Executable Documentation of Template-Hook Interactions in Frameworks using JML

Neeraj Khanolkar and Gary T. Leavens

TR #06-18
June 2006

Keywords: Formal specification, object-oriented framework, runtime assertion checking, template, hook, temporal process documentation, preconditions, postconditions, flow-based assertions, JML.

2006 CR Categories: D.1.5 [*Programming Techniques*] Object-oriented programming; D.2.1 [*Software Engineering*] Requirements/Specifications — languages, JML; D.2.4 [*Software Engineering*] Software/Program Verification — assertion checkers, programming by contract; F.3.1 [*Theory of Computing*] Logics and Meanings of Programs — assertions, pre- and post-conditions, specification techniques.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

Executable Documentation of Template-Hook Interactions in Frameworks using JML

Neeraj Khanolkar and Gary T. Leavens

Department of Computer Science
Iowa State University
{neerajsk, leavens}@cs.iastate.edu

Abstract. Object-oriented frameworks are an important technique for capturing design expertise. However, the learning curve for a framework is usually quite steep and can be the biggest obstacle in its adoption. We propose an executable and yet readable method for framework documentation using the Java Modelling Language (JML), based on the specification of the interaction between a framework's template methods and its customizable hooks. This method is geared toward allowing the developers to quickly instantiate a prototype application from the framework, which can be later tweaked using some other detailed and usually non-executable documentation. We use flow-based assertions to specify the hook method preconditions and template method postconditions. The flow-based precondition for a particular hook serves as a modular documentation of when and how that hook is called in the framework's overall call-sequence. Similarly, the flow-based postcondition of a template method tells the possible sequences of hook invocations that its execution may cause. Flow-based assertions are written using a few types, which we precisely specify. We also briefly describe a case study that uses our technique to document a Model-View-Controller framework.

1 Introduction

1.1 Background

Object-oriented *frameworks* are skeletons for applications. They can be instantiated into a specific application by customization. The users of a framework, i.e., the *developers*, customize the framework by supplying it with concrete subclasses that override callback methods to form the application-specific custom code. These overridden methods—also known as *hook methods*—are invoked by the framework in response to events (such as user input or model state changes). The control logic of the framework resides in its *template methods*, which invoke hook methods according to a predefined control sequence. These template methods are usually not overridden by the developers.

A framework's documentation plays the crucial role of a how-to manual for the developers; developers must learn how to customize the framework into an application by using the framework's documentation and code. We will focus

on formalizing documentation for frameworks using specifications written in the Java Modelling Language (JML) [1], which is a behavioral interface specification language for Java.

1.2 Motivation and Objective

In order to understand the problem, it will help to see it from a developer's point of view. The developer adds extra state to a framework, in the form of fields declared in subtypes of the framework's types. Thus the state of an instantiated application consists of two parts: the *framework-specific state*, which is contained in the superclasses that the framework declares, and the *application-specific state*, which is declared in the application's customizing subclasses.

A simplified view of the process of designing applications from a framework, is thus as follows:

State Design The developer decides what information must be kept in the application-specific state.

Customization Using the framework's documentation, the developer assigns responsibilities to each hook method, so that the application-specific state is manipulated according to the application's requirements.

Specification This detailed design for each hook can then be specified using additional pre- and postconditions that describe its effects on the application-specific state.

This paper provides a technique for framework documentation that eases customization. Customization is important because each method has to be customized before its details can be specified. The perceived ease of customization is critical to the framework's adoption.

An example of the process of customization for a hook is as follows. Consider the hook method `createView()` in the interface `ViewCreator` given in Fig. 1.

```
public interface ViewCreator {
  /* ... */
  public View createView();
}
```

Fig. 1. The ViewCreator interface

To understand the role of this hook in managing the application-specific state, and thus correctly override it, the developer must at least have information that answers the following questions:

1. What are the possible application-specific states in which this hook is invoked?

2. What are the expectations of the framework, with respect to the framework-specific state, for this hook's invocation?

The second question can be mainly handled by specifying a frame axiom [2] (assignable clause in JML) and a postcondition for the hook method in an interface such as Fig. 1.

Thus, we will focus on the first question in this paper.

Some hook methods do validation of the application-specific state, while others transform it.[1] Thus, to effectively override a hook method, the developer must comprehend the possible application-specific states in which it may be called. To this end, the developer must know what possible sequences of hook invocations may have preceded this invocation.

The other—less critical—information that a developer must know is what other hooks may be collaborating or depending on this hook and can possibly be affected by its customization. These collaborative hooks are usually grouped together in one template method. Thus knowing about which possible template methods can invoke a hook, aids the developer in figuring out the hook's collaborators.

There is a lot more information that can help the developer, but as we point out in the Sec. 6, we want to keep the specifications readable and executable. Following the theory of minimalist instruction [3], we assume that the developer will use her intuition and/or consult more detailed documentation to fill in the missing pieces. So we will not include any more information in our flow-based formal specifications.

To sum up, the problem is to provide information that can answer the following questions for the developer:

1. What are the various hook invocation traces that may precede a given hook's invocation?
2. What are the different template methods that may directly or indirectly invoke a particular hook? In other words, which template methods are *active* in the sense that they may have been entered but not exited when this hook is invoked?

Using the answers to the above questions, the developer can reason about the application-specific state at a hook's invocation point and also—if needed—reason about what other hooks may be affected by that hook.

One way to answer the above questions for a given hook is to list all possible sequences of active templates and hook calls that can precede the invocation of that hook. As can be seen in Fig. 1, the signature of the hook method does not carry enough information to construct such sequences. At most, it tells us about the static types of the arguments and return values. The developer will have to analyze the source code of all the template methods in the framework to decide what hooks invocations may precede the hook in question. This source

[1] It is also possible for a hook method to both validate and transform the application-specific state.

code can be complex and will often contain details that make it not especially readable, at least in terms of answering such a specific question. Therefore, some other method would be helpful for capturing this information in a more readable format. It would also help if this information was modular i.e. if for each hook, the information about the possible preceding hook invocations and the active templates, could be located in one module, preferably where the hook is declared by the framework.

One readable and modular method of capturing this kind of information is to use a visual artifact such as a flow graph, which represents the call sequences graphically. Figure 2 shows a flow graph that depicts an example callback context for the above hook method `createView()`. From the flow graph it is clear that there are two different paths of call sequences that may precede the invocation of this hook. It also shows that the invocation of `ViewCreator`'s `createView()` method can happen within the temporal extent of two different template methods in `InputDataBroker`: `processForm()` and `getFirstForm()`.

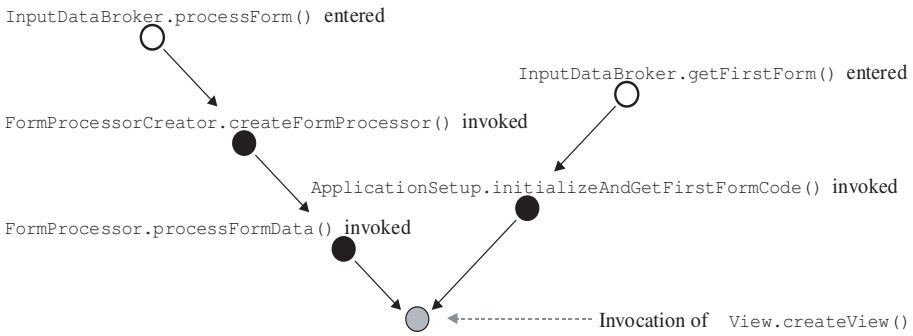

Fig. 2. Flow graph showing the possible sequences that can reach the invocation of `ViewCreator.createView()`. The hollow circles denote the activations of the template methods, and the solid circles represent hook invocations.

While flow graphs are quite valuable, as they intuitively convey the context information, they must be converted into some textual or other easily manipulated representation to be used for automatic checking. For our initial explorations of framework specifications, we use a textual representation directly. More sophisticated tools could use a graphical format directly, and convert it internally into an equivalent representation. Our "low tech" solution has the advantage of being directly usable within textual source code and in other existing design by contract style specification languages.

Our goal is thus to provide an easy-to-use technique to encode flow graphs as succinct, readable and machine-checked control flow assertions that can be used

to document hooks and template methods. Machine-checked assertions can be used to guarantee that the specifications are always up to date.

Our technique uses JML's runtime assertion checker to provide machine-checked, executable specifications for frameworks. Runtime assertion checking in this style allows documentation of how the framework-specific state is manipulated, in addition to the flow-based assertions that are the focus of this paper. With both kinds of assertions, bugs in the instantiated application become easier to trace as either due to faulty framework code or due to developer's application-specific subclasses.

As a preview of our technique, consider the JML precondition specification given in Fig. 3. Although we have not yet presented any details of our specification types, we believe that the formal specification is readable enough to be apparent that Fig. 3 is the textual representation of the visual graph in Fig. 2.

```
public interface ViewCreator {
  /* ... */
  /*@ requires
    @    new Flow().
    @      withinTemplate(InputDataBroker.processForm).
    @        hookInvoked(FormProcessorCreator.createFormProcessor).
    @        hookInvoked(FormProcessor.processFormData).
    @        here()
    @ || new Flow().
    @      withinTemplate(InputDataBroker.getFirstForm).
    @        hookInvoked(ApplicationSetup.initializeAndGetFirstFormCode).
    @        here();
    @*/
  public View createView();
}
```

Fig. 3. JML precondition specification for `ViewCreator.createView()`

1.3 Outline

The rest of this paper is structured as follows. Section 2 describes the various flow based constructs that can be used to create the control flow assertions. Section 3 provides a formal, high-level semantic description of the various classes involved in the specifications. Section 4 describes the instrumentation process for capturing the execution trace. Section 5 summarizes our experiences in testing our technique on a Model-View-Controller framework. We discuss the underlying theme of our approach in Sec. 6, and cover related work in Sec. 7. We conclude the paper with future directions and conclusions.

2 Flow Specification Constructs

Here, we illustrate with examples the various flow constructs that can be used to build the control flow assertions. For an exact semantic description of the various specification types and operations, please see Sec. 3.

2.1 Simple Composition

The simplest flow construct is one where the specifier adds on atomic events to create a linear, one-path flow graph. An example of this is given in Fig. 4. As shown in the listing, the precondition of the hook `Form.reBuildContent()` simply mandates that before this hook is invoked, the template method `InputDataBroker.processForm()` should be entered—but not exited, followed by invocations of the hooks `FormProcessorCreator.createFormProcessor()`, `FormProcessor.processFormData()`, and `ViewCreator.createView()` in that order.

```
/*@
  @ requires
  @  new Flow().
  @    withinTemplate(InputDataBroker.processForm).
  @       hookInvoked(FormProcessorCreator.createFormProcessor).
  @       hookInvoked(FormProcessor.processFormData).
  @       hookInvoked(ViewCreator.createView).
  @       here();
  @*/
public abstract void reBuildContent(...);
```

Fig. 4. A linear flow precondition for `Form.reBuildContent()`

This linear composition construct forms the base case for the higher level construction of flows using the concatenate, looping and choice operators—each of which will explained in turn, next.

2.2 Concatenation

Using the `Flow.concatenate()` method, we can attach new flows to existing flows. When the same flow i.e the same method call sequence repeats several times in a specification, or when a flow needs to be constructed dynamically, then the concatenate operation in conjunction with JML model methods can be used to make the specification concise and readable. We will see examples of its use in the looping and deterministic choice constructs, shortly.

2.3 Looping

The operations for attaching atomic events and for concatenation, can be parameterized with an integer that indicates how many times that operation should repeat. This is useful in specifying a loop in the call sequence. For example, consider the specification snippet given in Fig. 5. Here, an inner flow sequence is attached consecutively to the outer flow using the concatenate operation to create a new flow. This new flow has a single path which starts with an invocation of `ApplicationSetup.getFormCodeList()`, followed by invocations of `ApplicationSetup.getViewCreator()` and `ApplicationSetup.getFormProcessorCreator()`—in that order—repeated `numberOfFormCodes` times, and ending with an invocation of `ApplicationSetup.setupCompleted()`.

```
/*@   ...
  @    new Flow().
  @      hookInvoked(ApplicationSetup.getFormCodeList).
  @      concatenate(
  @         new Flow().
  @           hookInvoked(ApplicationSetup.getViewCreator).
  @           hookInvoked(ApplicationSetup.getFormProcessorCreator),
  @         numberOfFormCodes
  @      ).
  @      hookInvoked(ApplicationSetup.setupCompleted);
  @   ...
  @*/
```

Fig. 5. Specifying a loop using the concatenate operation

2.4 Choice

The choice construct is used to specify branching within a flow. This construct can be used in two different ways depending on whether the branching can be specified in a deterministic manner or not.

Non-Deterministic Choice. It is possible that the specifier does not have enough information to indicate the conditions for each branch in a flow. It may also be that doing this may expose internal details, or is just plain inconvenient. In such cases, the branching can specified in a non-deterministically by using the non-deterministic choice constructs. We have already seen an example of non-deterministic choice in Fig. 3. There, we used JML's logical OR operator '`||`', to specify that the paths of either of the two flows can be a possible trace prior to the invocation of `ViewCreator.createView()`.

The '||' operator works fine when the multiple paths are completely disjoint. However, if the different paths have a common prefix or a common suffix or both, then the '||' operator forces these common parts to be repeated for each path specification. To avoid this repetition, we can use the `Flow.choice()` method to indicate non-deterministic branching within a flow. Figure 6 shows an example of using this operator. The listing shows two flows merging into a common suffix flow—`ViewCreator.createView()` followed by `ApplicationSetup.getForm()`. The initial split is specified non-deterministically with the choice operator.

```
/*@
  @ requires
  @  new Flow().
  @    choice(
  @        new Flow().
  @          withinTemplate(InputDataBroker.processForm).
  @            hookInvoked(...).
  @            hookInvoked(...),
  @
  @        new Flow().
  @          withinTemplate(InputDataBroker.getFirstForm).
  @            hookInvoked(...)
  @    ).
  @    hookInvoked(ViewCreator.createView).
  @    hookInvoked(ApplicationSetup.getForm).
  @    here();
  @*/
public abstract void buildContent(...);
```

Fig. 6. Non-deterministic choice with a common suffix

Deterministic Choice. If it is possible to specify the conditions under which a certain branch of a flow will be taken, then JML model methods and the concatenate operator can be used together to specify the deterministic choice. This is illustrated in Fig. 7. Here, the model method `processingBranch()` constructs and returns a flow depending on which of the three branches of its `if` conditional is satisfied. This dynamically returned flow is then concatenated with a statically specified flow to construct the complete postcondition flow specification. Although the final flow specification can only be determined at runtime, the JML model method `processingBranch()` does provide a static specification of the conditions under which the three different branches may be taken.

```
/*@
  @ public model pure Flow processingBranch(){
  @   if(this.inputDataHasErrors){
  @     return new Flow();
  @   }
  @   else{
  @     if(this.dataProcessingHasErrors){
  @       return
  @         new Flow().
  @          hookInvoked(...);
  @     }
  @     else{
  @       return
  @         new Flow().
  @          hookInvoked(...).
  @          hookInvoked(...).
  @          hookInvoked(...);
  @     }
  @   }
  @ }
  @*/

/*@
  @  ensures
  @   new Flow().
  @    hookInvoked(...).
  @    concatenate(this.processingBranch()). //<- deterministic choice
  @    hookInvoked(...).
  @    since(processFormData.lastEntry());
  @*/
protected final ProcessingResult processFormData(...);
```

Fig. 7. Deterministic choice using a JML model method and flow concatenation

3 Formal Semantics

3.1 Preliminaries

Here, we define some functions and relations on sequences, which will be used later in the semantic descriptions of the specification types.
We define a 'sequence' as an ordering of some elements (of the same type). Our sequences will begin with index 1 from the left. The empty sequence is denoted by the epsilon character i.e ϵ. This character will always be the rightmost element in a non-empty sequence and denotes the end of the sequence.
We can refer to the i^{th} element of the sequence by using the index i as a subscript of the sequence name. The $|\langle$ sequence name $\rangle|$ notation will be used to denote the length of the sequence. We can extract part of the sequence seq by using the notation $seq_{i...j}$ which denotes the sequence
$seq_i : seq_{i+1} : seq_{i+2} : \ldots : seq_j : \epsilon$.
For example, if $seq = x_1 : x_2 : x_3 : \ldots : x_n : \epsilon$, then $|seq| = n$ is the length of the sequence and $seq_2 = x_2$ is the 2^{nd} element of the sequence. To denote a suffix of seq starting from the third element, one would write $seq_{3...|seq|}$.
Next, we define some (total and partial) functions for sequences.
Appends an element to the end of a sequence.

$$appendElt(\epsilon, x) = x : \epsilon$$

$$appendElt(y : Ys, x) = y : appendElt(Ys, x)$$

Concatenates two sequences.

$$appendSeq(\epsilon, X) = X$$

$$appendSeq(x : Xs, Ys) = x : appendSeq(Xs, Ys)$$

Returns the index of the last occurrence of an element in a sequence.

$$lastOccurrenceOf(x, seq) = i, where\ (seq_i = x) \wedge (\forall j : |seq| \geq j > i : seq_j \neq x)$$

Deletes from a sequence, the element at a given index.

$$deleteEltAt(x : Xs, i) = \begin{cases} x : deleteEltAt(Xs, i-1) & \text{if i} > 1 \\ Xs & \text{if i} = 1 \end{cases}$$

Next, we define the relation in to check if an element occurs in a sequence.

$$x\ in\ (y : Ys) \iff \begin{pmatrix} (x = y) \\ \vee \\ (x \neq y \wedge x\ in\ Ys) \end{pmatrix}$$

Lastly, we define the binary relation *subsequenceOf*.

$$\epsilon\ subsequenceOf\ \ X$$

$$(x : Xs)\ subsequenceOf\ (y : Ys) \iff \begin{pmatrix} (x = y \wedge Xs\ subsequenceOf\ \ Ys) \\ \vee \\ (x \neq y \wedge (x : Xs)\ subsequenceOf\ \ Ys) \end{pmatrix}$$

It is straightforward to prove that *subsequenceOf* is transitive.

3.2 Semantics of Specification Types

Following are abstract semantic descriptions of the temporal specification types that can be used to record the actual execution trace, and construct temporal assertions based on this trace.
In order to maintain consistency with the functional notation, the instance methods will be specified as though they are functions, by explicitly listing the implicit `this` reference as the first parameter. For abstraction purposes, we consider the execution trace as a static global variable of type $(\texttt{EventID})^*$. It appears as the second parameter called *trace*, in several methods.
`EventID` class
This class is the supertype of event identifiers. `EventID` has two subclasses: `HookMethod` and `TemplateMethod`.
`HookMethod` class
$\texttt{recordInvocation} : \texttt{HookMethod} \times (\texttt{EventID})^* \rightarrow (\texttt{EventID})^*$
We specify this method by describing the effect it has on the execution trace.

$$\texttt{recordInvocation(h,}\mathit{trace}\texttt{)} = \mathit{appendElt}(\mathit{trace}, \texttt{h})$$

Informally, it appends the supplied `HookMethod` to the end of the execution trace.
`TemplateMethod` class
$\texttt{recordEntry} : \texttt{TemplateMethod} \times (\texttt{EventID})^* \rightarrow (\texttt{EventID})^*$
$\texttt{recordExit} : \texttt{TemplateMethod} \times (\texttt{EventID})^* \rightarrow (\texttt{EventID})^*$
$\texttt{lastEntry} : \texttt{TemplateMethod} \times (\texttt{EventID})^* \rightarrow \mathbb{N}$
We specify the first two methods, again by describing the effect each has on the execution trace.

$$\texttt{recordEntry(tm,}\mathit{trace}\texttt{)} = \mathit{appendElt}(\mathit{trace}, \texttt{tm})$$

$$\texttt{recordExit(tm,}\mathit{trace}\texttt{)} = \mathit{deleteEltAt}(\mathit{trace}, \mathit{lastOccurrenceOf}(\texttt{tm}, \mathit{trace}))$$

The first method just appends the supplied `TemplateMethod` to the end of the execution trace, while the second one deletes the last entry of the supplied `TemplateMethod` from the execution trace.
Lastly, the third method—`lastEntry`—can be abstractly specified as:

$$\texttt{lastEntry(tm,}\mathit{trace}\texttt{)} = i,\ \mathit{where}\ i = \mathit{lastOccurrrenceOf}(\texttt{tm}, \mathit{trace})$$

`Flow` class
$\texttt{new Flow} : \texttt{Flow}$
$\texttt{withinTemplate} : \texttt{Flow} \times \texttt{TemplateMethod} \rightarrow \texttt{Flow}$
$\texttt{withinTemplate} : \texttt{Flow} \times \texttt{TemplateMethod} \times \mathbb{N} \rightarrow \texttt{Flow}$
$\texttt{hookInvoked} : \texttt{Flow} \times \texttt{HookMethod} \rightarrow \texttt{Flow}$
$\texttt{hookInvoked} : \texttt{Flow} \times \texttt{HookMethod} \times \mathbb{N} \rightarrow \texttt{Flow}$
$\texttt{concatenate} : \texttt{Flow} \times \texttt{Flow} \rightarrow \texttt{Flow}$
$\texttt{concatenate} : \texttt{Flow} \times \texttt{Flow} \times \mathbb{N} \rightarrow \texttt{Flow}$
$\texttt{choice} : \texttt{Flow} \times (\texttt{Flow})^* \rightarrow \texttt{Flow}$
$\texttt{here} : \texttt{Flow} \times (\texttt{EventID})^* \rightarrow \{true, false\}$
$\texttt{since} : \texttt{Flow} \times (\texttt{EventID})^* \times \mathbb{N} \rightarrow \{true, false\}$

Of the above ten methods, the first eight methods are used to construct a `Flow` object, while the last two are query methods.
We specify the first eight methods using the abstraction function, *paths*.

$$paths : \texttt{Flow} \rightarrow \mathcal{P}((\texttt{EventID})^*)$$

$$\texttt{new Flow()} = \texttt{f},\ \textit{where}\ paths(\texttt{f}) = \{\epsilon\}$$

$$paths(\texttt{withinTemplate(f,t)}) = \{appendElt(tr, \texttt{t}) \mid tr \in paths(\texttt{f})\}$$

$$paths(\texttt{withinTemplate(f,t,}\boldsymbol{n}\texttt{)}) = \begin{cases} paths(\texttt{f}) & n = 0 \\ paths(\texttt{withinTemplate(withinTemplate(f,t,}\boldsymbol{n}\texttt{-1),t)}) & \text{otherwise} \end{cases}$$

$$paths(\texttt{hookInvoked(f,h)}) = \{appendElt(tr, \texttt{h}) \mid tr \in paths(\texttt{f})\}$$

$$paths(\texttt{hookInvoked(f,h,}\boldsymbol{n}\texttt{)}) = \begin{cases} paths(\texttt{f}) & n = 0 \\ paths(\texttt{hookInvoked(hookInvoked(f,h,}\boldsymbol{n}\texttt{-1),h)}) & \text{otherwise} \end{cases}$$

$$paths(\texttt{concatenate(f,f')}) = \{appendSeq(tr, tr') \mid tr \in paths(\texttt{f}),\ tr' \in paths(\texttt{f'})\}$$

$$paths(\texttt{concatenate(f,f',}\boldsymbol{n}\texttt{)}) = \begin{cases} paths(\texttt{f}) & n = 0 \\ paths(\texttt{concatenate(concatenate(f,f',}\boldsymbol{n}\texttt{-1),f')}) & \text{otherwise} \end{cases}$$

$$paths(\texttt{choice(f,}\boldsymbol{a}\texttt{)}) = \{appendSeq(tr, tr') \mid tr \in paths(\texttt{f}), (\forall i : 0 \leq i < |a| :\ tr' \in paths(a_i))\}$$

Next, we define the relation *satisfies* to formally specify when a `Flow` is 'satisfied' by an execution trace.

$$satisfies : (\texttt{EventID})^* \times \texttt{Flow}$$

$$trace\ satisfies\ \texttt{f} \iff (\exists p : p \in paths(\texttt{f})\ \wedge\ p\ subsequenceOf\ trace)$$

Informally, the above definition means that we consider a `Flow` to be satisfied by an execution trace if the `Flow` contains at-least one path of `EventID` objects in which the objects are specified to be in the same order as in some subsequence of the execution trace.
Now, we specify the last two methods of `Flow` using the above relation.

$$\texttt{here(f,}\boldsymbol{trace}\texttt{)} = \begin{cases} true & trace\ satisfies\ \texttt{f} \\ false & \text{otherwise} \end{cases}$$

$$\texttt{since(f,}\boldsymbol{trace}\texttt{,}\boldsymbol{mark}\texttt{)} = \begin{cases} true & trace_{(mark+1)\ldots|trace|}\ satisfies\ \texttt{f} \\ false & \text{otherwise} \end{cases}$$

4 Instrumentation

Evaluation of the flow based assertions requires a comparison with the actual execution trace as described in Sec. 3. This trace—which is a sequence of invocation events—can be recorded using methods provided by some of the specification types.[2] These types and their recording methods are used in a systematic manner to capture the execution trace. This process involves (a) defining specification type trace variables and (b) instrumentation of the template methods. Let us see each in detail.

4.1 Trace Variables

Trace variables can be defined to be of type `HookMethod` or `TemplateMethod`. The recording operators of these trace variables are invoked at runtime from within the instrumented template methods and possibly from their private helper methods. The general strategy for introducing these variables into the framework classes is as follows:

For each hook method[3], we add a static JML ghost variable of type `HookMethod` to the class or interface where the hook is first defined. Similarly for every template method, we add a static JML ghost variable of type `TemplateMethod` to the class containing the template. These ghost variables are used in the specifications[4] as representations of their respective methods. In interest of specification comprehensibility, the ghost variables names are kept similar to the names of the methods that they represent. If no hooks or templates are overloaded and there isn't a field already defined with this name, then the method names should be used as the names for the ghost variables. If there is already a field with the same name or in case of overloaded methods, the variable name should be the method name with some small suffix added.

4.2 Template Method Instrumentation

Since the control flow of the framework resides in the template methods, we assume that all hook invocations occur only within the templates and possibly within any private helper methods called directly or indirectly by the templates. Therefore, only the template methods and any of their helper methods that contain hook invocations in their lexical scope need to be instrumented. This instrumentation is set up as follows:

1. Upon entry into a template method, we invoke the `recordentry()` method of the `TemplateMethod` ghost variable representing this method.

[2] The precise semantics of these recording operations are given in Sec. 3.

[3] We assume that any framework method that can be overridden is a hook method, and that template methods cannot be overridden.

[4] Due to a bug in the JML checker, we had to abstract the ghost variables to model fields and use the model fields in the specifications.

2. Right after every return of a hook call within a template or its private helper method, we invoke the `recordInvocationDone()` of the `HookMethod` ghost variable representing that hook.
3. Just before every normal return of a template method, we invoke the `recordExit()` method of the `TemplateMethod` ghost variable representing this method.

The recording method invocations shown above are implemented by introducing JML `debug` statements at the appropriate points in the template methods, and in any of their private helpers if needed. Figure 8 shows an example template method with the instrumentation code added as JML `debug` statements.

```
public final Form getFirstForm() {
   //@ debug InputDataBroker.getFirstForm.recordEntry();
   ...
   Form firstForm = this.applicationSetup.getForm(...);
   //@ debug ApplicationSetup.getForm.recordInvocationDone();

   firstForm.buildContent(...);
   //@ debug Form.buildContent.recordInvocationDone();

   //@ debug InputDataBroker.getFirstForm.recordExit();
   return firstForm;
}
```

Fig. 8. An instrumented template method

5 Case Study

Following is a brief description of our experiences in documenting a simple Model-View-Controller (MVC) framework using our technique. All the examples used in this paper are adapted from this case study.

5.1 Testbed

The MVC framework that we used as our testbed is a single user, desktop version of Sun's Model 2 web application architecture [4]. It consists of a front-end which displays forms in a sequence, and a back-end which validates and processes the input forms and updates the model. The form display, the validation and processing of the forms and the model updates are all delegated to subclasses as hook invocations. The proper sequencing of these hook invocations is controlled by framework's template methods which encode the abstract MVC functionality.

5.2 Setup

We manually analyzed each template method to determine all possible paths of invocation events that could occur within the scope of that template. Using the results of this analysis and the specification constructs described in Sec. 2, we created the template's flow-based postcondition such that it depicted the analysis results correctly and concisely.

For each of the hook methods, we analyzed the framework classes to determine: (a) which templates could invoke that hook, and (b) what were the possible paths of invocation events that could reach the invocation of this hook. We then constructed the hook's precondition to reflect the analysis results.

Finally, we carried out the instrumentation process as described in Sec. 4.

5.3 Execution

Once the specification and instrumentation were completed, we instantiated the framework into a *real-world* application designed for soil scientists, which involves database reads, floating point computations and chart graphics. This application was then executed with the JML runtime assertion checker [5] using several different input cases.

5.4 Observation

Our test framework was quite small in that it only contained 5 different template methods. Analyzing these template methods to come up with a flow-based postcondition was quite straightforward, since these templates were non-recursive. There was one instance where a hook in the framework was redefined as a template method in a subclass in the same framework. So while we used a flow based precondition to document the hook in the superclass, we used a postcondition in the subclass to document the same method that was now overridden as a template method. Specification of hooks was also straightforward although we had to spend some time picking and choosing the most readable of several alternatives for constructing the same flow based precondition. Due to the small size of the framework, the instrumentation was also quite easy.

Four months after writing the framework's specifications, the first author updated the above-mentioned application by extending its state.[5] Adding this new application-specific state involved most of the design process. Thus the development problem was how to manage the new application-specific state. While the first author had a fair idea about what hooks needed to be overridden, he had forgotten enough about the design to need help reasoning about the details of the application-specific state in which the hooks would be invoked. He

[5] The first author was the creator of both the framework and the application, and thus these observations have to be taken with a grain of salt. As part of future work we plan to have developers not involved in the construction of the framework to instantiate an application and record their experiences.

found that, compared to previous application updates that were done without the benefit of flow-based specifications, this update took less time, since the author never had to consult the source code of the template methods. Consulting the source code of the template methods in previous updates was more painful and time consuming than looking at the succinct flow-based specifications of the hooks. The specifications of the template methods were also consulted in some cases to determine what other hooks would be affected. Overall the experience of using the flow-based specifications was very positive.

6 Discussion

Although we implemented our technique using JML, the method could be implemented in other Design by Contract specification languages. The features needed to implement our technique are fairly standard, however, in the instrumentation part we used ghost variables and debug statements. These features would have to be added to some specification languages or there would have to be some native support for capturing execution traces.

Our technique can be roughly described as a minimalist instruction [3] approach to the formal specification of a framework, which can complement the framework's minimalist natural language documentation [6]. As Chai points out [7], the minimalist approach is based on two ideas:

- People do not want extraneous information when trying to learn a particular task, and
- People are not good at following step-by-step instructions.

For frameworks, the idea is to give the developers the minimal amount of documentation to get the instantiation of the framework. This documentation should be preferably composed of self-contained modules so the developers can read them in whatever order suits them.

Our hope is that the flow-based hook preconditions in our approach can provide such minimal and modular documentation which is sufficient for customization of the hook methods and would enable the developers to quickly get an application up and running. Other approaches with more complex and detailed specifications would, according to the theory of minimalist instruction, make the customization process more difficult.

The original inspiration for this work came from Froehlich *et al.*'s "hook models" [8]. The "hooks" in that work refer to a framework's general abstract customization points and not necessarily just to the code-level overridable hook methods. So we will call them customization points for clarity below. These customization points are specified modularly by mapping each of them to a structured piece of mostly natural language documentation. This document contains sections such as requirements, uses, participants, constraints etc. that contain the respective information for the hook. The objective is to document the intended uses of the framework, and not the design details. The developer can browse a listing of all the hook methods in a framework using a "hook-book"

which contains references to their context descriptions. Although they provide a grammar for structuring the hook descriptions, the system remains at best a semi-formal depiction of each hook's context. Our technique can be seen as aiding the formal specifications of hooks in these hook books. We expect that the context for each high-level customization point will contain the names of the actual hook methods that need to be overridden. Once the developer creates a list of the hook methods to be overridden, she can refer to the flow-based specifications of each method to override it.

7 Related Work

There are several proposals for informally documenting framework design [9,7,10]. However such informal documentation is not easily kept up to date with the framework, whereas our executable specifications can be easily kept up to date.

Büchi and Weck's greybox specifications [11] are written with "abstract programs" in a refinement calculus. These abstract programs allow reasoning about the instantiated application, however, that is different than the customization problem in which we are interested. Abstract programs could, in theory, help with the customization problem, if there was a way to extract the active templates and sequence of hook methods that might be invoked prior to the invocation of a particular hook method. However, extracting such sequences from the abstract programs is non-trivial and currently there is no automated support. Finally, abstract programs do not, by themselves, provide a notation for recording, in the specification of a hook method, all such sequences, and so their notation does not give as much help as ours in for customization.

Soundarajan *et al.* [12,13,14,15], have carried out sustained investigations into using hook invocation traces to formally specify frameworks and applications instantiated from frameworks. Their technique supports incremental reasoning about the instantiated application. However, the only part relevant to the problem we address in this paper is their trace notation.

Their trace notation allows one to specify considerable details about the hook invocations such as argument values, return values, intermediate states and history constraints. Following the minimalist philosophy, we did not attempt to document this level of detail with our flow specification types. Their trace notation can quickly lead to unweildy specifications and is quite poor at expressing flow constructs like deterministic choice and looping, which we feel are essential to effectively document the template control sequence. It is also not clear how much of this notation can be executed (without intolerable overhead). To remedy the problems with their notation, they introduced a programmer-friendly notation scheme called "macros" [12], which are similar to the abstract programs of the greybox technique. However, these macros are currently not executable. Also in the case of complicated template sequences, reading the macros to understand the sequence can be as difficult as reading the actual source code of the template method, because it looks like code.

Next, we will look at several specification languages with trace-based assertions.

The Jass specification language [16,17] can be used to write trace assertions for dynamic checking of object protocols. However, because their notation seems limited to expressing protocols on individual objects, it does not seem easy to express a collaborative protocol distributed across many hook classes. Also, the CSP-based notation for trace specification may be unfamiliar to many developers. Our flow notation is in familiar Java syntax with operators that have straightforward semantics.

Similarly, Cheon and Perumendla's extension to JML [18] does not currently check protocol properties of a set of collaborating objects. Thus it is not suitable for sequences which that span multiple hook classes, as is often the case in frameworks. Also their notation, based on regular expressions, is not rich enough to express the flow constructs that we need; in particular it may not be possible to specify deterministic choice or looping. Our technique, because it uses JML's model methods, is able to express sets of sequences that are not describable by a regular expression.

8 Future Work

The current instrumentation scheme makes the ghost variables that capture the execution trace globally visible and modifiable. We plan to remedy this using some sort of ownership construct that would only allow the template methods to modify the ghost variables of its hooks.

We also plan to investigate how these flow based specifications can be used in model checking of a framework's temporal properties.

We also need a detailed case study on a larger framework using developers who are totally new to the framework. Their experiences in using the flow specifications during instantiation would be the best indicators of the strengths and weaknesses of this documentation approach.

9 Conclusion

Our contribution lies in creating a simple, easy-to-use, trace-based assertion technique for documenting the template-hook interactions of a framework. This technique has been implemented, using JML, and was used in a small case study. Our (limited) experience using this technique was positive. This experience is consistent with the ideas of minimalist instruction theory, which guided the technique's design.

The technique is aimed at providing developers enough information to customize the framework for a particular application's requirements. This customization step involves assigning responsibilities to individual hook methods with respect to their manipulation of the application-specific state. The flow-based specification technique allows each hook method's documentation to have an expressive, readable, and executable description of the necessary information.

References

1. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06-rev29, Iowa State University, Department of Computer Science (2006) To appear in *ACM SIGSOFT Software Engineering Notes*.
2. Borgida, A., Mylopoulos, J., Reiter, R.: On the frame problem in procedure specifications. IEEE Transactions on Software Engineering **21** (1995) 785–798
3. Carroll, J.M.: The Nurnberg funnel: designing minimalist instruction for practical computer skill. MIT Press, Cambridge, MA, USA (1990)
4. Singh, I., Stearns, B., Johnson, M.: Designing Enterprise Applications with the J2EE Platform, Second Edition. Addison-Wesley Professional (2002)
5. Cheon, Y., Leavens, G.T.: A runtime assertion checker for the Java Modeling Language (JML). In Arabnia, H.R., Mun, Y., eds.: Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA, June 24-27, 2002, CSREA Press (2002) 322–328
6. Aguiar, A.: A minimalist approach to framework documentation. In: OOPSLA '00: Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum), New York, NY, USA, ACM Press (2000) 143–144
7. Chai, I.: Pedagogical framework documentation: how to document object-oriented frameworks. An empirical study. PhD thesis, Univiersity of Illinois, Urbana-Champaign (2000)
8. Froehlich, G., Hoover, H.J., Liu, L., Sorenson, P.: Hooking into object-oriented application frameworks. In: ICSE '97: Proceedings of the 19th International Conference on Software Engineering, New York, NY, USA, ACM (1997) 491–501
9. Campbell, R.H., Islam, N.: A technique for documenting the framework of an object-oriented system. Computing Systems **6** (1993) 363–389
10. D'Souza, D.F., Wills, A.C.: Objects, Components, and Frameworks with UML: The Catalysis Approach. Object Technology Series. Addison Wesley, Reading Mass. (1999)
11. Büchi, M., Weck, W.: The greybox approach: When blackbox specifications hide too much. Technical Report 297, Turku Center for Computer Science (1999) http://www.abo.fi/~mbuechi/publications/TR297.html.
12. Hallstrom, J., Soundarajan, N.: Incremental development using object oriented frameworks: A case study. Journal of Object Technology **1** (2002) 189–205
13. Soundarajan, N., Fridella, S.: Framework-based applications: From incremental development to incremental reasoning. In Frakes, W.B., ed.: Software Reuse: Advances in Software Reusability, 6th International Conference, ICSR-6, Vienna, Austria, June 27-29, 2000, Proceedings. Volume 1844 of Lecture Notes in Computer Science., Springer-Verlag (2000) 100–116
14. Soundarajan, N.: Documenting framework behavior. ACM Computing Surveys **32** (2000)
15. Soundarajan, N.: Understanding frameworks (1999)
16. Bartetzko, D., Fischer, C., Moller, M., Wehrheim, H.: Jass - Java with assertions. In: Workshop on Runtime Verification held in conjunction with the 13th Conference on Computer Aided Verification, CAV'01. (2001) Published in *Electronic Notes in Theoretical Computer Science*, K. Havelund and G. Rosu (eds.), 55(2), 2001.
17. Brörkens, M., Möller, M.: Jassda trace assertions, runtime checking the dynamic of Java programs. In Schieferdecker, I., König, H., Wolisz, A., eds.: Trends in Testing

Communicating Systems, International Conference on Testing of Communicating Systems, Berlin, Germany. (2002) 39–48
18. Cheon, Y., Perumendla, A.: Specifying and checking method call sequences of Java programs. Technical Report 05-36, Department of Computer Science, The University of Texas at El Paso, 500 West University Ave., El Paso, TX, 79968 (2006)