

Static Verification of Indirect Data Sharing in Loosely-coupled Component Systems

Lieven Desmet, Frank Piessens, Wouter Joosen, and Pierre Verbaeten

DistriNet Research Group, Department of Computer Science
Katholieke Universiteit Leuven, Celestijnenlaan 200A, B-3001 Leuven, Belgium

Lieven.Desmet@cs.kuleuven.be

WWW home page: <http://www.cs.kuleuven.be/~lieven/research/>

Abstract. To maintain loose coupling and facilitate dynamic composition, components in a pipe-and-filter architecture have a very limited syntactic interface and often communicate indirectly by means of a shared data repository. This severely limits the possibilities for compile time compatibility checking. Even static type checking is made largely irrelevant due to the very general types given in the interfaces. The combination of pipe-and-filter and a shared data repository is widely used, and in this paper we study this problem in the context of the Struts framework. We propose simple, but formally specified, behavioural contracts for components in such frameworks and show that automated formal verification of certain semantical compatibility properties is feasible. In particular, our verification guarantees that indirect data sharing through the shared data repository is performed consistently.

1 Introduction

Current component systems often promote loosely-coupled components to enhance component reuse. The pipe-and-filter style [1] for example is a very popular architectural style for constructing flow-oriented component frameworks. It is often combined with the repository style [1] to support anonymous communication between components. Current state-of-the-art web component frameworks such as Java Servlets [2] or the popular Struts framework [3] are examples of such frameworks.

The main advantage of this kind of architecture is that it makes “wiring” of components at the syntactical level very simple: components are independent entities and interact with the shared data repository through a generic untyped interface. The corresponding drawback is that semantical compatibility checks are absolutely minimal: even compile-time or composition-time type checking is circumvented. For instance, retrievals from the repository are done under the Object type, and the retrieved object is then downcasted to the expected type at run time, potentially leading to exceptions at run time. This in turn significantly hinders independent extensibility of applications built in such frameworks, and reuse of components in new compositions. It is for instance up to the composer to make sure that all data that a given component expects to find on the repository

is guaranteed to be present in the constructed composition. Oversights of the composer can lead to run-time errors.

To enhance component reuse and third-party composability, a precise documentation of the semantical behaviour of the components is essential. By making parts of the component contract formal, automated tool support for verifying some level of semantical compatibility at composition time becomes feasible. As a consequence, certain types of bugs can be detected at compile time or at composition time instead of at run time.

In this paper we propose formal component contracts written in JML, the Java Modeling Language [4], that specify part of the behaviour of components in the Struts framework and we show that static verification with state-of-the-art verifiers for JML is feasible. Our contracts specify for instance what data a component expects on the repository, and what data the component puts onto the repository. Verification checks whether (1) implementations of components honour their contract, and whether (2) compositions always respect the contracts of their constituents. Our approach has been validated on GatorMail [5], an open-source, Struts-based webmail application.

While we have worked out our contracts for the case of Struts, the same idea is applicable to any framework based on the pipe-and-filter and repository architectural styles.

The rest of this paper is structured as follows. Section 2 provides some background information on the web technologies used, component contracts and static verification. Next, the problem statement is elaborated in Sect. 3 and solutions for verifying two composition properties are proposed in Sect. 4. Section 5 validates the proposed solutions in the open-source webmail application GatorMail. In Sect. 6, the presented work is related to existing research and, finally, Sect. 7 summarises the contributions of this paper.

2 Background

2.1 Java Servlets, JavaServer Pages and the Struts Framework

Java Servlets. The Java Servlet technology is part of the J2EE specification [6]. It is a server-side component model for extending the functionality of a web server [2]. A J2EE web application is typically a collection of Java Servlets, deployed in a servlet-based web container such as Tomcat, JBoss or WebSphere. A container casts incoming HTTP requests into an object-oriented form (i.e. a *HttpServletRequest* object) and checks to see if there is a servlet registered for processing that request. During request processing, a servlet can decide to either dispatch the request to another servlet (and by doing so, form a pipe of servlets) or to return a response to the user.

Within a web application, servlets are loosely-coupled with each other (through a very generic interface) and support for dispatching between servlets is provided by the web container. The servlets can communicate anonymously by means of a shared data repository.

JavaServer Pages. The JavaServer Pages (JSP) technology is also part of the J2EE specification and is built upon Java Servlets. JSP enables separation of content from presentation in developing dynamic websites.

JavaServer Pages are used to develop the user interface (or view) of a web application. They are also loosely-coupled, and can communicate anonymously with other JavaServer Pages or Servlets by using the same shared data repository.

The Struts Framework. Apache Struts [3] is a widespread, open-source application framework on top of Java Servlets and JavaServer Pages. Struts encourages developers to use the JavaServer Pages Model 2 architecture [7], a variation on the Model-View-Controller design pattern for web applications.

In a Struts application (illustrated in Fig. 1), incoming HTTP requests are encapsulated in *HttpServletRequest* objects and dispatched to the *ActionServlet*. This *ActionServlet* is the Controller of the Struts application. According to the requested URL, an appropriate action is selected and the *HttpServletRequest* (*Req* in Fig. 1) is processed. An action interacts with the Model and fetches the necessary data for the View. After processing the request, an *ActionForward* object (*AF* in Fig. 1) is returned to the *ActionServlet*, indicating which action or view has to be processed next. This process continues until a JSP view is reached and output is sent back to the web browser. In this architecture only the implementations of the different actions and JSP views are application-specific, the other parts are provided by the Struts framework.

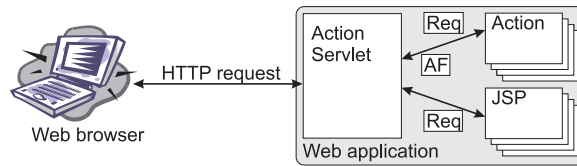


Fig. 1. Request processing in Struts

Actions resemble Java Servlets in that they both process a *HttpServletRequest* and that both are able to use the associated shared data repository that is propagated through the flow together with the request.

In order to achieve reusable actions, an extra forward indirection is used in Struts. Actions use logical names to identify forwards, and the Struts configuration file (which is specific for each configuration) specifies the declarative mapping between logical forwards and actual forwards. In this way, the logical names are mapped to actual forwards at run time using the *ActionMapping* class. The mapping can either be action-specific (local forward) or composition-wide (global forward).

What is important in the context of this paper is the fact that the declarative forwarding and indirect data sharing ensure that actions, servlets and JSP views are very loosely-coupled from a syntactical point of view.

2.2 Component Contracts and Static Verification

Component contracts have already often been proposed before for various purposes [8]. For components written in Java, The Java Modeling Language (JML) [4] is a popular formal contract specification language. In this paper, JML notation is used to specify pre- and post-conditions as well as frame conditions for methods that process HTTP requests. Frame conditions specify what part of the state a method is allowed to modify.

One of the main advantages of JML is the large amount of tool support that is available [9]. Tools are available for run-time contract checking, test generation, static verification and inference of specifications. Of particular interest to us are tools for static verification of JML contracts. A variety of verification tools is available that make different trade-offs in verification power and need for user interaction. In the experiments reported on in this paper, we used the ESC/Java2 verifier [10]. The main advantage of this verifier is that it requires no user interaction. On the downside, the verifier is far from complete, and has some known sources of unsoundness [11, 12]. In Sect. 4.3, we explain how this impacts verification of our proposed contracts.

3 Problem Statement

Although the declarative forwarding mechanism and indirect data sharing in Struts highly facilitate the composition of a web application from a syntactical point of view, they also introduce hidden complexities for the software composer. In order to achieve correctly functioning compositions, the software composer needs to bear in mind all the hidden data interactions through the shared data repository, and anticipate all possible forwards of the actions.

This hidden complexity should not be underestimated. We investigated GatorMail [5], an open-source webmail application of the University of Florida, built upon the Struts framework. In this web application (consisting of about 20.000 lines of code), we identified 36 Struts actions and 29 JSP views, reused in 52 request processing flows [13]. The *FolderAction* for instance was reused in more than 20 processing flows. All the flows contributed to 147 declarative control flow transitions in the webmail application, and to 1369 data repository interactions. The control flow transitions were specified in the composition configuration by means of global and local forwards, but none of the data interactions with the shared repository were documented.

It should be clear that under these circumstances it is not obvious how to reuse existing components or to contribute to an open-source project such as GatorMail, without breaking any of the existing, hidden data dependencies between actions, or without leaving some dangling control flow transitions¹, unless

¹ With a dangling control flow transition, we mean that at run time the action returns a logical forward, but that no mapping can be found to an actual forward in the list of local or global forwards of the running configuration.

of course, a full source code study is undertaken to identify the declarative forwards and the data repository interactions.

To focus on the essence of the problem, we now define a simplified version of the Struts application model. This simplified version mainly takes the declarative forwarding mechanism and the indirect data sharing into account. The presented application model is then used to define some desired composition properties at the end of the section. The problem we address in this paper is how we can verify these properties statically.

The simplified application model is sufficiently generic to reflect the common characteristics of many pipe-and-filter applications with a shared data repository. Hence, the proposed solution of Sect. 4 is generally applicable to this kind of applications. In Sect. 5, the simplified model is further specialised towards the Struts application framework in order to apply our solution to real, existing Struts applications.

3.1 Simplified Application Model

In the simplified application model (shown in Fig. 2), an application is still a composition of actions. All actions implement an *execute* method taking two parameters: a *Request* and a *Form*. A *Request* is a first class entity representing the request that is being processed and the request provides access to the shared data repository (*setDataItem*, *getDataItem* and *removeDataItem*) associated with the request. The *Form* encapsulates the request parameters provided by the client for processing the request.

The *execute* method of an *Action* returns a string, logically indicating which control flow transition should be taken. A *Configuration* object encapsulates the local and global forwards of a composition and maps the strings to corresponding actions. The *RequestProcessor* then repeatedly executes an action for a given request and based on the return value it selects an appropriate succeeding action from the *Configuration*. JSP views are reduced to normal actions in the simplified application model, but they do not produce a forward.

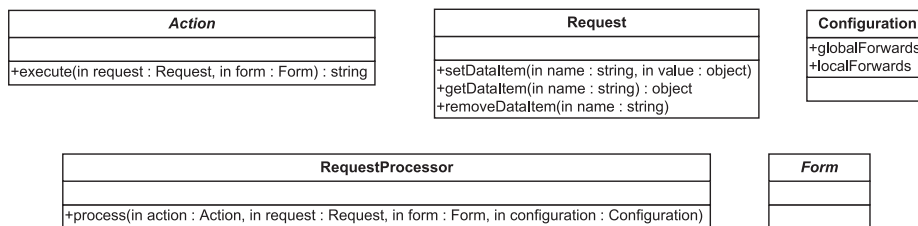


Fig. 2. The simplified application model

3.2 Composition Example

To illustrate the simplified application model, a basic composition example is now introduced. The composition is part of an online calendar system and allows a user to schedule a meeting with several participants at a given time slot and location. The composition consists of four actions and is shown in Fig. 3. The rounded boxes represent actions and the solid arrows indicate control flow transitions.

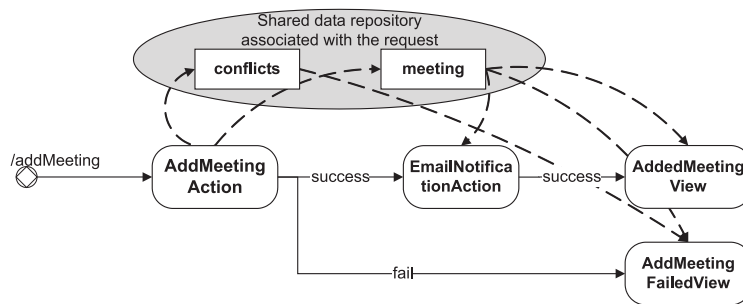


Fig. 3. Composition example: scheduling a meeting

The first action to be executed in scheduling a meeting is the *AddMeetingAction*. This action tries to schedule the requested meeting. On success, the request is processed by an *EmailNotificationAction* which sends a notification to the participants of the meeting. Afterwards, the scheduled meeting is shown to the web user (*AddedMeetingView*). On failure, the *AddMeetingFailedView* lists the different conflicts which make the scheduling impossible.

The labels on the control flow transitions represent the return values of the different actions. The *AddMeetingAction* can either return “success” or “fail”, indicating whether or not the scheduling was successful. The *EmailNotificationAction* only returns “success”, whereas views do not produce a forward.

The interactions with the shared data repository are indicated by dashed lines. The *AddMeetingAction* stores the meeting information (containing the participants, time slot and location) on the shared repository. In case the meeting cannot be scheduled, a list of conflicts is saved as well. All other actions retrieve the meeting information from the shared repository. In addition, the *AddMeetingFailedView* also reads the list of conflicts.

3.3 Desired Composition Properties

Based on the simplified application model, a number of desired composition properties can be defined in loosely-coupled compositions with a declarative control flow and indirect data sharing. Some examples are:

No dangling forwards: Every logical forward in the composition is mapped to an actual forward in the configuration.

No broken data dependencies: A shared data item is only read after being written. For each shared data read interaction, the shared data item that is already written on the repository is of the type expected by the read operation.

In the next section, solutions are proposed to statically verify these composition properties in the simplified application model.

4 Solution

In order to statically verify the composition properties of the previous section, each action is extended with an appropriate action contract. These contracts are then verified in two phases. Firstly, the compliance of the action implementation with the action contract is checked. Secondly, the composition properties are checked based on the different action contracts.

The action contracts are expressed in a framework-specific contract language. Listing 1.1 for example, shows such a framework-specific contract of *AddMeetingAction*. These framework-specific contracts are then translated into JML contracts in order to verify them with existing verifiers. For the rest of the paper we have chosen to show the translated JML contracts since JML is a fairly well-known contract language.

Listing 1.1. Framework-specific contract of *AddMeetingAction*

```
//spec: forwards {"success", "fail"};  
//spec: writes {Meeting meeting};  
//spec: on forward == "fail" also writes {Vector conflicts};
```

This section only highlights key points of the solution. Some additional specification decisions and the full action contracts of the composition example (in the framework-specific contract language and in JML) can be found on the paper's accompanying website [14].

4.1 No Dangling Forwards Property

Action Contracts for the No Dangling Forwards Property. In order to verify the no dangling forwards property, the action contract needs to include sufficient information about the possible declarative forwards (i.e. the different return values). This can simply be done in a JML specification by restricting the return values of an action as part of the action's post-condition. In Listing 1.2 for example, two possible return values are declared in the action's contract: the strings "success" and "fail".

Listing 1.2. Contract for declarative forwarding (*AddMeetingAction.spec*)

```
public class AddMeetingAction extends Action {  
    //@ also
```

```
    //@ ensures \result == "success" || \result == "fail";
    public String execute(Request request, Form form);
}
```

Static Checking of the No Dangling Forwards Property. To check the compliance of the action’s contract with its implementation, a very pragmatic approach such as applying a simple search pattern on the Java source could be used. If however the source code is not that straightforward anymore (e.g. if programming constants are used, or if the return value is constructed in several statements), a static checker tool such as ESC/Java2 can be used to verify the compliance with the ensures clause.

Verifying the no dangling forwards property itself is trivial and can be done by using a simple algorithm that verifies that for each possible return value of the action either a corresponding local forward or global forward exists in the composition-specific configuration. In practice, the declarative forwarding property is not individually verified, but is verified in combination with the no broken data dependencies property as will be explained in Sect. 4.2.

4.2 No Broken Data Dependencies Property

Action Contracts for the No Broken Data Dependencies Property.

The action contracts have to specify the interactions between actions and the shared data repository. These interactions can be expressed in terms of the pre- and post-state of the repository by using the *getDataItem* method of the *Request*.

Because methods used in specifications may not have side-effects, the *getDataItem* method is declared *pure*, i.e. the method will not modify the program state. A more precise definition of purity can be found in [15].

For read interactions, the action’s contract indicates that the action requires that a non-null data item of the specified type can be read from the shared repository, as is shown in Listing 1.3.

Listing 1.3. Contract for indirect data sharing (EmailNotificationAction.spec)

```
public class EmailNotificationAction extends Action {
    //@ also
    //@ requires request != null;
    //@ requires request.getDataItem("meeting") instanceof Meeting;
    //@ ensures \result == "success";
    public String execute(Request request, Form form);
}
```

For write interactions, the ensures pragma states which data items on the shared repository will be non-null and of the specified type after method execution. In Listing 1.4 for example, the JML contract of the *execute* method of *AddMeetingAction* states that the shared data item *meeting* will be a non-null *Meeting* object. Since write interaction may also depend on certain conditions (e.g. if a write interaction occurs in an if-then-else structure), this must also be reflected in the action’s contract. In Listing 1.4 an implication expression (\implies) is used to express that the data item *conflicts* is only written in case the return value equals “fail”.

Listing 1.4. JML contract for indirect data sharing (AddMeetingAction.spec)

```
public class AddMeetingAction extends Action {
    //@ also
    //@ requires request != null;
    //@ ensures request.getDataItem("meeting") instanceof Meeting;
    //@ ensures \result == "fail" ==> request.getDataItem("conflicts") instanceof Vector;
    //@ ensures \result == "success" || \result == "fail";
    public String execute(Request request, Form form);
}
```

Static Checking of the No Broken Data Dependencies Property. To verify the no broken data dependencies property, ESC/Java2 is used to verify both the compliance of the implementation of the *execute* method with the contract, and the composition property itself.

To check the compliance of the action, a specification of the shared repository is introduced, as listed in 1.5. Hereby, *explicit JML pragmas* and a *ghost variable* are introduced for each shared data item, since the current version of the ESC/Java2 tool does not support reasoning about hashtable indirections.

Listing 1.5. JML contract of the shared data repository (Request.spec)

```
public class Request {
    //@ public ghost Object meeting;
    //@ public ghost Object conflicts;

    //@ requires isKey(name);
    //@ ensures name == "meeting" ==> this.meeting == value;
    //@ ensures name == "conflicts" ==> this.conflicts == value;
    public void setDataItem(String name, Object value);

    //@ requires isKey(name);
    //@ ensures name == "meeting" ==> \result == this.meeting;
    //@ ensures name == "conflicts" ==> \result == this.conflicts;
    public /*@ pure @*/ Object getDataItem(String name);

    //@ ensures \result <==> key == "meeting" || key == "conflicts";
    public /*@ pure @*/ boolean isKey(String key);
}
```

To verify the first and second composition property, a *composition-specific check method* is automatically generated and then verified by ESC/Java2. The check method (shown in Listing 1.6) firstly initializes the different actions used in the composition. Secondly, based on the local and global forwards of the composition configuration, a complete control flow graph is statically constructed, similar to what would happen at run time by repeatedly using the *RequestProcessor*.

The *unreachable* pragmas are able to detect violations to the no dangling forwards property, since they are only reachable if an action returns a value that does not match any of its local or global forwards.

The no broken data dependencies property is implicitly verified. Since, for every method call in the method body, ESC/Java2 checks that the preconditions are fulfilled, each data item read must be preceded by a data item write in the execution path and comply with the type requirements in order to satisfy the JML contract of the read interaction.

Listing 1.6. Composition-specific check method to be verified by ESC/Java2

```
//@ requires request != null;
public void check_addMeeting(Request request, Form form){
    AddMeetingAction addMeetingAction = new AddMeetingAction();
    EmailNotificationAction emailNotificationAction = new EmailNotificationAction();
    AddedMeetingView addedMeetingView = new AddedMeetingView();
    FailedAddedMeetingView failedAddedMeetingView = new FailedAddedMeetingView();

    String forward1 = addMeetingAction.execute(request,form);
    if(forward1.equals("success")){
        String forward2 = emailNotificationAction.execute(request,form);
        if(forward2.equals("success")){
            addedMeetingView.execute(request,form);
        } else { //@ unreachable; }
    } else if(forward1.equals("fail")){
        failedAddedMeetingView.execute(request,form);
    } else { //@ unreachable; }
}
```

4.3 Unsoundness with ESC/Java2

ESC/Java2 has a number of known sources of unsoundness [11, 12]. One of these sources also impacts the soundness of our approach, namely ESC/Java2's default handling of framing. As defined in JML, ESC/Java2 has a default for missing modifies clauses (i.e. *modifies everything*) to unhide unexpected changes to variables caused by calling a routine, but logic to reason about routine bodies that contain these modifies clauses has not yet been implemented in ESC/Java2 [12]. As a result, methods without explicit modifies clauses can be verified since the default frame condition includes everything. However in calling such methods, the current implementation of ESC/Java2 does not take this default frame condition into account resulting in an unsound verification. In our case this means that an intermediate action can break the dependencies between one action writing shared data and another action retrieving that data, without ESCJava/2 being able to detect that violation.

To counter this unsoundness, each action annotation is extended with a frame condition, explicitly stating which data items on the shared repository are changed. Also the methods in the *Request* to store and retrieve data from the repository need to have explicit frame conditions. By doing so, ESCJava/2 is able to detect unspecified write interaction with the repository. In addition, other methods interacting with the repository (such as library methods) also require an explicit modifies clause and their contracts need to be verified as well.

Since the current JML notations do not support modifies pragmas in terms of pure methods or hashtable values, the inserted pragmas in the actions are quite verbose (Listing 1.7). In the examples of this paper the modifies pragmas are omitted, but the full annotation with frame conditions can be found at [14].

Listing 1.7. Frame condition of EmailNotificationAction

```
//@ ensures (\forallall String s; request.isKey(s) ==>
           \old(request.getDataItem(s)) == request.getDataItem(s));
```

5 Validation

In this section, we validate the solutions of Sect. 4 in the open-source web-mail application GatorMail. Firstly, we introduce some slight refinements to the presented solution in order to be applicable to real Struts web applications. Secondly, we investigate the JML annotation overhead of the presented approach and the performance of the ESC/Java2 verification tool while verifying the GatorMail web application. Finally, we discuss our validation results.

5.1 Verifying Struts Applications: an Example

To illustrate the verification of Struts applications, a small composition example extracted from the GatorMail application is used. In GatorMail, the web URL `/createFolder.do` is mapped to the composition of Fig. 4 and allows a web user to create a new IMAP mailfolder. The composition consists of three Struts actions and two JSP views. Four control flow transitions occur in the composition: all three action can return a “success” forward, and in addition the `CreateFolderAction` can return a “fail” forward. The interactions of the composition with the shared data repository are listed in table 1.

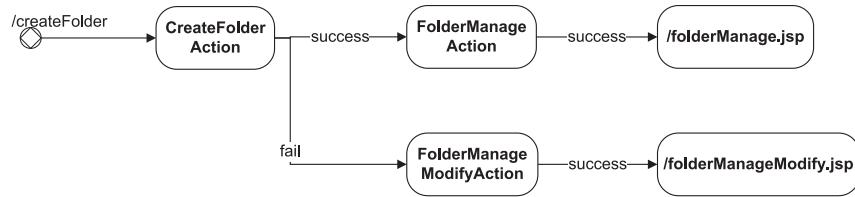


Fig. 4. `/createFolder.do` composition in GatorMail

Table 1. Indirect data dependencies in `/createFolder.do`

Folder folder: FolderManageAction (write) folderManage.jsp (read) FolderManageModifyAction (write) folderManageModify.jsp (read)	String requestStartTime: CreateFolderAction (read/write) FolderManageAction (write) folderManage.jsp (read) FolderManageModifyAction (write) FolderManageModifyAction (write)
ResultBean result: CreateFolderAction (write)	String isSubscribed: FolderManageModifyAction (write) folderManageModify.jsp (read)
List quotaList: FolderManageAction (write) folderManage.jsp (read) FolderManageModifyAction (write) folderManageModify.jsp (read)	List folderBeanList: FolderManageAction (write) folderManage.jsp (read)

Verifying the Declarative Forwarding. In the Struts framework, the *execute* method of an action returns an *ActionForward* object instead of a string. This *ActionForward* does not only encapsulate the declarative forward, but also contains the composition-specific forward path associated with the declarative forward. To do so, the Struts application framework loads the local and global forwards of the composition into the *ActionMapping* object at run time, and the returned *ActionForward* is then constructed by calling the *findforward* method on the *ActionMapping* parameter (Listing 1.8).

Listing 1.8. Declarative forwarding in Struts

```
public class FolderManageAction extends Action {
    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response) throws Exception {
        // ...
        return mapping.findForward("success");
    }
}
```

To be able to express the local forward string in the JML contracts of the actions, extra specification is introduced for *ActionMapping* (Listing 1.9). The specification states that the declarative forward used as parameter of the *findForward* method is equal to the name property of the returned *ActionForward*. By doing so, the declarative forwards can be expressed in term of the name property of the returned result (Listing 1.10).

Listing 1.9. JML specification of ActionMapping

```
public class ActionMapping extends ActionConfig {
    //@ requires name != null;
    //@ ensures \result != null;
    //@ ensures \result.getName() == name;
    public ActionForward findForward(String name);
}
```

Listing 1.10. Declarative forward specification of FolderManageAction

```
//@ ensures \result.getName() == "success";
```

Verifying Indirect Data Sharing. Since indirect data sharing via a shared repository in Struts is identical to the simplified application model, the solution of Sect. 4 can be applied to Struts applications without any modification.

5.2 Results of the GatorMail Experiment

To validate the applicability of our approach, we annotated 12 actions and 8 views of the GatorMail webmail application. With these annotations we were able to verify the declarative forwarding and indirect data sharing properties in 17 composition flows (i.e. one third of all flows in GatorMail). We used this subset of the application to investigate the annotation overhead and the performance of the verification. Only the results are reported in this subsection. The full annotations and a short description of how to verify both the implementation conformance and the composition properties can be found at [14].

JML Annotation Overhead. As a quantification of the annotation overhead, a JML line count is performed on the annotated actions. As shown in table 2, at most 15 lines of JML annotation are used in an action contract to express the control flow transitions and the shared repository interactions. The JML contract of *FolderAction* for example, consists of 9 annotation lines, illustrated in Listing 1.11. But this quite verbose JML contract is actually generated from a more concise, Struts-specific contract specified in Listing 1.12.

The Struts-specific contracts are at most 4 lines of annotations, and they are much easier to write by a Struts developer or to read by a software composer. The Struts-specific contracts of the GatorMail case and a tool for converting them into the verifiable JML annotation can be found at [14].

Table 2. JML notation overhead in GatorMail

Action	# JML lines	Action	# JML lines
ChangeSubscribedAction ²	14	FolderManageAction	10
CheckSessionAction	7	FolderManageModifyAction	11
CreateFolderAction	10	ModifyFolderAction ²	15
DeleteFolderAction	10	MoveCopyAction	11
DeleteMessagesAction	10	PerformDeleteFolderAction ²	15
FolderAction	12	RenameFolderAction	9

Listing 1.11. JML contract of FolderAction

```

//@ also
//@ requires request != null;
//@ requires mapping != null;
//@ ensures \result != null;
//@ ensures \result.getName() == "success" || \result.getName() == "inbox";
//@ ensures request.requestStartTime instanceof Long;
//@ ensures \result.getName() == "success" ==> request.folderBeanList instanceof List;
//@ ensures \result.getName() == "success" ==> request.folder instanceof Folder;
//@ ensures \result.getName() == "success" ==> request.messages instanceof List;
//@ ensures \result.getName() == "success" ==> request.quotaList instanceof List;
//@ requires form instanceof FolderForm;

```

Listing 1.12. Struts-specific contract of FolderAction

```

//struts: forwards {"success","inbox"};
//struts: writes {Long requestStartTime};
//struts: on forward == "success" also writes {List folderBeanList, Folder folder,
List messages, List quotaList};

```

Verification Performance with ESC/Java2. To evaluate the performance of the verification process, the verification time and memory usage is measured for verifying the implementation compliance and the composition properties. The performance tests were run on a Pentium M 1.4 with 512MB RAM, running Debian Linux, while using Java 1.4.2.09, ESC/Java2 2.0a9 and Simplify 1.5.4.

² These actions extend the *LookupDispatchAction*, and have alternative substitutes of the *execute* method. Thus, it's obvious that these actions have a higher JML line count, since several methods are annotated.

Table 3 shows the performance results of verifying a subset of GatorMail. Both verification steps can be done in a reasonable amount of time (less than 15 seconds per verification) and limited memory resources (not exceeding 25MB). If also the frame conditions are checked, the verification takes up to 700 seconds, but since most bugs are already found without checking the frame conditions, this type of verification has to be run less regularly. In addition, since the verification is done modularly (i.e. action per action), the verification complexity is linear and the the verification process is scalable to larger software projects as well.

Table 3. Verification performance

Action	Verification time (with frame cond.)	Mem. usage	Composition flow	Verif. time	Mem. usage
ChangeSubscribedAction	1.960 s (13.151 s)	16 MB	/folder.do	0.853 s	14 MB
CheckSessionAction	0.252 s (2.241 s)	13 MB	/folderManage.do	0.506 s	15 MB
CreateFolderAction	0.951 s (5.106 s)	15 MB	/folderManageModify.do	0.555 s	15 MB
DeleteFolderAction	0.978 s (61.193 s)	17 MB	/createFolder.do	1.639 s	17 MB
DeleteMessagesAction	4.607 s (24.542 s)	20 MB	/renameFolder.do	1.741 s	17 MB
FolderAction	14.18 s (711.654 s)	24 MB	/changeSubscribed.do	1.733 s	18 MB
FolderManageAction	1.407 s (10.475 s)	16 MB	/deleteFolder.do	1.145 s	18 MB
FolderManageModifyAction	2.126 s (205.791 s)	16 MB	/performDeleteFolder.do	2.497 s	19 MB
ModifyFolderAction	0.831 s (1.699 s)	14 MB	/modifyFolder.do	7.638 s	22 MB
MoveCopyAction	4.334 s (20.957 s)	19 MB	/deleteMessages.do	1.819 s	23 MB
PerformDeleteFolderAction	1.390 s (5.833 s)	16 MB	/moveMessage.do	2.468 s	24 MB
RenameFolderAction	0.844 s (4.993 s)	15 MB	/copyMessage.do	1.960 s	25 MB
			/moveMessages.do	2.338 s	17 MB
			/copyMessages.do	1.936 s	19 MB
			/errorCopy.do	0.435 s	20 MB
			/errorCopyToSent.do	0.725 s	20 MB
			/errorCopyTrash.do	0.446 s	18 MB

5.3 Discussion

One of the problems that we were confronted with was ESC/Java2's poor support to reason about hashtable indirections. Since the dynamics of loosely-coupled component systems such as Struts strongly rely on hashtable indirections in the implementation, we were forced to circumvent this lack of support by introducing very verbose specifications or statically constructing the complete control flow graph. Additionally, ESC/Java2 is far from complete, for instance reasoning about loops is fairly weak. Also, known sources of unsoundness, related to framing and reentrancy need to be avoided. Again, this made specifications more verbose than they could be. This is however a temporary problem and future versions of the tool are expected to improve in the different domains.

Another issue that we encountered in verifying GatorMail was the violation of the Liskov substitution principle. The *DeleteMessagesAction* for example extends the *FolderAction*, while having a stronger precondition regarding the expected data items on the shared repository for the *execute* method. Since verification tools rely on the Liskov substitution principle, we had to slightly refactor GatorMail in order to comply with the Design by Contract concept.

While the GatorMail case study shows that annotation overhead and verification performance are fine, it can not give us data about the usefulness of our approach for detecting bugs early. Since GatorMail is a mature application, bugs due to broken dependencies have been ironed out already. Therefore, it would be interesting to apply our approach to less mature software systems or to study a development process that incorporates our approach in future research.

6 Related Work

To the best of our knowledge, this is the first proposal for automatic verification of indirect data sharing in Java-based component frameworks. However, our approach is strongly inspired by ongoing research in several research domains.

In software architecture research, several Architecture Description Languages (such as Wright, Darwin and Rapide) are proposed to support architecture-based reasoning, ranging from semi-formal diagrams with boxes and lines to formal notations [16]. Architecture analysis techniques have already been developed to detect problems such as deadlock and component mismatch [17, 18].

Comparable approaches (such as CL [19] and Piccola [20]) are proposed in the domain of coordination and software composition. CL, for example, is a composition language for predictable assembly from certifiable components. In CL, the run-time behaviour of an assembly of components can be predicted from known properties of components and their patterns of interaction [19].

The use of JML or related languages such as Spec# [21] for verifying component properties is a very active research domain. For example, Smans et al. [22] specify and verify code access security properties, Jacobs et al. [23] verify data-race-freeness in concurrent programs, and Pavlova et al. [24] focus on security properties of applets. Other applications of JML are surveyed in [9].

7 Conclusion

This paper has focussed on two desirable composition properties in pipe-and-filter and repository based component systems. We proposed framework-specific component contracts to specify a component's possible forwards and its interactions with the shared repository and translated them into JML annotations. The contracts are sufficiently simple to have an acceptable annotation overhead and a very reasonable automatic verification time.

Although, as discussed in Sect. 5.3, there are still some drawbacks with the current state of the verification tool, the conducted experiments show that using existing contract annotation languages and verification tools in order to achieve more robust compositions looks promising.

Acknowledgements

The authors would like to thank Bart Jacobs, Adriaan Moors and Jans Smans for their useful comments and insights while proofreading this paper.

References

1. Shaw, M., Garlan, D.: Software Architecture - Perspectives on an emerging discipline. Prentice-Hall (1996)
2. Java Servlet Technology. (<http://java.sun.com/products/servlet/>)
3. The Struts Framework. (<http://jakarta.apache.org/struts/>)
4. The Java Modeling Language (JML). (<http://www.jmlspecs.org/>)
5. GatorMail WebMail. (<http://sourceforge.net/projects/gatormail/>)
6. J2EE platform specification. (<http://java.sun.com/j2ee/>)
7. Seshadri, G.: Understanding JavaServer Pages Model 2 architecture. (<http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc.html>)
8. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
9. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer (STTT) **7**(3) (2005) 212–232
10. KindSoftware: The Extended Static Checker for Java version 2 (ESC/Java2). (<http://secure.ucd.ie/products/opensource/ESCJava2/>)
11. Leino, K.R.M., Nelson, G., Saxe, J.B.: (ESC/Java User's Manual)
12. Cok, D.R.: (ESC/Java2 Implementation Notes)
13. Desmet, L., Piessens, F., Joosen, W., Verbaeten, P.: Dependency analysis of the Gatormail webmail application. Report CW 427, Department of Computer Science, K.U.Leuven, Leuven, Belgium (2005)
14. Desmet, L., Piessens, F., Joosen, W., Verbaeten, P.: Static verification of composition properties. (<http://www.cs.kuleuven.be/~lieven/research/SC2006/>)
15. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06-rev28, Iowa State University, Department of Computer Science (2005)
16. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Trans. Softw. Eng. **26**(1) (2000) 70–93
17. Inverardi, P., Tivoli, M.: Automatic synthesis of deadlock free connectors for com/dcom applications. In: Proceedings of the 8th ESEC held jointly with 9th ACM SIGSOFT FSE, ACM Press (2001) 121–131
18. Allen, R., Garlan, D.: A formal basis for architectural connection. ACM Trans. Softw. Eng. Methodol. **6**(3) (1997) 213–249
19. Ivers, J., Sinha, N., Wallnau, K.: A Basis for Composition Language CL. Technical Report CMU/SEI-2002-TN-026, SEI, Carnegie Mellon University (2002)
20. Achermann, F., Nierstrasz, O.: Applications = Components + Scripts — A Tour of Piccola. In Aksit, M., ed.: Software Architectures and Component Technology. Kluwer (2001) 261–292
21. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview. Lecture Notes in Computer Science **3362** (2004)
22. Smans, J., Jacobs, B., Piessens, F.: Static verification of code access security policy compliance of .NET applications. Journal of Object Technology **5**(3) (2006)
23. Jacobs, B., Leino, K.R.M., Piessens, F., Schulte, W.: Safe concurrency for aggregate objects with invariants. In: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods, IEEE Computer Society (2005) 137–146
24. Pavlova, M., Barthe, G., Burdy, L., Huisman, M., Lanet, J.L.: Enforcing high-level security properties for applets. In: CARDIS. (2004) 1–16