# Checking Framework Plugins

Ciera Jaspan
Carnegie Mellon University
cchristo @ cs.cmu.edu

Jonathan Aldrich
Carnegie Mellon University
jonathan.aldrich @ cs.cmu.edu

## Abstract

Software frameworks contain many internal constraints which plugins cannot break. These constraints are relative to the context of the plugin, and they can involve multiple framework objects. This work creates a lightweight mechanism specifying semantic constraints on the framework and checking plugins against this specification.

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

***General Terms*** Design, Documentation, Verification

***Keywords*** framework constraints

## 1. Software Frameworks

Software frameworks impose many constraints on the *plugins* that use them. While these constraints might be documented, it is difficult to find all the relevant documentation, especially if the developer does not know to look for it. For this reason, even experiences developer have difficulty keeping track of all the constraints that they must comply with.

Framework constraints are characterized by four properties:

- *Multiple classes and objects* Unlike library 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.
- *Semantic nature* Framework constraints are not all about method naming conventions or which methods to override. The developer must be aware of the temporal ordering of operations and the relationships that the framework object have with each other.
- *Non-local enforcement* Frequently, framework constraints are not checked until much after the plugin developer broke the constraint. In some cases, they are never checked, and unusual runtime behavior may occur.
- *Independence of other constraints* Many different constraints may operate on the same objects an operations. They must be allowed to share the same information about how the objects relate, 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.

This poster proposes a lightweight mechanism for specifying framework constraints and checking plugins for defects produced by these constraints. Using this mechanism, the plugin developers will not have any up-front cost, they will only need to run the tool. The framework developers will be able to specify the framework constraints in a modular manner, so that each constraint is specified and enforced separately from the other constraints. The constraint specifications will take both the syntactic and semantic nature of framework constraints into account.

## 2. Relationships and Scopes

We propose two constructs, *relationships* and *constraints* to express framework constraints and discover inconsistencies in plugin code. *Constraints* allows us to define paths of code through the plugin. Along these paths, new operations may be available, or old operations may be disabled. Constraints describe these paths and operations syntactically, but they must also rely on the state of the framework. *Relationships* provide a way to express and track state of framework objects and provide a semantic context for the constraints.

*Relationships* associate two objects with a user-defined meaning. Relationships are specified on the framework's methods to declare what knowledge we have after calling the method. The attribute `[Add("Child", o1, o2)]` creates a relationship between `o1` and `o2`, while `[Remove("Child", o1, o2)]` removes this relationship from the current context. After calling a method, we acquire (or remove) a set of relationships. We use a dataflow analysis to track relationships through the plugin and associate every expression in the plugin with a set of relationships available at that point.

*OOPSLA'07,* October 21–25, 2007, Montréal, Québec, Canada.
ACM 978-1-59593-865-7/07/0010.

*Constraints* are modeled as a path through the plugin code with a specified `start` and `end` point. A constraint path may start and end at any expression or language construct. We call these places where paths can start and end *program points*. We will use relationships to define the semantic aspect of the constraint by associating each program point with a set of relationships. A constraint path also defines a set of objects that it operates over. An actual path in the plugin code is known as a *constraint instance*. A constraint instance is unique when the constraint's variables are bound to a unique set of plugin variables.

These relationships must exist at the program point in order for the program point to apply. For example, if a path starts at `oldSel.setSelected(false)` when we have the relationships `Child(oldSel, ctrl)` and `Selected(oldSel)`, then the constraint will only start if the plugin has the declared relationships at that expression. When we start a constraint path, we will create a new instance of the constraint by binding the plugin objects to the variables declared by the constraint. Eventually, we end the constraint instance through a similar mechanism.

We also use program points and relationships to restrict operations and objects on the path. If an operation is `enabled` by a constraint, it can *only* be used *on* the path. We may also `forbid` operations on the path, or we may require that some object be `scoped` so that it is not used off the path. Like the `start` and `end` points of a path, the program points for `enabled`, `forbid` and `scoped` are all associated with a set of relationships that provide a semantic requirement.

We propose an analysis that will check `enabled`, `forbid` and `scoped` program points. At each program point in the plugin, the analysis will check whether the program point and relationships are controlled by a constraint. If so, then the analysis will check it accordingly. For example, an `enabled` program point must exists on a path, and the variable bindings of the constraint instance must match the program point and relationships. If the analysis can not find a constraint instance with the correct variable bindings, then the analysis generates an error.

## 3. Related Work

SCL [3] allows framework developers to create a specification for the structural constraints of the framework. The proposed work focuses on semantic constraints rather than structural constraints. Some of the key ideas from SCL could be used to drive the structural parts of the specifications.

Object typestates [1] 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 to 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 object typestates. Some typestate work has explored inter-object typestate cite-Nanda05, Lam05Typestate, but this work still considers each object to have an individual typestate.

Like the proposed work, contracts [2] also view the relationships between objects as a key factor in specifying systems. Contracts allow all the power of first-order predicate logic and can express very complex invariants. Contracts differ from the proposed work because they do not make the tie directly back to the plugin code and have a higher complexity for the writer of the contract.

## 4. Conclusion

Frameworks place constraints on plugins that are relative to a semantic context of the code. We have proposed a lightweight mechanism to specify framework constraints. A plugin developer does not need to add any specifications to their code, and framework developers can add specifications in a modular way.

We have also proposed an analysis which reads the constraint specifications and the plugin code to detect misuses of the framework. The analysis checks that a plugin only uses operations and objects that are allowed in the current state of the framework. When an incorrect operation is used, the analysis will produce a local error at the expression in the plugin that was not allowed under the current context.

In future work, we will implement this analysis and apply it to several examples we have found from frameworks such as ASP.NET, Eclipse, and EJB. We will also explore ways to make the constraint specifications more concise. Finally, we will attempt to provide additional error information to the plugin developer, including suggestions for how to fix the error based upon the plugin's context and the broken constraint.

## References

[1] R. DeLine and M. Fahndrich. Typestates for objects. In *ECOOP*, 2004.

[2] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. In *OOPSLA*, 1990.

[3] D. Hou and H. J. Hoover. Using SCL to specify and check design intent in source code. *IEEE TSE*, 32(6), 2006.

[4] P. Lam, V. Kuncak, and M. Rinard. Generalized typestate checking for data structure consistency. In *Verification, Model Checking, and Abstract Interpretation*, 2005.

[5] M. G. Nanda, C. Grothoff, and S. Chandra. Deriving object typestates in the presence of inter-object references. In *OOPSLA*, 2005.