VERIFICATION OF WEB APPLICATIONS WITH A MODEL CHECKER

Christian Ammann
Faculty of Engineering and Computer Science
University of Applied Sciences Osnabrück
49009 Osnabrück, Germany
email: c.ammann@hs-osnabrueck.de

ABSTRACT
Web applications are hosted on a web server and executed within a browser environment. They are widely used and typically developed with HTML and Javascript. They often interact with web servers which makes stability and security important aspects in the development process. One way to increase both aspects is the usage of a model checker which can verify whether a software model contains errors and does not meet its requirements. The integration of formal verification in a web application software project is challenging because expert knowledge is necessary. This paper solves this problem with the usage of model-driven development and presents a domain-specific web application modelling language. Furthermore, an algorithm for an automated transformation into the model checker input language Promela is provided. Promela is used by the Spin model checker which tests automatically whether the corresponding web application model meets the given specification, informs the developer about possible errors and therefore increases software stability and security.

KEY WORDS
Modelling Languages, Formal Methods, Web Applications, Domain-Specific Languages, Model Checking

1 Introduction
Web applications provide benefits like cross-platform compatibility (e.g. Mac and Windows) and a simple upgrade procedure (a new version is deployed on the web server and automatically delivered to the users). They can run standalone in a web browser but also communicate with web servers (e.g. via web services [1]) and are typically developed in Javascript [2] and the HyperText Markup Language (HTML) [3]. Well-known examples for web applications are the Google Calendar or Redmine [4].

The Google Web Toolkit (GWT) [5] is an open source framework for the development of web applications. GWT applications can be divided into a client part which runs in a browser and a server part which is executed on a web server. The server side of a GWT project is implemented with Java. The client side consists of static and dynamic elements. Static elements are described with HTML while dynamic elements are written in Java and automatically transformed by GWT into JavaScript. This provides the following advantages:

- Almost the complete web application can be developed with Java and therefore the existing tool support for Java like the Eclipse IDE is available.
- It is possible to debug the GWT Java code while the web application is running in a browser because GWT provides a browser plugin which performs a real time conversion from Java to Javascript and Javascript to Java.
- No expert knowledge about the different browsers is necessary because GWT automatically generates optimized Javascript code for the corresponding platforms (Chrome, Firefox, etc.)

A GWT application can use remote procedure calls (RPCs) for the communication with a server. RPCs are asynchronous and therefore GWT provides a special callback object. This object contains the method onSuccess() which is executed when the RPC terminates.

Web applications run in a browser environment which uses the HTTP communication protocol. Pure HTTP is stateless and therefore GWT supports session-ids which are implemented with session objects. Each session object is associated with a client and contains a hash map. The hash map is used by the server side as data storage (e.g. the user name after a successful login).

Web applications can be business critical and security relevant: E.g. a high degree of stability and security is mandatory for a bank and their stock trading web application because an attacker could abuse the stock trading RPCs to corrupt internal server data. Therefore, quality assurance focusing on security is an important aspect in the web application development process.

One way to improve quality is the usage of a model checker which verifies automatically whether a software system meets its requirements. For a successful verification, the web application is transformed into the model checker’s input language and enriched with formal requirements (e.g. assertions or LTL-formulas [6]). Afterwards, the model checker tests automatically whether the model meets the specification and typically generates an error path if a requirement is not met. This approach leads to two problems:
1. Consistency: Most parts of a GWT application are written in Java and have to be transformed into the model checker input language. Therefore, it is necessary after each modification of the GWT application to validate whether both models (Java and model checker input language) are still equivalent.

2. Model Transformation: Expert knowledge about the corresponding model checker and its concepts are necessary for a transformation into the model checker's input language.

It is the objective of this paper to solve both problems and allow developers an easy formal verification of their web applications.

Our solution is a combination of model checking and model-driven development (MDD) [7]. MDD is a software engineering approach which focuses on creating models instead of writing software directly. Additionally, transformation templates are created which automatically transform models into executable software. One way to describe models are domain-specific languages (DSLs) which usually operate on a higher abstraction level than programming languages like Java and which are a well-suited for formal verification.

Therefore, we propose in this paper the usage of a domain-specific language for web applications and an automated transformation of the DSL into a Java GWT project and (as an extension of MDD) a model checker input language. The workflow of this approach is shown in figure 1: A developer models a web application with the DSL and uses the transformation templates for an automated translation into a model checker input language. The generated model checker input language is passed to a model checker which verifies whether all requirements are met. When an error is reported the model is fixed and verified again. When no error is reported the model is automatically transformed into a Java GWT project, passed to GWT, compiled and deployed on a web server.

The advantages of this approach are: A developer can use the existing transformation templates which perform an automated model transformation into the target languages. Detailed knowledge about model checking is not necessary and modifications (e.g. if the model checker reports an error) have to be applied to the DSL and affect automatically the Java GWT project and the model checker code.

The corresponding domain-specific language for the description of GWT applications is presented in section 2 (together with a login mask case study). It provides language constructs for the description of RPCs, the client side, session ids and callback objects. Behaviour is expressed with an action language which also allows the specification of requirements via *assert statements*. The DSL is developed with the Xtext/Xpand [8] framework for the Eclipse IDE. Xtext allows the creation of custom DSLs while Xpand provides a language (and the corresponding interpreter) for the description and execution of model transformation templates which automatically translate an Xtext DSL into a target language. Compared with other frameworks like Yacc/Bison [9], Xtext and Xpand provide advanced features like native scoping support or the automated generation of an editor with syntax highlighting and validation.

A translation algorithm from our DSL into a model checker input language has to be invented before the Xpand language can be used for the implementation of a transformation template. Therefore, we present in section 3 a transformation algorithm into Promela which is the input language of the Spin model checker [10]. We use Spin because it is designed for the verification of distributed software systems which is also the domain of web applications. Furthermore, Spin contains optimisation techniques like state space compression or *Bitstate Hashing* [11] for the verification of larger models.

Other research projects also investigate the verification of web applications. We present corresponding related work in section 4 and compare it with our approach.

The following aspects are not described in this paper and are therefore further work (which is described in more detail in section 5):

---

Figure 1. Extending MDD with a Model Checker
• A translation algorithm from our DSL into Java.

• An Xpand template which implements the Java translation algorithm and allows an automated model transformation from the DSL into Java.

• An Xpand template which implements the Promela translation algorithm from section 3 and allows an automated model transformation into Promela.

This paper is part of the KoverJa [12] (Correct distributed Java Applications) project which is supported by the German Federal Ministry of Education and Research (BMBF).

2 A DSL for Web Applications

This section introduces the Xtext/Xpand framework for the implementation and transformation of custom DSLs. Afterwards, we present a simple GWT case study and derive from it an Xtext DSL for the description of web applications.

Xtext and Xpand are plugins for Eclipse and allow the implementation and transformation of DSLs in a software project. Xtext contains a language for the description of grammars and a parser generator. The grammar description language consists of production rules. Xtext invokes ANTLR [13] after the implementation of the production rules and creates a parser for the corresponding language, classes for the abstract syntax tree, etc. Furthermore, an editor for the DSL is generated with features like validation and syntax highlighting.

Xpand is a template language which can be used to transform an Xtext DSL into a target language. It contains language features to navigate through a models abstract syntax tree (with loops, if/else statements, etc.) and to write textual output into a file. It is also possible to access Java and its class library. Xpand has an open architecture and provides interfaces for the integration of pretty printers or other functionality.

We present now a simple GWT application and use Xtext to develop a GWT DSL. The Xtext grammar for the web application DSL is available at [14]. A screenshot of this case study can be seen in figure 2. It consists of an interface which allows a user to enter his name and a password. Both are transmitted with a remote procedure call to a webserver. The server verifies whether userName and password are correct and associates the result with the corresponding browser session. Afterwards, a new HTML page is opened which displays a welcome message if the login was successful or otherwise an error message. The complete GWT Java project is available at [14]. The DSL for the implementation of the case study consists of the following elements:

• Each web client has a unique session-id which is bound to a session object. A session object consists of a set of variables which can be accessed by the server side.

• The client side is represented by a set of pages. A page is comparable with a HTML page and contains GUI elements (e.g. buttons). Furthermore, it can provide callback objects for the RPCs and an onLoad() function which is executed when the page is loaded. Each callback object contains an onSuccess() method which is executed when the RPC returns.

• The server side contains a set of methods which are called as RPCs by the client. RPCs are asynchronous and therefore a reference to a callback object is added to the parameter list.

Next, we demonstrate the usage of our GWT DSL and describe the login case study. We start with the description of the server side. The first listing contains the session object:

```
1    session { bool valid_user = false; }
```

Listing 1. Session Object

The session object contains a boolean variable which is set to true after a successful login. The next listing provides the corresponding RPCs which use the session object to verify whether a login was successful:

```
1    server {
2        void verifyLogin (string user,
3                    string password){
4            if (user.equals("smith") &&
5              password.equals("secret")){
6                valid_user=true;
7            }
8            else {
9                valid_user=false;
10           }
11           return;
12        }  
13    }
14    string getWelcomeMessage (){   
15        if (valid_user){ return "Welcome"; }
16        else{ return "Not a valid user"; }
17    }
```

Figure 2. GWT Login Application
The first RPC verifyLogin() has two parameters: username and password. The function evaluates whether both are correct (the only valid username/password pair in this example is smith/secret) and stores the result in valid_user. The second RPC getWelcomeMessage() checks whether the calling client is a valid user and returns a string depending on the result.

We present now the client side of our GWL DSL case study which consists of two pages: Login and Welcome.

The page Login consists of the following elements: a callback object for the verifyLogin() RPC, a textbox for the username, a textbox for the password, a login button and some labels. The elements are aligned within vertical and horizontal environments. Textbox objects provide a get() method which returns the user input. Each button object contains a behaviour block which is executed when the corresponding button is pressed by the user. When the login button in listing 3 is pressed, the content from the textboxes is read and transmitted via remote procedure call (verifyLogin()) to the server. RPCs are asynchronous and have a reference to a callback object as an additional parameter. The callback object loginCallback opens the Welcome page when the RPC is finished:

```java
page Login{
    callback loginCallback {
        void onSuccess() { openPage(Welcome); }
    }

    vertical{
        horizontal{
            label usernameLabel { text = "User: " }
            textbox username{}
        }
        horizontal{
            label passwordLabel { text = "Pass: " }
            textbox password{}
        }
        horizontal{
            button login{
                text = "login",
                behaviour{
                    verifyLogin(username.get(),
                                password.get(),
                                loginCallback);
                }
            }
        }
    }
}
```

Listing 3. Client Side Login Page

The page Welcome calls the RPC getWelcomeMessage() in its onLoad() function. The callback object welcomeCallback receives the return value of the RPC and displays it on the Welcome page. The complete DSL example is available at [14].

### 3 Transformation to Promela

In the last section we presented a domain-specific language for the description of GWT web applications. This section provides a transformation algorithm from the DSL to Promela.

Our DSL contains an action language which allows the declaration of statements, expressions and variables. Variables support the data types boolean, byte, integer and string. Except for string variables, the elements of the action language are Promela compatible and do not have to be modified during the transformation process. Strings are special because Promela does not provide a string data type. Instead, it is possible to implement strings with byte arrays. The disadvantage of this approach is: Each Promela variable (also byte arrays) is part of the model’s state vector, increases the model checker’s memory consumption and can lead to the state space explosion problem [15].

Therefore, we use abstraction for the implementation of strings: Promela supports enumerations (so called mtypes). During the model transformation we generate a special enumeration which is used for string abstraction. The enumeration elements represent the strings which are used in the model. The strings in listing 2 are transformed into the following Promela code:

```java
mtypes{ smith, secret, unknown }
```

Listing 5. Strings in Promela

Besides smith and secret, a user could enter something else in a textbox. We model this class of input strings with the element unknown. The next listing demonstrates the implementation of the get() method which is provided by each textbox object:
inline get(ret) {
    if :: ret = smith;
    :: ret = secret;
    :: ret = unknown;
    fi;
}

Listing 6. Get() Method in Promela

The if-block assigns non-deterministically smith, secret or unknown to ret which has one major consequence: The model checker verifies the system behavior for all possible return values of the function and therefore all possible user inputs in the test field.

This abstraction technique can also be customized because our DSL so-called interface methods. Interface methods are functions without a function body. They can be used for property preserving abstraction of components in which the concrete implementation is not important for the model checker or can not be described in the model checker input language. A use case could be an interface to a database which contains a set of users and their corresponding age. The server side of a web application can query the database and grant access to a certain content if the user is 12 years or older. In this case, for the web application server side it is only interesting whether a user is old enough or not. A complete description of a database applications server side it is only interesting whether a user is 12 years or older. In this case, for the web applications server side it is only interesting whether a user is old enough or not. A complete description of a database and its interaction with the web server is not necessary and would increase the models state space. Instead, an interface function can be used which is illustrated in the following listing:

enum Age{younger_than_12, old_enough}

interface Age queryDataBase
    (string username)

Listing 7. Interface Function Example

An enum (also part of the GWT DSL) describes the possible age of a user. The function queryDataBase is an interface method and returns either younger_than_12 or old_enough. The transformation into Promela is performed similar to listing 5 and 6 which causes the model checker to verify all possible return values each time the method is called.

Another important aspect on the client side is the execution of the onSuccess() method which is a part of each callback object. An RPC is asynchronous which makes it difficult to estimate the exact time when the RPC returns and onSuccess() is executed. This is a problem because the execution time of the onSuccess() method influences the Promela model. We performed empirical measurements to estimate whether the onSuccess execution time is deterministic or non-deterministic and describe our results using the following example:

page ExecutionTimeExample{
    byte a;
    callback callback1{
        void onSuccess() { a = 1; }
    }
    callback callback2{
        void onSuccess() { a = 2; }
    }
    onLoad{
        rpc1(callback1);
        rpc2(callback2);
        a = 3;
    }
}

Listing 8. OnSuccess Execution Time Example

Listing 8 is written in the GWT DSL and contains the page ExecutionTimeExample. It calls the RPCs rpc1(), rpc2() and assigns a value to the variable a. For simplification, the RPCs are not part of this example and we assume that they perform some calculations. A non-deterministic execution time of the onSuccess() method would lead to the following Promela model:

byte a;

if :: true -> a = 1; a = 2; a = 3;
    :: true -> a = 2; a = 1; a = 3;
    :: true -> a = 1; a = 3; a = 2;
    :: true -> a = 2; a = 3; a = 1;
    :: true -> a = 3; a = 1; a = 2;
    :: true -> a = 3; a = 2; a = 1;
fi;

Listing 9. Non-deterministic Execution Time

An if block is used to model the interleavings which allows the model checker to chose non-deterministically the execution order of the onLoad and onSuccess assignments. Obviously, this is not practicable for larger web applications because onSuccess() and onLoad() methods with more than one statement would lead to very large models and provoke the state space explosion.

Luckily, our empirical measurements showed a deterministic behaviour of the onSuccess execution time which uses the following algorithm:

- When a page is loaded, the content of the onLoad() method is executed. The called RPCs run as concurrent processes.
- After the execution of the last onLoad statement the client waits until the called RPCs are finished.
- The corresponding onSuccess() methods are executed (the execution order is equivalent the RPC call order).
This behaviour allows us an easy transformation of DSL callback objects into Promela. The next listing contains the transformation of listing 8 into Promela and demonstrates our transformation algorithm:

```java
byte a;
chan rpc1_finished = [0] of { bool };
chan rpc2_finished = [0] of { bool };

// onLoad starts here
run rpc1( rpc1_finished );
run rpc2( rpc2_finished );
a = 3;
// onLoad ends here
rpc1_finished?true;
rpc2_finished?true;
a = 1;
a = 2;
```

Listing 10. Deterministic Execution Time

A synchronous communication channel is created for each remote procedure call. The RPCs are implemented with Promela processes because they run concurrently to the web clients. An RPC is called with the run statement which starts the corresponding concurrent process and can also be used to pass the RPC parameters to the Promela process. As an additional parameter each RPC gets a reference to a message channel and sends a signal to this channel before it terminates. After the execution of the onLoad block, the web client waits until both RPCs are finished and afterwards begins with the execution of the onSuccess() methods.

We have described the transformation of the action language, the callback objects and the server side RPCs. Yet, the translation of the page elements is still missing. One problem is the behaviour of pages in different browsers. We have already mentioned that a page can provide an onLoad() method which is executed when the page is opened. The question in this case is: How does a GWT application behave when a user presses the forward or backward button of his browser while the onLoad() method is still running? Empirical tests have shown the following behaviours:

- The Internet Explorer 9 continues in background with the execution of the onLoad() method while the new page is loading.
- Google Chrome 14 blocks the forward/backward buttons while the onLoad() method is running.

We have decided to implement the behaviour of Google Chrome in this paper and leave investigations for other browsers for further work. The next listing contains a GWT DSL example which is used to demonstrate our transformation of page elements into Promela:

```java
page A{
    button button_a{
        text = "button_a"
        behaviour{ /* ... */ }
    }
}

onLoad{ /* ... */ }
}

page B{ /* ... */ }
```

Listing 11. Page Transformation Example

In listing 11 we removed some elements for simplification and inserted comments instead. The example consists of the pages A and B. Page A contains a button with a behaviour block and an onLoad function. We now show the transformation of listing 11 into Promela:

```java
process WebClient(){
    if
        :: goto page_a;
        :: goto page_b;
    fi;
    page_a:
        onLoad();
        if
            :: goto select_page;
            :: goto button_a;
        fi;
        button_a:
            behaviour();
            if
                :: goto button_a;
                :: goto select_page;
            fi;
            page_b:
                /*...*/
}
```

Listing 12. Transformation of Page Elements into Promela

Each web client is modeled as a Promela process (it is therefore necessary to start at the beginning of the verification process the total amount of supported web clients). The model checker can choose a page nondeterministically which is opened when the client is started. Afterwards, it can execute the onLoad() method and decide whether a new page is opened or a button was pressed. This reflects the behaviour of a user who enters a new URL or uses the forward/backward buttons of his browser when a script is finished. The complete transformation example of the login web application is available at [14].

4 Related Work

Leung et al. [16] describe a modeling and verification approach for web pages based on Harel statecharts [17].
Harel statecharts are finite state machines which support concurrency, can be hierarchical and communicate with synchronous or asynchronous events. Leung et al. divide a web application into a set of pages and a set of links which connect pages with each other. Afterwards a mapping from web applications to statecharts is presented. Basically pages are mapped to states while links a mapped to transitions. When the mapping is done, Spin can be used as a model checker for formal verification [18]. This approach leads to two problems: It is difficult to implement the dynamic elements of a web application because Harel statecharts have a fixed amount of states. Furthermore, the communication with a web server (which can be critical because of deadlocks) is not taken into account.

Deutsch et al. [19] describe a web application specification language which is inspired by WebML [20]. It can be used to model web applications which consist of web pages, a data base and user interaction. Each web application provides at least a start and an error page. Web applications are transformed into abstract state machines (ASM) and verified with a custom model checker for ASM transducers [21]. The requirements can be specified with LTL-formulas. Our approach differs from [19] in the following aspects:

- We verify a more complete set of possible user interactions: Deutsch et al. assume for example that the execution of a web application always begins with the start page. A user can access any page of a web application at any time and therefore our approach considers each page as a possible start page (see listing 12 for an example) during the verification process.
- Deutsch et al. allow only interaction with a data base while our property preserving abstraction technique can be used to model any external component.
- We use the Spin model checker which provides tools like TopSpin [22] for symmetry [23] reduction which is not supported by Deutsch et al.

Nakajima et al. [24] investigate the verification of web service flows. They adapt the WSFL (Web Services Flow Language) [25] and use the Spin model checker for formal verification. Their work is motivated by the fact that conventional testing of distributed web services is difficult because tremendous amount of publicly shared network resources would be consumed.

A web service flow is a directed graph which consists of activities (nodes) and a data/control flow (edges). The nodes are mapped to Promela processes while the edges are represented by message channels. The whole approach is demonstrated with a travel agency example which uses a variety of web services of business partners such as airline carriers, hotels, etc. A missing aspect is the concrete implementation of a web service. In contrast, our approach allows a developer either to use property preserving abstraction and to hide the concrete implementation or to use the action language for a detailed description. Furthermore, we can also verify complete web applications while in [24] only the server side is taken into account.

5 Conclusion

This paper describes a DSL for GWT web applications which is implemented with Xtext. We also provide a translation algorithm into the model checker input language Promela. After the transformation the model checker Spin verifies whether the GWT DSL model violates any assertion, contains deadlocks or out-of-bounds access on arrays. This allows e.g. a developer to check automatically whether a user without a valid password can gain access to protected regions of the web application through the exploitation of program errors. Furthermore, the DSL provides symmetric arrays and property preserving abstraction to avoid the state space explosion problem [15] for larger models.

Further work will be the development of the missing translation algorithm into a GWT Java project and the corresponding Xpand transformation templates. The generated Promela code reflects the behaviour of the Google Chrome browser. Support for other browsers like Firefox and Internet Explorer is still missing and has to be implemented.

Clients are implemented in Promela as concurrent processes. Spin supports only 256 active processes which means we can only verify the system behaviour for 256 clients. It is therefore necessary to implement an alternative Promela model which uses one process for a regular client and one process which emulates the behavior of n other clients.

This paper combines model-driven development with formal verification. We have already implemented another case study which also uses this approach and provides a DSL for statecharts and the corresponding transformation templates [26]. The implementation of both case studies was time consuming and required expert knowledge in model checking. Therefore, we are currently developing a framework which assists DSL developers in the generation of domain-specific languages and the automated transformation into a model checker input language. First ideas regarding this framework are already published in [27].

References


