin code. We define three variants of this analysis: one is sound, one is complete, and a pragmatic variant that balances these tradeoffs. We prove soundness and completeness for the appropriate variants, and we show that the pragmatic variant can effectively check constraints from real-world programs.

## 1 Introduction

Object-oriented frameworks have brought many benefits to software development, including reusable codebases, extensible systems, and encapsulation of quality attributes. However, frameworks are used at a high cost; they are complex and difficult to learn [1]. This is partially due to the complexity of the semantic constraints they place on the *plugins* that utilize them.

As an example, consider a constraint in the ASP.NET web application framework. The ASP.NET framework allows developers to create web pages with user interface controls on them. These controls can be manipulated programatically through callbacks provided by the framework. A developer can write code that responds to control events, adds and removes controls, and changes the state of controls.

One task that a developer might want to perform is to programmatically change the selection of a drop down list. The ASP.NET framework provides the relevant pieces, as shown in Fig. 1[1]. Notice that if the developer wants to change the selection of a `DropDownList` (or any other derived `ListControl`), she has to access the individual `ListItems` through the `ListItemCollection` and change the selection using `setSelected`. Based on this information, she might naïvely change the selection as shown in List. 1. Her expectation is that the framework will see that she has selected a new item and will change the selection accordingly.

When the developer runs this code, she will get the error shown in Fig. 2. The error message clearly describes the problem; a `DropDownList` had more than one item

[1] As the implementation of FUSION runs on Java, we translated the examples to Java syntax.

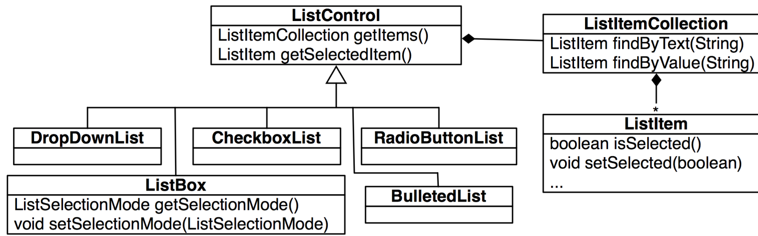



**Fig. 1:** ASP.NET ListControl Class Diagram

**List. 1:** Incorrect selection for a DropDownList

```
1  DropDownList list;
2
3  private void Page_Load(object sender, EventArgs e) {
4     ListItem newSel;
5     newSel = list.getItems().findByValue("foo");
6     newSel.setSelected(true);
7  }
```

selected. This error is due to the fact that the developer did not de-select the previously selected item, and, by design, the framework does not do this automatically. While an experienced developer will realize that this was the problem, an inexperienced developer might be confused because she did not select multiple items.

The stack trace in Fig. 2 is even more interesting because it does not point to the code where the developer made the selection. In fact, the entire stack trace is from framework code; there is no plugin code referenced at all! At runtime, the framework called the plugin developer's code in List. 1, this code ran and returned to the framework, and then the framework discovered the error. To make matters worse, the program control could go back and forth several times before finally reaching the check that triggered the exception. Since the developer doesn't know exactly where the problem occurred, or even what object it occurred on, she must search her code by hand to find the erroneous selection.

The correct code for this task is in List. 2. In this code snippet, the developer deselects the currently selected item before selecting a new item.

*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
```

**Fig. 2:** Error with partial stack trace from ASP.NET

**List. 2:** Correctly selecting an item using the ASP.NET API

```
1  DropDownList list;
2
3  private void Page_Load(object sender, EventArgs e) {
4    ListItem newSel, oldSel;
5    oldSel = list.getSelectedItem();
6    oldSel.setSelected(false);
7    newSel = list.getItems().findByValue("foo");
8    newSel.setSelected(true);
9  }
```

**List. 3:** Selecting on the wrong DropDownList

```
1  DropDownList listA, listB;
2
3  private void Page_Load(object sender, EventArgs e) {
4    ListItem newSel, oldSel;
5    oldSel = listA.getSelectedItem();
6    oldSel.setSelected(false);
7    newSel = listB.getItems().findByValue("foo");
8    newSel.setSelected(true);
9  }
```

This example, and many others we have found on the ASP.NET developer forum, shows three interesting properties of framework constraints.

**Framework constraints involve multiple classes and objects.** List. 2 requires four objects to make the proper selection. The framework code that the plugin used was located in four classes.

**Framework constraints are often extrinsic.** While the `DropDownList` was the class that checked the constraint (as seen by the stack trace), the constraint itself was on the methods of `ListItem`. However, the `ListItem` class is not aware of the `DropDown-List` class or even that it is within a `ListControl` at all, and therefore it should not be responsible for enforcing the constraint. Compare the extrinsic nature of these constraints to the intrinsic nature of a class invariant. In addition to being difficult to check, it is more difficult to document an extrinsic constraint as it is unclear where the documentation should go so that the plugin developer will naturally discover it.

**Framework constraints have semantic properties.** Framework constraints are not only about structural concerns such as method naming conventions or types; the developer must also be aware of semantic properties of the constraint. There are at least three semantic properties shown by the DropDownList example. First, the plugin developer had to know which objects she was using to avoid the problem in List. 3. In this example, the developer called the correct operations, but on the wrong objects. She also had to notice which primitive values (such as `true` or `false`) she used in the calls to change the selection. Finally, she had to be aware of the ordering of the operations. In List. 2, had she swapped lines 5 and 6 with lines 7 and 8, she would have caused unexpected runtime behavior where the selection change does not occur. This behavior

occurs because `getSelectedItem` returns the first selected `ListItem` that it finds in the `DropDownList`, and that may be the newly selected item rather than the old item.

In previous work [2], we proposed a preliminary specification approach and sketched a hypothetical analysis to discover semantic mismatches, such as the ones described above, between the plugin code and the declared constraints of the framework. The previous work primarily discussed the requirements for such a system and explored a prototype specification. In this paper, we make three contributions:

1. We show that the concept of framework developer-defined relations across objects captures an underlying programming model used to interact with frameworks. We use these relations to specify framework constraints FUSION (Framework Usage SpecificatIONs). (Sect. 2)
2. We propose (Sect. 3) and formally define (Sect. 4) a static analysis that detects violations of constraints in plugins. We define three variants of the FUSION analysis: a sound variant, a complete variant, and a third variant that is neither sound nor complete. We prove soundness and completeness for the appropriate variants, and we argue that the pragmatic variant is better for practical use. There are only minor differences between the variants, so it is simple to switch between them.
3. We implemented the pragmatic variant of the FUSION analysis and ran it on code based on examples from framework help forums. As the FUISION does not require the entire framework to be specified, framework developers will be able to add specifications as they answer questions on these forums. We show that FUSION captures the properties described and that the pragmatic variant can handle real-world code with relatively few false positives and false negatives. (Sect. 5)

## 2 Developer-defined Relations over Objects

When a plugin developer programs to a framework, the primary task is not about creating new objects or data. In many cases, programming in this environment is about *manipulating the abstract associations between existing objects*. Every time the plugin receives a callback from the framework, it is implicitly notified of the current associations between objects. As the plugin calls framework methods, the framework changes these associations, and the plugin learns more about how the objects relate. Each method call, field access, or conditional test gives the plugin more information. For example, in List. 2, when the plugin made the call to `ListItemCollection.findItemByValue`, it learned about the association between the returned `ListItem` and the `DropDownList`. These may be direct associations within code, or they may represent an abstract association with no references in memory. Even when the plugin needs to create a new object, it is frequently done by calling abstract factory methods that set up the object and its relationships with other objects. Many frameworks, including ASP.NET, also use dependency injection, a mechanism in which the framework populates the fields of the plugin based on an external configuration file [3]. When using dependency injection, the plugin simply receives and manipulates pre-configured objects. In the `DropDownList` example, all the objects are provided by the framework through dependency injection, and the plugin simply changes their relationships with each other.

Since the primary mechanism of interaction is based on manipulating relationships

**List. 4:** The Child relation. Every relation must define `params`, `effect`, and `test`

```
@Relation({ListItem.class, ListControl.class})
public @interface Child {
    String[] params();
    Effect effect();
    String test() default "";
}
```

between objects, we will model it formally using a mathematical relation. A *relation* is a named set of tuples on several types $\tau$.[2] A *relationship* is a single tuple in a relation, represented as

$$name(\ell_1, \ldots, \ell_n)$$

where each $\ell$ is a static representation of a runtime object with the type defined by the relation.

In this section, we introduce FUSION and three specification constructs based on relationships. The first construct in FUSION, *relationship effects*, specify how framework operations change associations between objects. The second construct, *constraints*, uses relationships to specify extrinsic and semantic constraints across multiple objects. Finally, *relation inference rules* specify how relationships can be inferred based on the current state of other relationships, regardless of what operations are used.

### 2.1 Relationship Effects

*Relationship effects* specify changes to the relations that occur after calling a framework method. The framework developer annotates the framework methods with information about how the calling object, parameters, and return value are related (or not related) after a call to the method. These annotations describe additions and removals of relationships from a relation. For example, the annotation @Item({item, list}, ADD) creates an Item relationship between `item` and `list`, while @Item({item, list}, REMOVE) removes this relationship[3]. When a relationship is removed or added, we are simply marking whether or not its existence in the relation is known. Thereby, if a relationship is "removed", but there was no prior knowledge of whether it existed, it is marked as definitely not in the relation.

Relationship effects may refer to the parameters, the receiver object, and the return value of a method. They may also refer to primitive values. Additionally, parameters can be wild-carded, so @Item({*, list}, REMOVE) removes *all* the Item relationships between `list` and any other object.

In addition to the ADD and REMOVE effects, a TEST effect uses a parameter to determine whether to add or remove a relationship. For example, we might annotate the

[2] The relations shown in this paper are only unary and binary, but n-ary relations are supported.

[3] We are presenting a simplified version of the syntax for readability purposes. The correct Java syntax for the add annotation appears as @Item(params={"item", "list"}, effect=ADD). This is the syntax used in the implementation.

**List. 5:** Partial `ListControl` API with relationship effect annotations

```
1  public class ListControl {
2      @List({result, target}, ADD)
3      public ListItemCollection getItems();
4  
5      //After this call we know two pieces of information. The returned item is selected and it is a child of this
6      @Child({result, target}, ADD)
7      @Selected({result}, ADD)
8      public ListItem getSelectedItem();
9  }
10 public class ListItem {
11     //If the return is true, then we know we have a selected item. If it is false, we know it was not selected.
12     @Selected({target}, TEST, return)
13     public boolean isSelected();
14 
15     @Selected({target}, TEST, select)
16     public void setSelected(boolean select);
17 
18     @Text({result, target}, ADD)
19     public String getText();
20 
21     //When we call setText, remove any previous Text relationships, then add one for text
22     @Text({*, target}, REMOVE)
23     @Text({text, target}, ADD)
24     public void setText(String text);
25 }
26 public class ListItemCollection {
27     @Item({item, target}, REMOVE)
28     public void remove(ListItem item);
29 
30     @Item({item, target}, ADD)
31     public void add(ListItem item);
32 
33     @Item({item, target}, TEST, result)
34     public boolean contains(ListItem item);
35 
36     @Item({result, target}, ADD)
37     @Text({text, result}, ADD)
38     public ListItem findByText(String text);
39 
40     //if we had any items before this, remove them after this call
41     @Item({*, target}, REMOVE)
42     public void clear();
43 }
```

method `List.contains(Object obj)` with @Item({obj, target}, TEST, result) to signify that this relationship is added when the return value is true and removed when the return value `result` is false.

As relations are user-defined, they have no predefined semantics. Any hierarchy or ownership present, such as Child or Item relations, is only inserted by the framework developer. In fact, relationships do not have to reflect *any* reference paths found in the heap, but may exist only as an abstraction to the developer. This allows relations to

**List. 6:** Comments showing how the relationship context changes after each instruction

```
1  DropDownList ddl = ...;
2  ListItemCollection coll;
3  ListItem newSel, oldSel;
4  oldSel = ddl.getSelectedItem();
5     //Child(oldSel, ddl), Selected(oldSel)
6  oldSel.setSelected(false);
7     //Child(oldSel, ddl), !Selected(oldSel)
8  coll = ddl.getItems();
9     //Child(oldSel, ddl), !Selected(oldSel), List(coll, ddl)
10 newSel = coll.findByText("foo");
11    //Child(oldSel, ddl), !Selected(oldSel), List(coll, ddl), Item(newSel, coll), Text("foo", newSel)
```

**List. 7:** DropDownList Selection Constraints and Inferred Relationships

```
1  @Constraint(
2     op=''ListItem.setSelected(boolean select)'',
3     trigger=''select == false and Child(target, ctrl) and ctrl instanceof DropDownList'',
4     requires=''Selected(target)'', effect={''!CorrectlySelected(ctrl)''})
5  @Constraint(
6     op=''ListItem.setSelected(boolean select)'',
7     trigger=''select == true and Child(target, ctrl) and ctrl instanceof DropDownList'',
8     requires=''!CorrectlySelected(ctrl)'', effect={''CorrectlySelected(ctrl)''})
9  @Constraint(
10    op=''end-of-method'',
11    trigger=''ctrl instanceof DropDownList'',
12    requires=''CorrectlySelected(ctrl)'', effect={})
13 @Infer(trigger=''List(list, ctrl) and Item(item, list)'', infer={''Child(item, ctrl)''})
14 public class DropDownList {...}
```

be treated as an abstraction independent from code. This is a common specification paradigm; relations have a similar purpose to model fields in JML specifications [4].

To define a new relation, the framework developer creates an annotation type and uses the meta-annotation @Relation to signify it as a relation over specific types. List. 4 shows a sample definition of the Child relation from the `DropDownList` example.

Once the framework developer defines the desired relations, they can be used as relationship effects, as shown in List. 5. These annotations allow tools to track relationships through the plugin code at compile time. List. 6 shows a snippet from a plugin, along with the current relationships after each instruction. For example, after line 4 in List. 6, we apply the effects declared in List. 5, lines 6-8. Therefore, at line 5, we learn the two new relationships shown. This information, the *relationship context*, provides us with an abstract, semantic context that each instruction resides in. In the next section, we use this context to check the semantic parts of framework constraints.

## 2.2 Constraints

Framework developers can specify *constraints* on framework operations in a propositional logic over relationships. They are written as class-level annotations, but as con-

straints are extrinsic, they can constrain the operations on any other class. As the three examples in List. 7 show, a constraint has four parts:

1. *operation*: This is a signature of an operation to be constrained, such as a method call, constructor call, or even a tag signaling the end of a method. Notice that these constraints may be defined in another class, as in the first constraint in List. 7. This makes constraints more expressible that a class or protocol invariant.
2. *trigger predicate*: This is a logical predicate over relationships. The plugin's relationship context must determine that this predicate holds for this constraint to be triggered. If not, the constraint is ignored. While *operation* provides a syntactic trigger for the constraint, *trigger* provides the semantic trigger. The combination of both a syntactic and semantic trigger allows constraints to be more flexible and expressible than many existing protocol-based solutions.
3. *requires predicate*: This is another logical predicate over relationships. If the constraint is triggered, then this predicate must be true under the current relationship context. If the requires predicate is not true, this is a broken constraint and the analysis should signal an error in the plugin.
4. *effect list*: This is a list of relationship effects. If the constraint is triggered, these effects will be applied in the same way as the relationship effects described earlier. They will be applied regardless of the state of the requires predicate.

In the first example at the top of List. 7, the constraint is checking that at every call to `ListItem.setSelected(boolean)`, if the relationship context shows that the argument is false, the receiver is a Child of a `ListControl`, and if that `ListControl` is a `DropDownList`, then it must also indicate that the `ListItem` is Selected. Additionally, the context will change so that the `DropDownList` is not CorrectlySelected. The second constraint is similar to the first and it enforces proper selection of `ListItem`s in a `DropDownList`. The third constraint ensures that the plugin method does not end in an improper state by utilizing the "end-of-method" instruction to trigger when a plugin callback is about to end.

### 2.3 Inferred relationships

In some cases, the relationships between objects are implicit. Consider the `ListItem-Collection` from the `DropDownList` example. The framework developer would like to state that items in this list are in a Child relation with the `ListControl` parent. However, it does not make sense to annotate the `ListItemCollection` class with this information since `ListItemCollection`s should not know about `ListControl`s.

*Inferred relationships* describe these implicit relationships that can be assumed at any time. In List. 7, line 13 shows an example for inferring a Child relationship based on the relations Item and List. Whenever the relationship context can show that the "trigger" predicate is true, it can infer the relationship effects in the "infer" list. Inferred relationships allow the framework developer to specify relationship effects that would otherwise have to be placed on every location that the predicate is true; this would significantly drive up the cost of adding these specifications.

It is possible to produce inferred relationships that directly conflict with the relationship context. To prevent this, the semantics of inferred relationships is that they are ignored in the case of a conflict, that is, relationships from declared relationship

effects and constraints have a higher precedence. The rationale behind this is that the constraints and relationship effects are explicitly declared, and this should be reflected by the giving them precedence. Additionally, the inferred relationships are only used on an as-needed basis; to generate all possible inferred relations would be expensive for the analysis. An alternative mechanism would be to signal an error, though it is not currently clear whether this will increase the number of false positives.

## 3 The FUSION Analysis

We have designed and implemented a static analysis to track relationships through plugin code and check plugin code against framework constraints. The FUSION analysis is a modular, branch-sensitive, forward dataflow analysis[4]. It is designed to work on a three address code representation of Java-like source. We assume that the analysis runs in a framework that provides all of these features. In this section, we will present the analysis data structures, the intuition behind the three variations of the analysis, and a discussion of their tradeoffs. Sect. 4 defines how the analysis runs on each instruction.

The FUSION analysis is dependent on several other analyses, including a boolean constant propagation analysis and an alias analysis. The FUSION analysis uses the constant propagation analysis for the TEST effect. For this purpose, the relation analysis assumes there is a function $\mathcal{B}$ to which it can pass a variable and learn whether the represented value is true, false, or unknown.

The FUSION analysis can use any alias analysis which implements a simple interface. First, it assumes there is a context $\mathcal{L}$ that given any variable $\mathtt{x}$, provides a finite set $\bar{\ell}$ of abstract locations that the variable might point to. Second, it assumes a context $\Gamma_\ell$ which maps every location $\ell$ to a type $\tau$. The combination of these two contexts, $< \Gamma_\ell, \mathcal{L} >$ is represented as the alias lattice $\mathcal{A}$. This lattice must conservatively abstract the heap, as defined by Def. 1.

**Definition 1 (Abstraction of Alias Lattice).** *Assume that a heap* $\mathsf{h}$ *is defined as a set of source variables* $\mathtt{x}$ *which point to a runtime location* $\ell$ *of type* $\tau$. *Let* $\mathsf{H}$ *be all the possible heaps at a particular program point. An alias lattice* $< \Gamma_\ell, \mathcal{L} >$ *abstracts* $\mathsf{H}$ *at a program counter if and only if*

$$
\begin{array}{l}
\forall\, \mathsf{h} \in \mathsf{H}\,.\, \mathsf{dom}(\mathsf{h}) = \mathsf{dom}(\mathcal{L}) \text{ and} \\
\quad \forall\, (\mathtt{x}_1 \hookrightarrow \ell_1 : \tau_1) \in \mathsf{h}\,.\, \forall\, (\mathtt{x}_2 \hookrightarrow \ell_2 : \tau_2) \in \mathsf{h}\,. \\
\qquad (\text{if } \mathtt{x}_1 \neq \mathtt{x}_2 \text{ and } \ell_1 = \ell_2 \text{ then} \\
\qquad\quad \exists\, \ell'\,.\, \ell' \in \mathcal{L}(\mathtt{x}_1) \text{ and } \ell' \in \mathcal{L}(\mathtt{x}_2) \text{ and } \tau_1 <: \Gamma_\ell(\ell')) \text{ and} \\
\qquad (\text{if } \mathtt{x}_1 \neq \mathtt{x}_2 \text{ and } \ell_1 \neq \ell_2 \text{ then} \\
\qquad\quad \exists\, \ell'_1, \ell'_2\,.\, \ell'_1 \in \mathcal{L}(\mathtt{x}_1) \text{ and } \ell'_2 \in \mathcal{L}(\mathtt{x}_2) \text{ and } \ell'_1 \neq \ell'_2 \text{ and } \tau_1 <: \Gamma_\ell(\ell'_1) \text{ and } \tau_2 <: \Gamma_\ell(\ell'_2))
\end{array}
$$

This definition ensures that if two variables alias under any heap, then the alias lattice will reflect that by putting the same location $\ell'$ into each of their location lists. Likewise,

[4] By branch-sensitive, we mean that the true and false branches of a conditional may receive different lattice information depending upon the condition. This is not a path-sensitive analysis.

if the two variables are not aliased within a given heap, then the alias lattice will reflect this possibility as well by having a distinct location in each location set. The definition also ensures that the typing context $\Gamma_\ell$ has the most general type for a location.

If the alias analysis ensures Def. 1 and can provide the required interface, the variants of the FUSION analysis are provably sound or complete. Additionally, a more precise alias analysis will increase the precision of the FUSION analysis.

### 3.1 The Relation Lattice

We track the status of a relationship using the four-point dataflow lattice represented in Fig. 3, where `unknown` represents either true or false and `bot` is a special case used only inside the flow function. The FUSION analysis uses a tuple lattice which maps all relationships we want to track to a relationship state lattice element. We will represent this tuple lattice as $\rho$. We will say that $\rho$ is *consistent* with an alias lattice $\mathcal{A}$ when the domain of $\rho$ is equal to the set of relationships that are possible under $\mathcal{A}$.

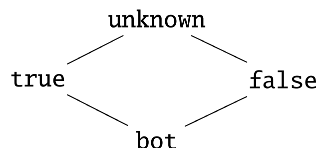



**Fig. 3:** The relationship state lattice

Notice that as more references enter the context, there are more possible relationships, and the height of $\rho$ grows. Even so, the height is always finite as there is a finite number of locations and a finite number of relations. As the flow function is monotonic, the analysis always reaches a fix-point.

### 3.2 Flow Function

The analysis flow function is responsible for two tasks; it must check that a given operation is valid, and it must apply any specified relationship effects to the lattice. The flow function is defined as

$$f_{\mathcal{C};\mathcal{A};\mathcal{B}}(\rho, \mathtt{instr}) = \rho'$$

where $\mathcal{C}$ is all the constraints, $\mathcal{A}$ is the alias lattice, $\mathcal{B}$ is the boolean constant lattice, $\rho$ is the starting relation lattice, $\rho'$ is the ending relation lattice, and $\mathtt{instr}$ is the instruction the analysis is currently checking. The analysis goes through each constraint in $\mathcal{C}$ and checks for a match. It first checks to see whether the operation defined by the constraint matches the instruction, thus representing a syntactic match. It also checks to see whether $\rho$ determines that the trigger of the constraint applies. If so, it has both a syntactic and semantic match, and it binds the specification variables to the locations that triggered the match. These bindings will be used for the remaining steps.

Once the analysis has a match, two things must occur. First, it uses the bindings generated above to show that the requires predicate of the constraint is true under $\rho$. If it is not true, then the analysis reports an error on $\mathtt{instr}$. Second, the analysis must use the same bindings to produce $\rho'$ by applying the relationship effects.

### 3.3 Soundness and Completeness

Soundness and completeness allow the user of the analysis to either have confidence that there are no errors at runtime if the analysis finds none (if it is sound) or that any errors

**Table 1:** Differences between sound, complete, and pragmatic variants

| Variant | Trigger Predicate checks when... | Requires Predicate passes when... |
|---|---|---|
| Sound | True or Unknown | True |
| Complete | True | True or Unknown |
| Pragmatic | True | True |

the analysis finds will actually occur in some runtime scenario (if it is complete). For the purposes of these definitions, an error is a dynamic interpretation of the constraint which causes the requires predicate to fail. In the formal semantics, an error is signaled as a failure for the flow function to generate a new lattice for a particular instruction.

We define soundness and completeness of the FUSION analysis by assuming an alias analysis which abstracts the heap using $\mathcal{A}$, as described above. For both of these theorems, we let $\mathcal{A}^{conc}$ define the actual heap at some point of a real execution, and we let $\mathcal{A}^{abs}$ be a sound approximation of $\mathcal{A}^{conc}$. We also let $\rho^{abs}$ and $\rho^{conc}$ be relationship lattices consistent with $\mathcal{A}^{abs}$ and $\mathcal{A}^{conc}$ where $\rho^{abs}$ is an abstraction of the concrete runtime lattice $\rho^{conc}$, defined as $\rho^{conc} \sqsubseteq \rho^{abs}$.

For the sound variant, we expect that if the flow function generates a new lattice using the imprecise lattice $\rho^{abs}$, then any more concrete lattice will also produce a new lattice for that instruction. As the flow function only generates a new lattice if it finds no errors, then there may be false positives from when $\rho^{abs}$ produces errors, but there will be no false negatives. To be locally sound for this instruction, the new abstract lattice must conservatively approximate any new concrete lattice. Thm. 1 captures the intuition of local soundness formally. Global soundness follows from local soundness, the monotonicity of the flow function, and the initial conditions of the lattice.

**Theorem 1 (Local Soundness of Relations Analysis).**
if $f_{\mathcal{C};\mathcal{A}^{abs};\mathcal{B}}(\rho^{abs}, \mathit{instr}) = \rho^{abs\prime}$ and $\rho^{conc} \sqsubseteq \rho^{abs}$
then $f_{\mathcal{C};\mathcal{A}^{conc};\mathcal{B}}(\rho^{conc}, \mathit{instr}) = \rho^{conc\prime}$ and $\rho^{conc\prime} \sqsubseteq \rho^{abs\prime}$

If the FUSION analysis is complete, we expect a theorem which is the opposite of the soundness theorem and is shown in Thm. 2. If a flow function generates a new lattice given a lattice $\rho^{conc}$, then it will also generate a new lattice on any abstraction of $\rho^{conc}$. An analysis with this property may produce false negatives, as the analysis can find an error using the concrete lattice yet generate a new lattice using $\rho^{abs}$, but it will produce no false positives. Like the sound analysis, the resulting lattices must maintain their existing precision relationship.

**Theorem 2 (Local Completeness of Relations Analysis).**
if $f_{\mathcal{C};\mathcal{A}^{conc};\mathcal{B}}(\rho^{conc}, \mathit{instr}) = \rho^{conc\prime}$ and $\rho^{conc} \sqsubseteq \rho^{abs}$
then $f_{\mathcal{C};\mathcal{A}^{abs};\mathcal{B}}(\rho^{abs}, \mathit{instr}) = \rho^{abs\prime}$ and $\rho^{conc\prime} \sqsubseteq \rho^{abs\prime}$

The FUSION analysis can be either sound, complete, or pragmatic by making only minor changes to the analysis. Proofs of soundness and completeness, for the sound and complete variants respectively, can be found in our associated technical report [5]. The differences between the variants are summarized in Tab. 1 and are described below.

**Trigger condition.** The trigger predicate determines when the constraint will check

```
public class ListItemCollection {
    @Item({*, target}, REMOVE)
    public void clear() {...}
}
```

```
@Constraint(op = "ListItemCollection.clear()",
    trigger = "x instanceof ListItem",
    requires = "true",
    effect = {"!Item(x, target)"})
```

**Fig. 4:** Translating a relation effect with wildcards into a constraint. The Item relation has type `Item(ListItem, ListItemCollection)`.

the requires predicate and when it will produce effects. The sound variant will trigger a constraint whenever there is even a possibility of it triggering at runtime. Therefore, it triggers when the predicate is either true or unknown. The complete variant can produce no false positives, so it will only check the requires predicate when the trigger predicate is definitely true. Regardless of the variant, if the trigger is either true or unknown, the analysis produces a set of changes to make to the lattice based upon the effects list. The pragmatic variant will work the same as the complete variant when determining whether to trigger the constraint. The rationale here is to try to reduce the number of false positives by only checking constraints when they are known to be applicable.

**Error condition.** The requires predicate should be true to signal that the operation is safe to use. The sound variant will cause an error whenever the requires predicate is false or unknown. The complete variant, however, can only cause an error if it is sure there is one, so it only flags an error if the requires predicate is definitely false. In this case, the pragmatic variant will work the same as the sound variant. If the analysis has come to this point, it already has enough information to determine that the trigger was true. Therefore, we will require that the plugin definitely show that the requires predicate is true, with the expectation that this will reduce the false negatives.

While the pragmatic variant can produce false positives and false negatives, we believe it will be the most cost-effective in practice based on our experience described in Sect. 5. Additionally, this variant may use inferred relationships, a feature which is not sound or complete but reduces the specification burden on the framework developer.

# 4 Abstract Semantics of FUSION

In this section, we present formal semantics for a simplified version of the specifications and analysis, the grammar for which is shown in Fig. 5. We do not formalize `TEST` effects or specialized relations for equality (==) and typing (instanceof). A semantics with `TEST` effects can be found in our technical report [5], and it is possible to add specialized relations by calling out to other flow analyses in the same manner as is done with both `TEST` effects and aliasing.

Relationship effects and wildcards are both syntactic sugar that can be easily translated into a constraint form. Relationship effects are translated by considering them as a constraint on the annotated method with a `true` trigger predicate, a `true` requires predicate, and the effect list as annotated. Wildcards are easily rewritten by declaring a fresh variable in the trigger predicate and constraining it to have the desired type. Fig. 4 shows an example effect with a wildcard translated into a constraint.

| | |
|---|---|
| constraint | $cons ::= op : P_{trg} \Rightarrow P_{req} \Downarrow \overline{R}$ |
| predicate | $P ::= P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \implies P_2 \mid R \mid true \mid false$ |
| relation predicate | $R ::= rel(\bar{\alpha}) \mid \neg rel(\bar{\alpha})$ |
| source instruction | $instr ::= x_{rslt} = x_{tgt}.m(\overline{x}) \mid x_{rslt} = \text{new}\ \tau(\overline{x}) \mid eom \mid \ldots$ |
| instruction signature | $op ::= \tau_{tgt}.m(\overline{y} : \overline{\tau}) : \tau_{rslt} \mid \text{new}\ \tau(\overline{y} : \overline{\tau}) \mid \text{end-of-method} \mid \ldots$ |
| meta variable | $\alpha ::= y \mid \ell$ |
| ternary logic | $t ::= \text{True} \mid \text{False} \mid \text{Unknown}$ |
| lattice elements | $E ::= \text{unknown} \mid \text{true} \mid \text{false} \mid \text{bot}$ |
| flow lattice | $\rho ::= rel(\overline{\ell}) \mapsto E, \rho \mid \varnothing$ |
| set of lattices | $\mathcal{P} ::= \{\rho\} \cup \mathcal{P} \mid \varnothing$ |
| substitution | $\sigma ::= (y \mapsto \ell), \sigma \mid \varnothing$ |
| set of substitutions | $\Sigma ::= \{\sigma\} \cup \Sigma \mid \varnothing$ |
| alias lattice | $\mathcal{A} ::= < \Gamma_\ell ; \mathcal{L} >$ |
| aliases | $\mathcal{L} ::= (x \mapsto \overline{\ell}), \mathcal{L} \mid \varnothing$ |
| location types | $\Gamma_\ell ::= (\ell : \tau), \Gamma_\ell \mid \varnothing$ |
| spec variable types | $\Gamma_y ::= (y : \tau), \Gamma_y \mid \varnothing$ |
| relation type | $\mathcal{R} ::= rel \mapsto \bar{\tau}, \mathcal{R} \mid \varnothing$ |
| constraints | $\mathcal{C} ::= cons, \mathcal{C} \mid \varnothing$ |
| relation inference rules | $\mathcal{I} ::= P \Downarrow \overline{R}, \mathcal{I} \mid \varnothing$ |

x is a source variable
m is a method name
rel is a relation name
$\tau$ is a type
y is a spec variable, where the variables target and result have special meanings
$\ell$ is a label for a runtime object
$\bot_{\mathcal{A}}$ is a $\rho$ where $\bot_{\mathcal{A}}$ is consistent with $\mathcal{A}$ and $\forall\ rel(\overline{\ell}) \mapsto E \in \bot_{\mathcal{A}}\ .\ E = \text{bot}$

**Fig. 5:** Abstract syntax

The lattice $\rho$ has the usual operators of join ($\sqcup$) and comparison ($\sqsubseteq$), which work as expected for a tuple lattice. We also introduce three additional operators, defined in Fig. 6. Equivalence join ( $\boxminus$ ) will resolve to unknown if the two sides are not equal. Overriding meet ( $\leftarrow$ ) has the property that if the right side has a defined value (not bot), then it will use the right value, otherwise it will use the left value. The polarity operator ($\updownarrow$) will push all non-bottom values to the top of the lattice. Finally, we also define $\bot_{\mathcal{A}}$ as a tuple lattice which is consistent with the alias lattice $\mathcal{A}$ and which maps every relationship to bot.

$$\frac{}{\updownarrow \mathtt{bot} = \mathtt{bot}}\text{(POLAR-BOT)} \qquad \frac{E \neq \mathtt{bot}}{\updownarrow E = \mathsf{unknown}}\text{(POLAR-UNKNOWN)} \qquad \frac{}{E \mathbin{\overset{=}{\sqcup}} E = E}\text{(EQJOIN-=)}$$

$$\frac{}{E \mathbin{\overleftarrow{\sqcup}} \mathtt{bot} = E}\text{(OVR-BOT)} \qquad \frac{E_r \neq \mathtt{bot}}{E_l \mathbin{\overleftarrow{\sqcup}} E_r = E_r}\text{(OVR-NOT-BOT)} \qquad \frac{E_l \neq E_r}{E_l \mathbin{\overset{=}{\sqcup}} E_r = \mathsf{unknown}}\text{(EQJOIN-}\neq\text{)}$$

**Fig. 6:** Unusual lattice operations

$$\boxed{\rho \vdash P\ t}$$

$$\frac{\rho(\mathsf{rel}(\bar{\ell})) = \mathtt{true}}{\rho \vdash \mathsf{rel}(\bar{\ell})\ \mathsf{True}}\text{(REL-T)} \qquad \frac{\rho(\mathsf{rel}(\bar{\ell})) = \mathtt{false}}{\rho \vdash \mathsf{rel}(\bar{\ell})\ \mathsf{False}}\text{(REL-F)}$$

$$\frac{\rho(\mathsf{rel}(\bar{\ell})) = E \qquad E \neq \mathtt{true} \qquad E \neq \mathtt{false}}{\rho \vdash \mathsf{rel}(\bar{\ell})\ \mathsf{Unknown}}\text{(REL-U-SND/CMP)}$$

$$\frac{\begin{array}{c}\rho(\mathsf{rel}(\bar{\ell})) = E \qquad E \neq \mathtt{true} \qquad E \neq \mathtt{false} \\ \rho \text{ infers } \rho' \qquad \rho \mathbin{\overleftarrow{\sqcup}} \rho' \vdash \mathsf{rel}(\bar{\ell})\ t \qquad t \text{ is True or False}\end{array}}{\rho \vdash \mathsf{rel}(\bar{\ell})\ t}\text{(INFER-PRG)}$$

$$\frac{\begin{array}{c}\rho(\mathsf{rel}(\bar{\ell})) = E \qquad E \neq \mathtt{true} \qquad E \neq \mathtt{false} \\ \neg\exists\, \rho' .\ \rho \text{ infers } \rho' \ \wedge\ \rho \mathbin{\overleftarrow{\sqcup}} \rho' \vdash \mathsf{rel}(\bar{\ell})\ t \ \wedge\ t \text{ is True or False}\end{array}}{\rho \vdash \mathsf{rel}(\bar{\ell})\ \mathsf{Unknown}}\text{(REL-U-PRG)}$$

$$\boxed{\rho \text{ infers } \rho'}$$

$$\frac{P \Downarrow \overline{R} \in \mathcal{I} \qquad \rho \vdash P[\sigma]\ \mathsf{True} \qquad \rho' = \mathit{lattice}(\overline{R}[\sigma]) \qquad \rho' \sqsubset \rho}{\rho \text{ infers } \rho'}\text{(DISCOVER)}$$

**Fig. 7:** Check predicate truth under a lattice. The remaining rules are as expected for ternary logic and can be found in [5].

### 4.1 Checking predicate truth

Before we show how constraint checking works, we must describe how the analysis tests the truth of a relationship predicate. The judgment for this is written as

$$\rho \vdash P\ t$$

and is read "the lattice $\rho$ shows that predicate $P$ is $t$", where $t$ is either True, False, or Unknown. The rules for this judgment are similar to three-valued logic, and the interesting subset of them are in Fig. 7.

In the sound and complete variants, the rules are trivial. The analysis inspects the lattice to see what the value of the relationship is to determine whether it is True (REL-T), False (REL-F), or Unknown (REL-U-SND/CMP). If the lattice maps the relationship to either

unknown or bot, then the predicate is considered Unknown. The rest of the predicate rules work as expected for a three-valued logic.

The interesting case is in the pragmatic variant when the relationship does not map to true or false. Instead of using the rule (REL-U-SND/CMP), the pragmatic variant admits the rules (REL-U-PRG) and (INFER-PRG). These rules attempt to use the inferred relationships, defined in Sect. 2.3, to retrieve the desired relationship. The rule for the inference judgement $\rho$ infers $\rho'$ is also defined in Fig. 7. This rule first checks to see if the trigger of an inferred relation is true, and if so, uses the function lattice to produce the inferred relationships described by $\bar{R}[\sigma]$. For all relationships not defined by $\bar{R}[\sigma]$, lattice defaults to bot to signal that there are no changes. There are two properties to note about the rules (REL-U-PRG), (INFER-PRG), and (DISCOVER):

1. The use of inferred relationships does not change the original lattice $\rho$. This allows the inferred relationships to disappear if the generator, $P$, is no longer true.
2. Any inferred values must be *strictly more precise* than the relationship's value in $\rho$, as enforced by $\rho' \sqsubset \rho$. This means that relationships can move from unknown to true, but they can not move from false to true. This property guarantees termination and gives declared effects precedence over inferred ones.

Inferred relationships can not be used in the sound and complete variants. This does not limit the expressiveness of the specifications, as inferred relations can always be written directly within the constraints. Doing so does make the specifications more difficult to write; the framework developer must add the inferred relations to any constraint which will also prove the trigger predicate. Since inferred relations do change the semantics, they are not syntactic sugar, but they are not necessary for reasons beyond the ease of writing specifications.

### 4.2 Matching on an operator

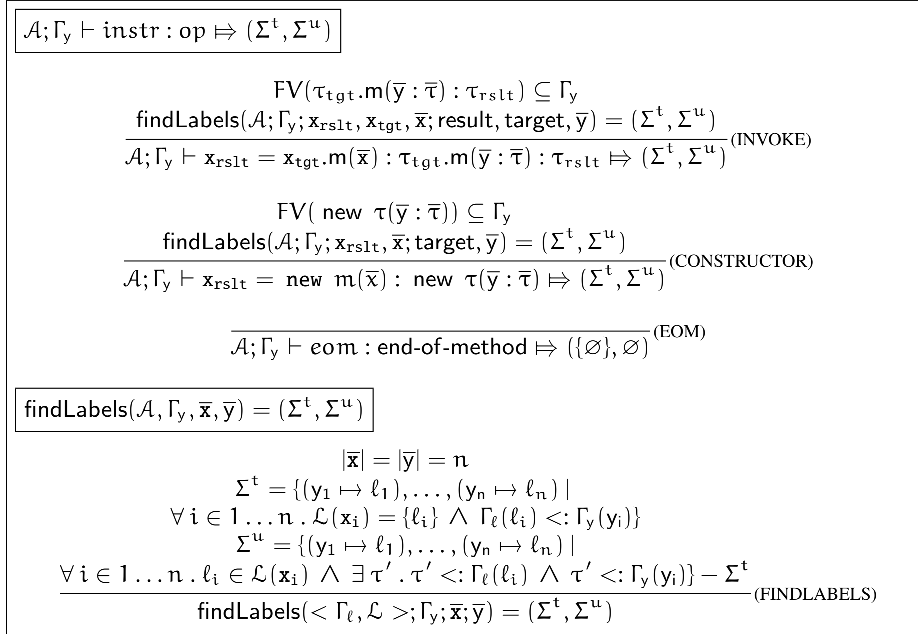
In order to check a constraint, the analysis must determine whether a source instruction, called instr, matches the syntactic operation op defined by a constraint. This is realized in the judgment

$$\mathcal{A}; \Gamma_y \vdash \mathtt{instr} : \mathsf{op} \Rrightarrow (\Sigma^t, \Sigma^u)$$

with rules defined in Fig. 8. Given the alias lattice $\mathcal{A}$ and a typing environment for the free variables in op, this judgment matches instr to op and produces two disjoint sets of substitutions that map specification variables in op to heap locations. The first set, $\Sigma^t$, represents possible substitutions where the locations are all known to be a subtype of the type required by the variables. The second set, $\Sigma^u$, are potential substitutions where the locations may or may not have the right type at runtime.

As an example, we will walk through the rule (INVOKE) in Fig. 8. The first premise checks that the free variables in op are in $\Gamma_y$, and the second premise builds the substitution set using the findLabels function. Each substitution in the set will map the specification variables in op (target, result, and $y_1 \ldots y_n$) to a location in the heap that is aliased by the appropriate source variables in instr ($x_{tgt}$, $x_{rslt}$, and $x_1 \ldots x_n$).

To produce the set $\Sigma^t$, the findLabels function must generate a substitution for each $y_i$ in $\bar{y}$. It starts by verifying that the corresponding source variable $x_i$ points to only



$$\boxed{\mathcal{A}; \Gamma_y \vdash instr : \mathrm{op} \mapsto (\Sigma^t, \Sigma^u)}$$

$$\frac{\mathsf{FV}(\tau_{tgt}.\mathsf{m}(\overline{y} : \overline{\tau}) : \tau_{rslt}) \subseteq \Gamma_y \qquad \mathsf{findLabels}(\mathcal{A}; \Gamma_y; \mathtt{x_{rslt}}, \mathtt{x_{tgt}}, \overline{\mathtt{x}}; \mathsf{result}, \mathsf{target}, \overline{y}) = (\Sigma^t, \Sigma^u)}{\mathcal{A}; \Gamma_y \vdash \mathtt{x_{rslt}} = \mathtt{x_{tgt}}.\mathtt{m}(\overline{\mathtt{x}}) : \tau_{tgt}.\mathsf{m}(\overline{y} : \overline{\tau}) : \tau_{rslt} \mapsto (\Sigma^t, \Sigma^u)} \text{(INVOKE)}$$

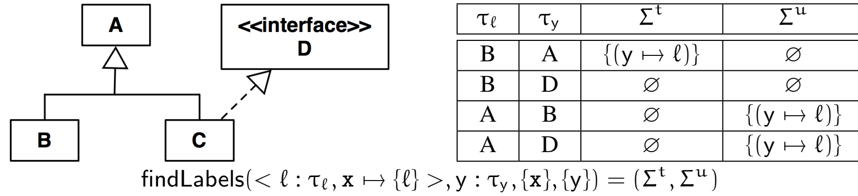
$$\frac{\mathsf{FV}(\text{ new } \tau(\overline{y} : \overline{\tau})) \subseteq \Gamma_y \qquad \mathsf{findLabels}(\mathcal{A}; \Gamma_y; \mathtt{x_{rslt}}, \overline{\mathtt{x}}; \mathsf{target}, \overline{y}) = (\Sigma^t, \Sigma^u)}{\mathcal{A}; \Gamma_y \vdash \mathtt{x_{rslt}} = \mathtt{new}\ m(\overline{\mathtt{x}}) : \text{ new } \tau(\overline{y} : \overline{\tau}) \mapsto (\Sigma^t, \Sigma^u)} \text{(CONSTRUCTOR)}$$

$$\frac{}{\mathcal{A}; \Gamma_y \vdash eom : \text{end-of-method} \mapsto (\{\varnothing\}, \varnothing)} \text{(EOM)}$$

$$\boxed{\mathsf{findLabels}(\mathcal{A}, \Gamma_y, \overline{\mathtt{x}}, \overline{y}) = (\Sigma^t, \Sigma^u)}$$

$$\frac{\begin{array}{c} |\overline{\mathtt{x}}| = |\overline{y}| = n \\ \Sigma^t = \{(y_1 \mapsto \ell_1), \ldots, (y_n \mapsto \ell_n) \mid \\ \forall\, i \in 1 \ldots n\,.\, \mathcal{L}(\mathtt{x_i}) = \{\ell_i\} \wedge \Gamma_\ell(\ell_i) <: \Gamma_y(y_i)\} \\ \Sigma^u = \{(y_1 \mapsto \ell_1), \ldots, (y_n \mapsto \ell_n) \mid \\ \forall\, i \in 1 \ldots n\,.\, \ell_i \in \mathcal{L}(\mathtt{x_i}) \wedge \exists\, \tau'\,.\, \tau' <: \Gamma_\ell(\ell_i) \wedge \tau' <: \Gamma_y(y_i)\} - \Sigma^t \end{array}}{\mathsf{findLabels}(< \Gamma_\ell, \mathcal{L} >; \Gamma_y; \overline{\mathtt{x}}; \overline{y}) = (\Sigma^t, \Sigma^u)} \text{(FINDLABELS)}$$

**Fig. 8:** Matching instructions to operations and type satisfaction




| $\tau_\ell$ | $\tau_y$ | $\Sigma^t$ | $\Sigma^u$ |
|---|---|---|---|
| B | A | $\{(y \mapsto \ell)\}$ | $\varnothing$ |
| B | D | $\varnothing$ | $\varnothing$ |
| A | B | $\varnothing$ | $\{(y \mapsto \ell)\}$ |
| A | D | $\varnothing$ | $\{(y \mapsto \ell)\}$ |

$\mathsf{findLabels}(< \ell : \tau_\ell, \mathtt{x} \mapsto \{\ell\} >, y : \tau_y, \{\mathtt{x}\}, \{y\}) = (\Sigma^t, \Sigma^u)$

**Fig. 9:** Examples of the difference between $\Sigma^t$ and $\Sigma^u$

one location $\ell$, and it checks to see if the type of that location is a subtype of the type required for $y_i$. Every substitution $\sigma$ which fits these requirements is in $\Sigma^t$.

$\Sigma^u$ is a more interesting set. Unlike $\Sigma^t$, it checks all locations which $\mathtt{x_i}$ aliases and records a possible substitution for each. Additionally, when it checks the type, it allows the location if there is even a *possibility* of it being the right type. As an example, consider the class hierarchy and use of findLabels shown in Fig. 9. In the first row, $\ell$ is definitely substitutable for y, so it is a substitution in $\Sigma^t$. In the second row, y can never be substituted by $\ell$, so both sets are empty. In the third and fourth rows, $\ell$ may be substitutable for y (if $\ell$ has type B or C, respectively), so both substitutions are possibly, but not definitely, allowed and are therefore in $\Sigma^u$.

The need for $\Sigma^u$ may seem surprising, but the rationale behind it is that framework constraints do not always adhere to behavioral subtyping [6]. Consider analyzing the `DropDownList` constraint on the code below:

```
ListControl list = ...;
ListItem item = list.getItems().findByValue("foo");
item.setSelected(true);
```

Since `list` is of type `ListControl`, the trigger clause of the first constraint in List. 7 will not be true, and the constraint will never trigger an error. However, we would like this to trigger a potential violation in the sound variant since `list` could be a `DropDownList`. The root of the problem was that `DropDownList` is not following the principle of behavioral subtyping; it has added preconditions to methods that the base class did not require. Therefore, a `DropDownList` is not always substitutable where a `ListControl` is used! While frustrating for verification, this is common in frameworks; by trading off substitutability, the framework developers received code reuse internally. Other verification proposals have also recognized the need to support broken behavioral subtyping for this reason [7, 8]. Inheritance was used here rather than composition because the type is structurally the same, and it is almost behaviorally the same. In fact, the methods on `DropDownList` itself do appear to be behaviorally substitutable. However, the subtype added a few constraints to *other* classes, like the `ListItem` class.

By keeping track of $\Sigma^{t}$ and $\Sigma^{u}$ separately, it will allow the variants of the analysis to use them differently. In particular, the sound variant will trigger errors from substitutions in $\Sigma^{u}$, while the complete and pragmatic variants will only use it to propagate lattice changes from the effect list.

### 4.3 Checking a single constraint

We will now show how the analysis checks an instruction for a single constraint. This is done with the judgment

$$\mathcal{A}; \rho; \mathtt{cons} \vdash \mathtt{instr} \hookrightarrow \rho^{\Delta}$$

shown in Fig. 10. This judgment takes the lattices and a constraint, and it determines what changes to make to the relation lattice for the given instruction. The lattice changes are represented in $\rho^{\Delta}$, where a relationship mapped to `bot` signifies no changes.

The analysis starts by checking whether the instruction matches the constrained operation. If not, the instruction matching rules will return no substitutions and the rule (NO-MATCH) will apply. If there are substitutions, as shown in rule (MATCH), then the analysis must check this constraint for every aliasing configuration possible, as represented by $\Sigma^{t}$ and $\Sigma^{u}$. This rule checks that for each substitution $\sigma$, the constraint passes and produces a change lattice $\rho^{\Delta}$. If the substitution was from $\Sigma^{u}$, then the analysis must use the $\updownarrow$ operator on $\rho^{\Delta}$. This is done because the analysis cannot be sure if the substitution is valid at runtime, so it can only make changes into `unknown`. Setting all changes to `unknown` could cause the analysis to lose precision when $\rho^{\Delta}$ prescribes a change that already exists in $\rho$. A possible solution is to let the polarizing operator return `bot` if the prescribed changes already exist in the lattice $\rho$, but we have not yet proven this extension is sound.

The last step the rule makes is to combine all the lattice changes, from all substitutions, using $\boxminus$. The use of $\boxminus$ means that a change is only made to `true` or `false` if all the aliasing configurations agree to it. Likewise, a signal to make no changes by

$$\boxed{\mathcal{A};\rho;\mathsf{cons} \vdash \mathsf{instr} \hookrightarrow \rho^{\Delta}}$$

$$\frac{\begin{array}{c}\mathsf{cons} = \mathsf{op} : \mathsf{P}_{\mathsf{trg}} \Rightarrow \mathsf{P}_{\mathsf{req}} \Downarrow \overline{\mathsf{R}} \qquad \mathcal{A};\mathsf{FV}(\mathsf{cons}) \vdash \mathsf{instr} : \mathsf{op} \mapsto (\Sigma^{t}, \Sigma^{u}) \\ \Sigma^{t} \cup \Sigma^{u} \neq \varnothing \qquad \mathcal{P}^{t} = \{\rho^{\Delta} \mid \sigma \in \Sigma^{t} \ \wedge\ \mathcal{A};\rho;\sigma \vdash_{\mathsf{part}} \mathsf{cons} \hookrightarrow \rho^{\Delta}\} \\ \mathcal{P}^{u} = \{\updownarrow \rho^{\Delta} \mid \sigma \in \Sigma^{u} \ \wedge\ \mathcal{A};\rho;\sigma \vdash_{\mathsf{part}} \mathsf{cons} \hookrightarrow \rho^{\Delta}\} \\ |\Sigma^{t}| = |\mathcal{P}^{t}| \qquad |\Sigma^{u}| = |\mathcal{P}^{u}| \qquad \mathcal{P}^{\Delta} = \mathcal{P}^{t} \cup \mathcal{P}^{u}\end{array}}{\mathcal{A};\rho;\mathsf{cons} \vdash \mathsf{instr} \hookrightarrow (\biguplus \mathcal{P}^{\Delta})}\ \text{(MATCH)}$$

$$\frac{\mathsf{cons} = \mathsf{op} : \mathsf{P}_{\mathsf{trg}} \Rightarrow \mathsf{P}_{\mathsf{req}} \Downarrow \overline{\mathsf{R}} \qquad \mathcal{A};\mathsf{FV}(\mathsf{cons}) \vdash \mathsf{instr} : \mathsf{op} \mapsto (\varnothing, \varnothing)}{\mathcal{A};\rho;\mathsf{cons} \vdash \mathsf{instr} \hookrightarrow \bot_{\mathcal{A}}}\ \text{(NO-MATCH)}$$

$$\boxed{\mathcal{A};\rho;\sigma \vdash_{\mathsf{part}} \mathsf{cons} \hookrightarrow \rho^{\Delta}}$$

$$\frac{\begin{array}{c}\mathsf{cons} = \mathsf{op} : \mathsf{P}_{\mathsf{trg}} \Rightarrow \mathsf{P}_{\mathsf{req}} \Downarrow \overline{\mathsf{R}} \\ \Gamma_{y} = \mathsf{FV}(\mathsf{op}) \cup \mathsf{FV}(\mathsf{P}_{\mathsf{trg}}) \cup \mathsf{FV}(\overline{\mathsf{R}}) \qquad \mathsf{allValidSubs}(\mathcal{A};\sigma_{\mathsf{op}};\Gamma_{y}) = (\Sigma^{t}, \Sigma^{u}) \\ \Sigma^{t} \cup \Sigma^{u} \neq \varnothing \qquad \mathcal{P}^{t} = \{\rho^{\Delta} \mid \sigma \in \Sigma^{t} \ \wedge\ \mathcal{A};\rho;\sigma \vdash_{\mathsf{full}} \mathsf{cons} \hookrightarrow \rho^{\Delta}\} \\ \mathcal{P}^{u} = \{\updownarrow \rho^{\Delta} \mid \sigma \in \Sigma^{u} \ \wedge\ \mathcal{A};\rho;\sigma \vdash_{\mathsf{full}} \mathsf{cons} \hookrightarrow \rho^{\Delta}\} \\ |\Sigma^{t}| = |\mathcal{P}^{t}| \qquad |\Sigma^{u}| = |\mathcal{P}^{u}| \qquad \mathcal{P}^{\Delta} = \mathcal{P}^{t} \cup \mathcal{P}^{u}\end{array}}{\mathcal{A};\rho;\sigma_{\mathsf{op}} \vdash_{\mathsf{part}} \mathsf{cons} \hookrightarrow (\biguplus \mathcal{P}^{\Delta})}\ \text{(BOUND)}$$

$$\frac{\begin{array}{c}\mathsf{cons} = \mathsf{op} : \mathsf{P}_{\mathsf{trg}} \Rightarrow \mathsf{P}_{\mathsf{req}} \Downarrow \overline{\mathsf{R}} \\ \Gamma_{y} = \mathsf{FV}(\mathsf{op}) \cup \mathsf{FV}(\mathsf{P}_{\mathsf{trg}}) \cup \mathsf{FV}(\overline{\mathsf{R}}) \qquad \mathsf{allValidSubs}(\mathcal{A};\sigma_{\mathsf{op}};\Gamma_{y}) = (\varnothing, \varnothing)\end{array}}{\mathcal{A};\rho;\sigma_{\mathsf{op}} \vdash_{\mathsf{part}} \mathsf{cons} \hookrightarrow \bot_{\mathcal{A}}}\ \text{(CANT-BIND)}$$

$$\boxed{\mathsf{allValidSubs}(\mathcal{A};\sigma;\Gamma_{y}) = (\Sigma^{t}, \Sigma^{u})}$$

$$\frac{\begin{array}{c}\Sigma^{t} = \{\sigma' \mid \sigma' \supseteq \sigma \ \wedge\ \mathsf{dom}(\sigma') = \mathsf{dom}(\Gamma_{y}) \ \wedge\ \forall\, y \mapsto \ell \in \sigma' \,.\, \Gamma_{\ell}(\ell) <: \Gamma_{y}(y)\} \\ \Sigma^{u} = \{\sigma' \mid \sigma' \supseteq \sigma \ \wedge\ \mathsf{dom}(\sigma') = \mathsf{dom}(\Gamma_{y}) \ \wedge \\ \forall\, y \mapsto \ell \in \sigma' \,.\, \exists\, \tau' \,.\, \tau' <: \Gamma_{\ell}(\ell) \ \wedge\ \tau' <: \Gamma_{y}(y)\} - \Sigma^{t}\end{array}}{\mathsf{allValidSubs}(< \Gamma_{\ell};\mathcal{L} >;\sigma;\Gamma_{y}) = (\Sigma^{t}, \Sigma^{u})}\ \text{(VALIDSUBS)}$$

**Fig. 10:** Checking a single constraint

way of bot must also show in all configurations. If any configurations disagree about a lattice change, then the lattice element changes to unknown.

Once the analysis has a syntactic match, it tries to find the aliasing configurations for a semantic match using

$$\mathcal{A};\rho;\sigma \vdash_{\mathsf{part}} \mathsf{cons} \hookrightarrow \rho^{\Delta}$$

The analysis must get all aliasing configurations that are consistent with the current aliases in $\sigma$ and the types of the remaining free variables in $\mathsf{cons}$. The substitutions are found by the allValidSubs function, shown in Fig. 10. The rule (BOUND) proceeds in a similar manner to the rule (MATCH), except it checks the constraint using the judgment

$$\mathcal{A};\rho;\sigma \vdash_{\mathsf{full}} \mathsf{cons} \hookrightarrow \rho^{\Delta}$$

The rules for this judgment, shown in Fig. 11, are the primary point of difference between the variants of the analysis.

**Sound Variant.** The sound variant first checks $P_{trg}[\sigma]$ under $\rho$. It uses this to determine which rule applies. If $P_{trg}[\sigma]$ is True, as seen in rule (FULL-T-SND), then the analysis must check if $P_{req}$ is True under $\rho$ given any substitution. Since this is the sound variant, it will only accept substitutions from $\Sigma^t$. If $P_{req}$ is not True with a substitution from $\Sigma^t$, then the analysis produces an error. If there is no error, the rule produces the effects dictated by $\bar{R}[\sigma]$. The function lattice simply converts this list to a lattice, where all unspecified relationships map to bot. If $P_{trg}[\sigma]$ is False, then the analysis uses rule (FULL-F-SND). In this situation the constraint does not trigger, so the requires predicate is not checked and the analysis returns no changes using $\perp_{\mathcal{A}}$.

In the case that $P_{trg}[\sigma]$ is Unknown, the sound variant proceeds in a similar manner to the case where $P_{trg}[\sigma]$ is True as it must consider the possibility that the trigger predicate is actually true. In fact the only difference in the rule (FULL-U-SND) is that the analysis must use the polarizing operator to be conservative with the effects it is producing in case the trigger predicate is actually false at runtime.

**Complete Variant.** Like the sound variant, the complete variant starts by checking $P_{trg}[\sigma]$ under $\rho$. If $P_{trg}[\sigma]$ is True, as seen in rule (FULL-T-CMP), then the analysis must check $P_{req}$ under $\rho$ given any substitution. As this is the complete variant, the analysis does not care whether the substitution came from $\Sigma^t$ or $\Sigma^u$, and it does not matter whether $P_{req}$ is True or Unknown. If no substitutions work, either because none exist or because they all show $P_{req}$ to be false, then the analysis produces an error. Otherwise, the rule produces some effects. Since the constraint trigger was true, it will produce exactly the effects dictated by $\bar{R}[\sigma]$. If the analysis determines that $P_{trg}[\sigma]$ is False, then it uses the rule (FULL-F-CMP). Like the sound variant, the requires predicate is not checked and the analysis returns no changes.

Finally, if $P_{trg}[\sigma]$ is Unknown, the complete variant will not check $P_{req}$ as it cannot be sure whether the constraint is actually triggered and it should not produce an error. However, it must still produce some conservative effects in case the constraint is triggered given a more concrete lattice. Like the sound rule in the case of an unknown trigger, the rule uses the polarizing operator $\updownarrow$ to produce only conservative effects.

**Pragmatic Variant.** The pragmatic variant is a combination of the sound and complete variants. It has the same rule for False as the other two variants, (FULL-F-PRG). The rule (FULL-T-PRG) is the same as the True rule for soundness, while the rule (FULL-U-PRG) is the same as the Unknown rule for completeness. This means that this variant can produce both false positives and false negatives. False negatives can occur when $P_{trg}$ is Unknown under $\rho$, but a more precise lattice would have found $P_{trg}$ to be True and eventually generated an error. False positives occur when $P_{trg}$ is True under $\rho$ and $P_{req}$ is Unknown under $\rho$, but $P_{req}$ would have been True under a more precise lattice.

### 4.4 The flow function

The flow function for the FUSION analysis checks all the individual constraints and produces the output lattice for the instruction. Using the judgments defined in the previous section, the flow function iterates through each constraint and receives a change lattice. As shown in below, these lattices are combined using the join operator. Once

$\boxed{\mathcal{A}; \rho; \sigma \vdash_{\mathsf{full}} \mathit{cons} \hookrightarrow \rho^{\Delta}, \text{Sound Variant}}$

$$\frac{\begin{array}{c}\mathit{cons} = \mathit{op} : P_{trg} \Rightarrow P_{req} \Downarrow \overline{R} \qquad \rho \vdash P_{trg}[\sigma]\ \mathsf{True} \\ \mathsf{allValidSubs}(\mathcal{A}; \sigma; \mathsf{FV}(\mathit{cons})) = (\Sigma^{t}, \Sigma^{u}) \\ \exists\, \sigma' \in \Sigma^{t}\ .\ \rho \vdash P_{req}[\sigma']\ \mathsf{True}\end{array}}{\mathcal{A}; \rho; \sigma \vdash_{\mathsf{full}} \mathit{cons} \hookrightarrow \mathit{lattice}(\bar{R}[\sigma])}\ \text{(FULL-T-SND)}$$

$$\frac{\mathit{cons} = \mathit{op} : P_{trg} \Rightarrow P_{req} \Downarrow \overline{R} \qquad \rho \vdash P_{trg}[\sigma]\ \mathsf{False}}{\mathcal{A}; \rho; \sigma \vdash_{\mathsf{full}} \mathit{cons} \hookrightarrow \bot_{\mathcal{A}}}\ \text{(FULL-F-SND)}$$

$$\frac{\begin{array}{c}\mathit{cons} = \mathit{op} : P_{trg} \Rightarrow P_{req} \Downarrow \overline{R} \qquad \rho \vdash P_{trg}[\sigma]\ \mathsf{Unknown} \\ \mathsf{allValidSubs}(\mathcal{A}; \sigma; \mathsf{FV}(\mathit{cons})) = (\Sigma^{t}, \Sigma^{u}) \\ \exists\, \sigma' \in \Sigma^{t}\ .\ \rho \vdash P_{req}[\sigma']\ \mathsf{True} \qquad \rho^{\Delta} = \mathit{lattice}(\bar{R}[\sigma])\end{array}}{\mathcal{A}; \rho; \sigma \vdash_{\mathsf{full}} \mathit{cons} \hookrightarrow\!\updownarrow \rho^{\Delta}}\ \text{(FULL-U-SND)}$$

$\boxed{\mathcal{A}; \rho; \sigma \vdash_{\mathsf{full}} \mathit{cons} \hookrightarrow \rho^{\Delta}, \text{Complete Variant}}$

$$\frac{\begin{array}{c}\mathit{cons} = \mathit{op} : P_{trg} \Rightarrow P_{req} \Downarrow \overline{R} \qquad \rho \vdash P_{trg}[\sigma]\ \mathsf{True} \\ \mathsf{allValidSubs}(\mathcal{A}; \sigma; \mathsf{FV}(\mathit{cons})) = (\Sigma^{t}, \Sigma^{u}) \\ \exists\, \sigma' \in \Sigma^{t} \cup \Sigma^{u}\ .\ \rho \vdash P_{req}[\sigma']\ \mathsf{True} \vee \rho \vdash P_{req}[\sigma']\ \mathsf{Unknown}\end{array}}{\mathcal{A}; \rho; \sigma \vdash_{\mathsf{full}} \mathit{cons} \hookrightarrow \mathit{lattice}(\bar{R}[\sigma])}\ \text{(FULL-T-CMP)}$$

$$\frac{\mathit{cons} = \mathit{op} : P_{trg} \Rightarrow P_{req} \Downarrow \overline{R} \qquad \rho \vdash P_{trg}[\sigma]\ \mathsf{False}}{\mathcal{A}; \rho; \sigma \vdash_{\mathsf{full}} \mathit{cons} \hookrightarrow \bot_{\mathcal{A}}}\ \text{(FULL-F-CMP)}$$

$$\frac{\begin{array}{c}\mathit{cons} = \mathit{op} : P_{trg} \Rightarrow P_{req} \Downarrow \overline{R} \qquad \rho \vdash P_{trg}[\sigma]\ \mathsf{Unknown} \\ \rho^{\Delta} = \mathit{lattice}(\bar{R}[\sigma])\end{array}}{\mathcal{A}; \rho; \sigma \vdash_{\mathsf{full}} \mathit{cons} \hookrightarrow\!\updownarrow \rho^{\Delta}}\ \text{(FULL-U-CMP)}$$

$\boxed{\mathcal{A}; \rho; \sigma \vdash_{\mathsf{full}} \mathit{cons} \hookrightarrow \rho^{\Delta}, \text{Pragmatic Variant}}$

$$\frac{\begin{array}{c}\mathit{cons} = \mathit{op} : P_{trg} \Rightarrow P_{req} \Downarrow \overline{R} \qquad \rho \vdash P_{trg}[\sigma]\ \mathsf{True} \\ \mathsf{allValidSubs}(\mathcal{A}; \sigma; \mathsf{FV}(\mathit{cons})) = (\Sigma^{t}, \Sigma^{u}) \\ \exists\, \sigma' \in \Sigma^{t}\ .\ \rho \vdash P_{req}[\sigma']\ \mathsf{True}\end{array}}{< \Gamma_{\ell}; \mathcal{L} >; \rho; \sigma \vdash_{\mathsf{full}} \mathit{cons} \hookrightarrow \mathit{lattice}(\bar{R}[\sigma])}\ \text{(FULL-T-PRG)}$$

$$\frac{\mathit{cons} = \mathit{op} : P_{trg} \Rightarrow P_{req} \Downarrow \overline{R} \qquad \rho \vdash P_{trg}[\sigma]\ \mathsf{False}}{\mathcal{A}; \rho; \sigma \vdash_{\mathsf{full}} \mathit{cons} \hookrightarrow \bot_{\mathcal{A}}}\ \text{(FULL-F-PRG)}$$

$$\frac{\begin{array}{c}\mathit{cons} = \mathit{op} : P_{trg} \Rightarrow P_{req} \Downarrow \overline{R} \qquad \rho \vdash P_{trg}[\sigma]\ \mathsf{Unknown} \\ \rho^{\Delta} = \mathit{lattice}(\bar{R}[\sigma])\end{array}}{\mathcal{A}; \rho; \sigma \vdash_{\mathsf{full}} \mathit{cons} \hookrightarrow\!\updownarrow \rho^{\Delta}}\ \text{(FULL-U-PRG)}$$

**Fig. 11:** Checking a fully bound constraint and producing effects. Shading highlights the differences between the three variants.

the analysis has the final change lattice $\rho^{\Delta}$, it applies the changes using the overriding meet operation. This will preserve the old values of a relationship if the change lattice maps to `bot`, but it will override the old value otherwise. This provides us with the new relationship lattice $\rho'$, which is used by the dataflow analysis to feed into the next instruction's flow function. This flow function is monotonic, and the lattice has a finite height, so the dataflow analysis will reach a fix point.

$$\frac{\forall\, \mathtt{cons}_i \in \mathcal{C} \,.\, \mathcal{A}'; \rho; \mathtt{cons}_i \vdash \mathtt{instr} \hookrightarrow \rho_i^{\Delta} \qquad \rho^{\Delta} = \sqcup\{\rho_i^{\Delta}\} \qquad (i \in 1 \ldots n)}{f_{\mathcal{C};\mathcal{A}}(\rho, \mathtt{instr}) = \rho \,\overleftarrow{\sqcap}\, \rho^{\Delta}}$$

## 5 Implementation and Experience with the Pragmatic Variant

We implemented the pragmatic variant of the FUSION analysis in the Crystal dataflow analysis framework, an Eclipse plugin developed at Carnegie Mellon University for statically analyzing Java source [5]. The implementation interfaces to a boolean constant propagation analysis and a basic alias analysis; either of these could be replaced with more sophisticated implementation in order to improve the results.

We specified three sets of constraints, one for the ASP.NET framework[6] and two for the Eclipse JDT framework. These were all constraints which we had misused ourselves and were common problems that were posted on the help forums and mailing lists. These constraints exercised several different patterns, and the specifications were able to capture each of these patterns.

The specifications allowed us to easily describe structured relationships, such as the `ListItem`s which are in a `DropDownList` and a tree of `ASTNode`s within the Eclipse JDT. In each of these cases, a relationship ties the "child" and "parent" objects together, and it is straightforward to check if two children have the same parent. Two of our constraints had a structured relationship where an operation required that some objects exist (or do not exist) in a structured relationship.

All three constraints had semantics which required operations to occur in a particular order. To define this pattern, we needed a relationship which binds relevant objects together. The operation which occurs first produces an effect which sets this relationship to true, and the operation which must occur second requires this relationship. An example of this was seen in the constraints on the `DropDownList` in List. 7. Additionally, relationships allowed us to specify partial orderings of operations. One of the Eclipse JDT constraints had this behavior, and in fact required three methods to be called before the constrained operation. Alternatively, the user could choose to call a fourth method that would replace all three method calls. We captured this constraint by having each of the four methods produce a relationship, and the constrained operation simply required either the three relationships produced from the group of three methods, or the single relationship produced from the fourth one.

Relationships also made it straightforward to associate any objects that were used in the same operation. For example, this allowed us to associate several fields of an object

[5] `http://code.google.com/p/crystalsaf`
[6] We translated the relevant parts of the API and the examples into Java.

so that we could later check that they were only used together. We did this by annotating the constructor of the object with a relationship effect that tied the field parameters together. We could also associate objects that were linked by some secondary object, but had no direct connection, such as a `DropDownList` and the `ListItems` received from calls to the associated `ListItemCollection`.

After specifying the constraints, we ran the pragmatic variant on 20 examples based on real-world code. The examples we selected are based on our own misuses of these frameworks and on several postings on internet help forums and mailing lists. Of these, the pragmatic variant worked properly on 16, meaning that it either found an expected error or did not find an error on correct code. Most of these examples had little aliasing and used exact types, which reflected what we saw on the help forums.

These examples identified two sources of imprecision. The pragmatic variant failed on one example because the example used an unconstrained supertype, and it failed on the remaining three examples because the constraint required objects which were not in scope. The unconstrained supertype resulted in a false negative, and the three examples with objects out of scope resulted in false positives. In all four of these cases, the sound variant would have flagged an error, and the complete variant would not have.

Unconstrained supertypes, such as using a `ListControl` instead of a `DropDown-List`, are the first potential source of imprecision for the pragmatic variant. While a sound analysis would have detected this type of error, in practice, using this superclass is not typical as it only exists for code reuse purposes. In fact, we never found code on the forum that used the superclass `ListControl`.

The more interesting, and more typical, source of imprecision occurs when a required object is not in scope. For example, one of the Eclipse JDT constraints required that an `ASTNode` have a relationship with an `AST` object. The plugin, however, did not have any `AST` objects in scope at all, even though this relationship did exist globally. Based on the examples we found, this does occur in practice, typically when the framework makes multiple callbacks in sequence, such as with a Visitor pattern.

Future revisions of the FUSION analysis could address the problem of out-of-scope objects with two changes. First, it should be possible for the framework to declare what relationships exist at the point where the callback occurs. This would have provided the correct relationships in the previous example, and it should be relatively straightforward to annotate the interface of the plugin with this information. Second, an inter-procedural analysis on only the plugin code could handle the case where the relationship goes out of scope for similar reasons, such as calls to a helper function. These changes would increase the precision of all three variants of the analysis.

The two sources of imprecision affect all three variants, though in different ways. While imprecision when checking a constraint can produce a false positive in the sound variant or a false negative in the complete variant, the location of the imprecision in the constraint directly changes how the pragmatic variant handles it. When the imprecision occurs in the trigger predicate, the pragmatic variant results in a false negative. When the trigger predicate is precise but the requires predicate is imprecise, the pragmatic variant results in a false positive. This reflects what we expect from the analysis; we only wish to see an error if there is reason to believe that the constraint applies to our plugin. If the trigger predicate is unknown, it is less likely that the constraint is relevant.

## 6 Related Work

Typestates [9] are traditionally used for specifying protocols on a single object by using a state machine, but there are several approaches to inter-object typestate. Lam et al. manipulated the typestate of many objects together through their participation in data structures [10]. Nanda et al. take this a step further by allowing external objects to affect a particular object's state, but unlike relationships, it requires that the objects reference each other through a pre-defined path [11]. Bierhoff and Aldrich add permissions to typestates and allows objects to capture the permission of another object, thus binding the objects as needed for the protocol [12]. Relationships can combine multiple objects into a single state-like construct and are more general for this purpose than typestate; they can describe all of the examples used in multiple object typestate work.

With respect to the specifications, relationships are more incremental than typestate because the entire protocol does not need to be specified in order to specify a single constraint. Additionally, the plugin developer does not add any specifications, which she must do with some of the typestate approaches. However, because they require specifications on both sides, typestate analyses can soundly check that both the plugin and the framework meet the specification [9, 10, 12]. The relationship analysis assumes that the framework properly meets the specification and only analyzes the plugin.

Tracematches have also been used to enforce protocols [13]. Unlike typestate, which specifies the correct protocol, tracematches specify a temporal sequence of events which lead to an error state. This is done by defining a state machine for the protocol and then specifying the bad paths.

The tracematch specification approach is similar to that of relationships; the main difference is in how the techniques specify the path leading up to the error state. Tracematches must specify the entire good path leading up to the error state, which can lead to many specifications to define a single bad error state. In cases where multiple execution traces lead to the same error, such as the many ways to find an item in a `DropDownList` and select it incorrectly, a tracematch would have to specify each possibility. Instead of specifying the good path leading up to the error, relationships specify the context predicate, which is the same for all good paths. This difference affects how robust a specification is in the face of API changes. If the framework developer adds a new way to access `ListItems` in a `ListControl`, possibly through several methods calls, the existing tracematches will not cover that new sub-path. However, all the constraint specifications in the proposed technique will continue to work if the sub-path eventually results in the same relationships as other sub-paths.

Tracematches are enforced statically and dynamically using a global analysis [14]. The static analysis soundly determines possible violations, and it instruments the code to check them dynamically. Bodden et al. provide a static analysis which optimizes the dynamic analysis by verifying more errors statically [15], and Naeem and Lhoták specifically optimize with regard to tracematches that involve multiple objects [16]. While the FUSION analysis is static, it could be used in the same way by instrumenting all violations that are found by the sound variant but not by the complete variant.

Bierman and Wren formalized UML relationships as a first-class language construct [17]. The language extension they created gives relationships attributes and inheritance, and developers use the relationships by explicitly adding and removing them. Balzer et.

al. expanded on this work by describing invariants on relations using discrete mathematics and support semantic invariants and invariants between several relations [18]. In contrast to previous work, the relationships presented in this paper are added and removed implicitly through use of framework operations, and if inferred relationships are used, they may be entirely hidden from the developer.

Like the proposed framework language, Contracts [19] also view 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 do not check the conformance of plugins and the specifications are seemingly more complex to write.

The FUSION analysis is similar to a shape analysis, with the closest being TVLA [20]. TVLA allows developers to extend shape analysis using custom predicates that relate different objects. FUSION specifications could be written as custom TVLA predicates, but the lower level of abstraction would result in a more complex specification and would require greater expertise from the specifier.

## 7 Conclusion

Relationships capture the interaction between a plugin and framework by describing how abstract object associations change as the plugin makes calls to the framework. We can then use these relationships to describe constraints on framework operations. We have shown that FUSION's relationship-based constraints can describe many constraint paradigms found in real frameworks, capturing relationship structure, operation order, and object associations that may or may not derive from direct references. As the specifications are written entirely by framework developers, plugin developers only need to run the analysis on their code, so that investments by a few framework developers pay dividends to many plugin developers.

A currently intra-procedural static analysis can check that the plugin code meets framework constraints. This analysis is particularly interesting because it is adjustable. While many analyses strive to only be either sound or complete, the FUSION analysis can be run either soundly, completely, or as a pragmatic balance of the two, thereby allowing the plugin developer to choose the variant that provides the most useful results.

**Acknowledgements.** This work was supported in part by NSF grant CCF-0811592, 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, and a fellowship from Los Alamos National Laboratory. The authors would also like to thank Donna Malayeri and Kevin Bierhoff for their feedback.

## References

1. Johnson, R.E.: Frameworks = (components + patterns). Commun. ACM **40**(10) (1997)
2. Jaspan, C., Aldrich, J.: Checking semantic usage of frameworks. In: Proc. of the symposium on Library Centric Software Design. (2007)

3. Fowler, M.: Inversion of control containers and the dependency injection pattern. `http://www.martinfowler.com/articles/injection.html` (2004)
4. 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)
5. Jaspan, C., Aldrich, J.: Checking framework interactions with relationships: Extended. Technical Report CMU-ISR-140-08, Institute for Software Research, Carnegie Mellon University (December 2008)
6. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. **16**(6) (1994) 1811–1841
7. Parkinson, M.J., Bierman, G.M.: Separation logic, abstraction and inheritance. In: Proc. of the symposium on Principles of Programming Languages. (2008)
8. Dhara, K.K., Leavens, G.T.: Forcing behavioral subtyping through specification inheritance. In: Proc. of the International Conference on Software Engineering. (1996)
9. DeLine, R., Fähndrich, M.: Typestates for objects. In: Proc. of the European Conference on Object Oriented Programming. (2004)
10. Kuncak, V., Lam, P., Zee, K., Rinard, M.: Modular Pluggable Analyses for Data Structure Consistency. IEEE Trans. Softw. Eng. **32**(12) (2006)
11. Nanda, M.G., Grothoff, C., Chandra, S.: Deriving object typestates in the presence of inter-object references. In: Proc. of the Conference on Object Oriented Programming, Systems, Languages, and Applications. (2005)
12. Bierhoff, K., Aldrich, J.: Modular typestate checking of aliased objects. In: Proc. of the Conference on Object Oriented Programming, Systems, Languages, and Applications. (2007)
13. Walker, R.J., Viggers, K.: Implementing Protocols via Declarative Event Patterns. In: Proc. of the symposium on Foundations of Software Engineering. (2004)
14. Bodden, E., Hendren, L., Lhoták, O.: A staged static program analysis to improve the performance of runtime monitoring. In: Proc. of the European Conference on Object Oriented Programming. (2007)
15. Bodden, E., Lam, P., Hendren, L.: Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In: Proc. of the symposium on Foundations of Software Engineering. (2008)
16. Naeem, N.A., Lhoták, O.: Typestate-like analysis of multiple interacting objects. In: Proc. of the Conference on Object Oriented Programming, Systems, Languages, and Applications. (2008)
17. Bierman, G., Wren, A.: First-class relationships in an object-oriented language. In: Proc. of the European Conference on Object Oriented Programming. (2005)
18. Balzer, S., Gross, T., Eugster, P.: A relational model of object collaborations and its use in reasoning about relationships. In: Proc. of the European Conference on Object Oriented Programming. (2007)
19. Helm, R., Holland, I.M., Gangopadhyay, D.: Contracts: specifying behavioral compositions in object-oriented systems. In: Proc. of the Conference on Object Oriented Programming, Systems, Languages, and Applications. (1990)
20. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. **24**(3) (2002) 217–298