# Verifying Configuration Files

Ciera Jaspan
Carnegie Mellon University
ciera@cmu.edu

## Abstract

Configuration files are commonly used to create applications by gluing together new or existing components. However, components may be missing, misconfigured, or connected improperly, resulting in exceptions and unusual runtime behavior. This research contributes a mechanism to statically verify configuration files alongside the component code.

***Categories and Subject Descriptors*** D.2.4 [*Software Engineering*]: Software/Program Verification

***General Terms*** Verification

***Keywords*** configuration files, relationships

## 1. Configuration Files

Configuration files are used by many software frameworks and platforms to describe how to connect components together in new ways to create an application. Apache Tomcat, Eclipse, Spring, and Ruby on Rails, all require developers on their platform to use configuration files to describe the component they are developing and how it connects with other components. While XML [3] is frequently used for these configuration files (and is by all the systems described above), systems may also define configuration files in a proprietary format.

When XML is used, the system will frequently provide a schema which describes the appropriate layout of the configuration file. However, these schemas can only be used to check the layout of the configuration file; they cannot check for more detailed semantic problems that might cause an exception or unexpected runtime behavior.

As an example, consider how a developer uses the Spring Web Application framework to create a simple web application [2]. In this framework, a developer must provide several components:

**Listing 1.** Definition of the Controller

```
1 package edu.cmu.cs.classquiz.web;
2 import org.springframework.web.servlet.mvc.*;
3
4 public class AddQuestionController
5  extends SimpleFormController {
6    protected ModelAndView onSubmit(Object command)
7     throws Exception {...}
8 }
```

**Listing 2.** Definition of the Validator

```
1 package edu.cmu.cs.classquiz.bus;
2 import org.springframework.validation.*;
3
4 public class QuestionValidator implements Validator {
5    public boolean supports(Class clazz) {
6       return clazz.equals(QuestionCommand.class);
7    }
8    public void validate(Object target, Errors errors) {...}
9 }
```

- A *controller* that handles submissions to a web form, as shown in Listing 1.
- A *command object* that represents the data being submitted to the web form.
- A *validator* that checks that the data submitted is valid, as shown in Listing 2.

These three components are connected through an XML file, such as the one shown in Listing 3.

While this configuration file appears simple, there are several constraints hidden within it:

- Since the class `AddQuestionController` derives from `SimpleFormController`, it must describe the properties shown in Listing 3.
- The type of the element referenced at line 6 in Listing 3 must be derived from `Validator`.
- The `Validator` must be able to validate the type defined at line 7, by way of returning true when its `supports` method is called.

Of course, all the classes described must exist, and there are more constraints about the other properties in lines 8-11. Unfortunately, the DTD schema declared in line 2 can only be used to check layout and not any interconnections between the XML elements or between XML elements and other code. Breaking any of the above constraints causes a


*SPLASH'10,* October 17–21, 2010, Reno/Tahoe, Nevada, USA.
ACM 978-1-4503-0240-1/10/10.

**Listing 3.** Snippet of a Spring config file

```
<xml version=''1.0'' encoding=''UTF−8''?>
<!DOCTYPE beans PUBLIC ''−//SPRING//DTD BEAN//EN'' ''http://www.springframework.org/dtd/spring−beans.dtd''>
<beans>
   <bean id=''addQuestionValidator'' class=''edu.cmu.cs.classquiz.bus.QuestionValidator''/>
   <bean id=''addQuestionForm'' class=''edu.cmu.cs.classquiz.web.AddQuestionController''>
      <property name=validator><ref bean=''addQuestionValidator''/></property>
      <property name=''commandClass''><value>edu.cmu.cs.classquiz.bus.QuestionCommand</value></property>
      <property name=''commandName''><value>newQ</value></property>
      <property name=''sessionForm''><value>true</value></property>
      <property name=''formView''><value>add_question</value></property>
      <property name=''successView''><value>view_questions.html</value></property>
   </bean>
</beans>
```

runtime error either when the application is loaded, or worse, when the controller is invoked by a user submitting a form.

## 2. Collaboration Constraints and Relationships

The constraints described above are an example of a *collaboration constraint*, a state-based restriction on how multiple objects may interact. Prior work on collaboration constraints has used the FUSION tool to statically analyze code for violations of collaboration constraints [1]. In the FUSION system, framework developers specify collaboration constraints using *relationships*. A relationship is a user-defined predicate over objects that describes how they are related. For example, a relationship called Validator could connect together a `Validator` and a `SimpleFormController`. These relationships can then be used in logical predicates to describe pre- and post-conditions of operations.

This same specification system can be used to describe the collaboration constraints in configuration files. For example, there are three relationships between the validator, the controller, and the command: Validator(Validator, SimpleFormController), Validates(Validator, Class), and Submits(SimpleFormController, Class). Using these three relationships, the last constraint above can be specified as:

ctrlr instanceof SimpleFormController ⇒
(Validator(valid, ctrlr) ∧ Validates(valid, cls) ∧
Submits(ctrlr, cls))

That is, whenever, there is an object of type `SimpleFormController`, it must have a validator. Additionally, that validator must validate commands of the same type as the controller is submitting.

## 3. Querying Relationships

In [1], relationships are derived from the post-conditions of operations. For example, the Validates relationship would be derived as a post-condition of the `Validator.supports` method. However, the relationships Validator and Submits are *only* described in the configuration file.

To retrieve relationships from XML-based configuration files, the FUSION system allows framework developers to write XQuery that describes the relationships. Listing 4 shows sample XQuery code that retrieves the Validator relationship from any Spring configuration file. To verify this program, FUSION simply runs the XQuery to retrieve all the relationships from the configuration files and then runs the static analysis on the rest of the program as normal.

**Listing 4.** XQuery to retrieve the Validator relationship

```
let $beans := doc($doc)/beans/.
for $bean in $beans/bean*
let $validator in $bean/property[name="validator"]/ref
where isSubtype(type($bean),
 "org.springframework.web.servlet.mvc.SimpleFormController")
return <Relationship name="Validator">
        <Object name ="{name($validator)}"
         type="{type($validator)}"/>
        <Object name="{name($bean)}" type="{type($bean)}"/>
      </Relationship>
```

## 4. Future Work

While this solution works, it is far from ideal. The XQuery in Listing 4 depends on several developer-defined functions, like `type` and `name`, in order to retrieve semantic information that is not built into XML. In particular, XML is missing concepts such as subtyping and object identity. As each framework may have its own XML semantics for these concepts, there is no general way to specify them in XML.

This is perhaps not a fault of the solution to use XQuery to retrieve relationships, or even of XML, but of using XML for configuration files. XML is a data format language, but configuration files do not describe a data format but a program. Therefore, configuration files should be written in a programming language, complete with a semantics that is interoperable with the semantics of the component programming language. Understanding the constraints and relationships in existing XML files is a first step towards the larger goal of creating a true configuration language.

## References


[1] C. Jaspan and J. Aldrich. Checking framework interactions with relationships. In *ECOOP*, 2009.

[2] SpringSource. Spring Web Application Framework. `http://www.springsource.org/`.

[3] World Wide Web Consortium. Extensible Markup Language. `http://www.w3.org/XML/`.