Implementing the WebSocket Protocol Based on Formal Modelling and Automated Code Generation

Kent Fagerland Simonsen, Lars Michael Kristensen

Implementing the WebSocket Protocol based on Formal Modelling and Automated Code Generation

Kent Inge Fagerland Simonsen[1,2] and Lars Michael Kristensen[1]

[1] Department of Computing, Bergen University College, Norway
Email: {lmkr,kifs}@hib.no
[2] DTU Compute, Technical University of Denmark, Denmark
Email: {kisi}@dtu.dk

Abstract. Model-based software engineering offers several attractive benefits for the implementation of protocols, including automated code generation for different platforms from design-level models. In earlier work, we have proposed a template-based approach using Coloured Petri Net formal models with pragmatic annotations for automated code generation of protocol software. The contribution of this paper is an application of the approach as implemented in the PetriCode tool to obtain protocol software implementing the IETF WebSocket protocol. This demonstrates the scalability of our approach to real protocols. Furthermore, we perform formal verification of the CPN model prior to code generation, and test the implementation for interoperability against the Autobahn WebSocket test-suite resulting in 97% and 99% success rate for the client and server implementation, respectively. The tests show that the cause of test failures were mostly due to local and trivial errors in newly written code-generation templates, and not related to the overall logical operation of the protocol as specified by the CPN model.

1 Introduction

The vast majority of software systems today can be characterised as concurrent and distributed systems as their operation inherently relies on protocols executed between independently scheduled software components and applications. The engineering of correct protocols can be a challenging task due to their complex behaviour which may result in subtle errors if not carefully designed. Furthermore, ensuring interoperability between independently made implementations is also challenging due to ambiguous protocol specifications. The use of formal modelling in combination with verification and model checking provides a prominent approach to the development of reliable protocol implementations.

Coloured Petri Nets (CPNs) [8] is formal language combining Petri Nets with a programming language to obtain a modelling language that scales to large systems. In CPNs, Petri Nets provide the primitives for modelling concurrency and synchronisation while the Standard ML programming language provides the primitives for modelling data and data manipulation. CPNs have

been successfully applied for the modelling and verification of many protocols, including Internet protocols such as the TCP, DCCP, and DYMO protocols [2, 11]. Formal modelling and verification have been useful in gaining insight into the operation of the protocols considered and have resulted in improved protocol specifications. However, earlier work has not fully leveraged the investment in modelling by also taking the step to automated code generation as a way to obtain an implementation of the protocol under consideration.

In earlier work [15], we have proposed the PetriCode approach and developed a supporting software tool [17] for automatically generating protocol implementations based on CPN models. The basic idea of the approach is to enforce particular modelling patterns and annotate the CPN models with code generation *pragmatics*. The pragmatics are bound to code generation templates and used to direct the model-to-text transformation that generates the protocol implementation. As part of earlier work, we have demonstrated the use of the PetriCode approach on small protocols. In addition, we have shown that our approach supports code generation for multiple platforms, and that it leads to code that is readable and also upwards and downwards compatible with other software [16].

The main contribution of this paper is to demonstrate that our approach and tool scale to support an industrial-sized protocol by automatically generating code for the WebSocket [5] protocol for the Groovy [7] platform. The WebSocket protocol is a relatively new protocol currently under development by the IETF. The WebSocket protocol makes it possible to upgrade an HTTP connection to an efficient message-based full-duplex connection. The WebSocket protocol address the performance problems of the HTTP protocol caused by the request-response interaction model and verbose headers. This is done by allowing HTTP to upgrade to a WebSocket connection in which a session is kept alive and messages may be transmitted in both directions freely with much lower overhead than with HTTP. WebSocket has already become a popular protocol for several web-based applications where bi-directional communication with low latency is needed such as games and media streaming services. The contributions of this paper include showing how we have been able to model the WebSocket protocol following the PetriCode modelling conventions, and to verify the model through state space exploration. Furthermore, we demonstrate in this paper that the generated code is interoperable with other WebSocket implementations, and we test the our implementation using the Autobahn WebSocket test-suite [18].

Outline. Section 2 presents the CPN model of the WebSocket protocol. In Sect. 3 we show how state space exploration was used to verify the operation of the model focusing on the proper establishment and termination of WebSocket connections. Section 4 describes the procedure to generate an implementation of the WebSocket protocol from the CPN model using the PetriCode tool. In Sect. 5, we present the results from testing the generated code by showing that it is interoperable with other WebSocket implementations and by employing the Autobahn WebSocket test-suite. Finally, in Sect. 6 we provide a discussion of related work, and sum up the conclusions and directions for future work. Due to space limitations, we refer to [8] for a detailed introduction to CPN concepts.

2 The CPN WebSocket Code Generation Model

The CPN model of the WebSocket protocol follows the structure imposed by our code generation approach, and consists of a set of modules hierarchically organised into three levels: the protocol systems level, the principal level, and the service level. In the following, we present representative parts of the CPN model which was constructed using CPN Tools [4].

Figure 1 shows the top-level module of the CPN model constituting the protocol system level. The protocol system consists of a Client and a Server principal as modelled by the two accordingly named *substitution transitions* drawn as rectangles with a double-lined border. These two substitution transitions are annotated with the ⟨⟨principal⟩⟩ code generation pragmatic to denote that they represent protocol principals. The Channel substitution transition annotated with the ⟨⟨channel⟩⟩ pragmatic represents the channel connecting the two principals. The two substitution transitions are connected by *places* (drawn as ellipses) modelling send and receive buffers for the client and server. The rectangular tags attached to the substitution transitions specify the name of the *submodule* which refines the compound behaviour represented by the substitution transition.

The Client principal level module is depicted in Fig. 2. It is the submodule associated with the Client substitution transition in Fig. 1. The principal level makes explicit the *services* offered by the principal by means of the ⟨⟨service⟩⟩ and ⟨⟨internal⟩⟩ pragmatics attached to substitution transitions. The ⟨⟨service⟩⟩ pragmatic is used to denote substitution transitions where the attached submodule represents a service that is intended to be used by the application employing the protocol. Substitution transitions annotated with ⟨⟨internal⟩⟩ represent services that are used internally in the protocol principal. It can be seen that the client has six external and two internal services. A principal level module also models the *internal state* of the principal via places annotated with the ⟨⟨state⟩⟩ pragmatic, and captures the life-cycle of the principal via places annotated with the ⟨⟨LCV⟩⟩ pragmatic. The life-cycle determines the possible orders in which the services can be invoked. Initially, the only ⟨⟨LCV⟩⟩-annotated place that contains a token is the READY place (top) which enables the OpenConnection service. After the OpenConnection service has completed, there will be a token on the OPEN place, and all the external services (except OpenConnection) will be enabled allowing the employing application to send and receive messages, send

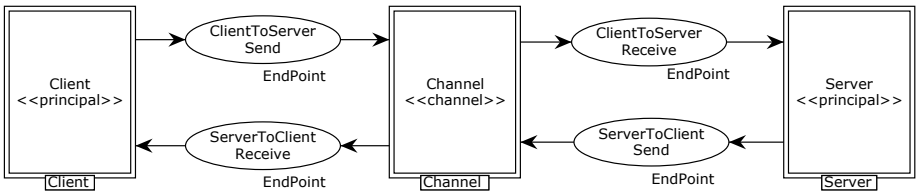

Fig. 1. The top level of the WebSocket protocol model

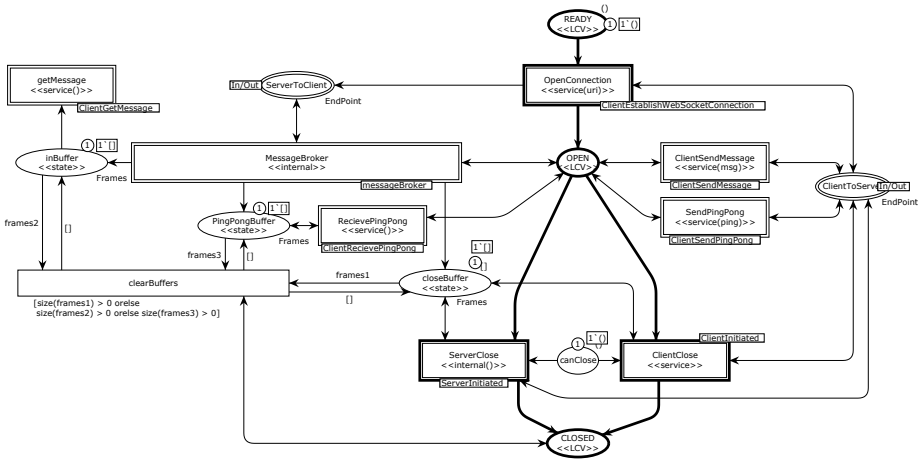

Fig. 2. The Client principal module

and receive ping and pong messages, and close the connection. The exchange of ping and pong messages provides a keep-alive mechanism in the protocol.

The MessageBroker module which is the submodule of the MessageBroker substitution transition is shown in Fig. 3. The MessageBroker is an example of an internal service. It is responsible for dispatching the incoming messages into the appropriate buffer represented in the module by places annotated with the ⟨⟨state⟩⟩ pragmatics. There is one such buffer for each of the message types: the inBuffer keeps text and binary messages, the pingpongBuffer keeps ping and pong messages, the closeBuffer keeps the closing messages while the fragments place keeps frames of messages that have not yet been completely received. The messages are dispatched by inspecting the type of the messages. The MessageBroker internal service is enabled when the WebSocket connection is in an OPEN state and the module captures the control flow in dispatching received messages as indicated by the places annotated with an ⟨⟨Id⟩⟩ pragmatic. The execution of the service starts at the transition ReceiveDataFrame. Then it enters a loop starting at place wait receive. At the transition receive a new frame is received. This is modelled as a single operation to keep the model at a high level of abstraction. This means that the details of actually receiving a message must be encoded in the code generation template associated with the ⟨⟨receive⟩⟩ pragmatic. If the frame is the last frame of a fragmented message, the entire message is reconstructed. Next, there is a branch in the model based on the Fin and OpCode fields in the frame. The message is dispatched to either the inBuffer (data), PingPongBuffer, closeBuffer, or nonFinal. After the message or frame has been dispatched, the branches merge before the next iteration.

The getMessage service shown in Fig. 4 returns the next message in the buffer. This service will be used to illustrate code generation in Sect. 4.

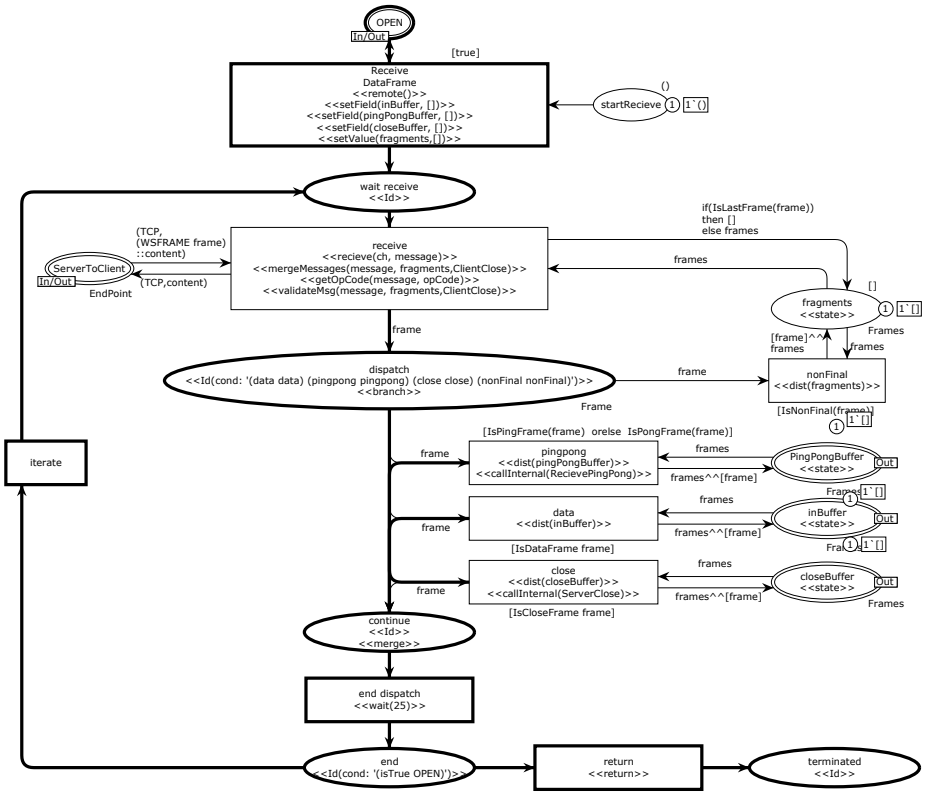

Fig. 3. The MessageBroker internal service

This is an example of a service with only a single transition. The transition is annotated with the $\langle\langle$service$\rangle\rangle$, $\langle\langle$getMessage$\rangle\rangle$ and $\langle\langle$return$\rangle\rangle$ pragmatics. This models that the service is first entered, then the $\langle\langle$getMessage$\rangle\rangle$ operation is performed, and the service terminates. The transition getMessage is only enabled when there is at least one message in the message buffer as modelled by the place inBuffer.

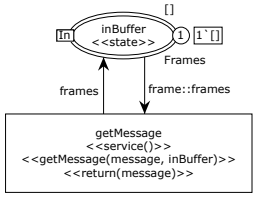

Fig. 4. The getMessage service

The complete CPN model consists of 19 modules. Each of the two principals have eight sub-modules which all correspond to the external and internal services in the protocol. In total, the model consists of 136 places and 84 transitions. This reflects the complexity of the protocol, but also the high-level nature of the model which has been important in keeping the number of elements manageable.

3 CPN WebSocket Model Verification

CPNs have a formal semantics which makes it possible to conduct model simulation and model checking (verification) prior to code generation. This is a major advantage of an approach based on a formal modelling language as this can be used to eliminate design errors prior to code generation and testing of the generated protocol implementation. CPN Tools used for construction of the WebSocket CPN model supports model checking of behavioural properties by means of *state space exploration*. The basic idea of state space exploration is to explore all the reachable states of the model to determine whether a model satisfies a given property or not. This means that state space exploration will exhaustively explore (test) all the possible executions of the CPN model. As the CPN model specifies the behaviour of both the client and the server, the state space exploration exercises the client against all the possible behaviours of the server and visa versa.

Our aim has been to apply state space exploration of the CPN model as a first test to eliminate possible errors in the logical specification of the WebSocket protocol. For this, we adopted a lightweight approach where we consider the following behavioural properties **P0**, **P1** and **P2** of the CPN model:

P0 From the initial state it is possible to reach states in which the WebSocket connection has been opened (i.e., both the client and the server are in the *open* state). In the model this means that the places names OPEN in the Client and the Server modules each have one token and none of the other places other places modelling the life-cycle of the principals have a token. It should be noted that we cannot establish that the WebSocket connection will eventually be opened since the server side may initiate a close before the client side is in the *open* state.

P1 All terminal states (i.e., states without enabled transitions) correspond to states in which the WebSocket connection has been properly closed (i.e., both the client and the server are in the *closed* state). In the model this means that the places named CLOSED in the Client and Server modules each have one token and that none of the other places modelling the life-cycle of the principals have a token.

P2 From any reachable state, it is always possible to reach a state in which the WebSocket connection has been properly closed. This means that independently of how messages are exchanged, it is always possible to properly close the WebSocket connection.

In order to check that in all terminal states both the client and the server are in the *closed* state, we wrote a simple query in the Standard ML language using functions that are built into CPN Tools. The query can be seen in Listing 1.1.

The functions `server_open` and `client_open` are predicates for the client and server that take a state as argument and return true if and only if the principal is in an open state, i.e., the place OPEN has one token, and all the other LCV places have no tokens. The functions `server_close` and `client_close`

Listing 1.1. The queries used to verify properties P0-P2.

```
fun client_open (n) = (State.Client'CLOSED 1 n) = [] andalso
 (State.Client'OPEN 1 n) = [()] andalso (State.Client'READY 1 n) = [];

fun server_open (n) =
 (State.Server'CLOSED 1 n) = [] andalso (State.Server'OPEN 1 n) = [()] andalso
 (State.Server'Idle 1 n) = []   andalso (State.Server'READY 1 n) = [];

fun client_closed (n) = (State.Client'CLOSED 1 n) = [()]
 andalso (State.Client'OPEN 1 n) = [] andalso (State.Client'READY 1 n) = [];

fun server_pred (n) =
 (State.Server'CLOSED 1 n) = [()] andalso (State.Server'OPEN 1 n) = [] andalso
 (State.Server'Idle 1 n) = []     andalso (State.Server'READY 1 n) = [];

fun IsProperOpen(n) = server_open(n) andalso client_open(n);
fun IsProperClosed(n) = server_closed(n) andalso client_closed(n);

PredAllNodes(IsProperOpen) <> []              (* property P0 *)
List.all IsProperClosed (ListTerminalStates()); (* property P1 *)
HomeSpace(ListTerminalStates());                (* property P2 *)
```

Table 1. Results of verification of the WebSocket CPN model

Client Messages	Server messages	Nodes	Arcs	Time (secs)	Terminal states
yes	no	2747	9,544	1	2
no	yes	2867	9,956	2	2
yes	yes	39189	177,238	246	4

are similar for the case of closed. The predicates are used to obtain the predicates `isProperOpen` and `isProperClosed` that characterises properly open and closed states, respectively. The property P0 is checked using the query function `PredAllNodes` which returns all states satisfying a given predicate (in this case `IsProperOpen`). It is then checked whether the resulting list of states is non-empty. For establishing P1, we check that all terminal states in the state space which are returned by the built-in query function `ListTerminalStates` satisfies the `IsProperClosed` predicate. Finally, P2 is checked using the query function `HomeSpace` which checks if the list of nodes provided constitute a *home space*, i.e, constitute a set of states where at least one of the states can always be reached.

Table 1 summarises the results from the verification. We have considered three possible configurations of the model. One where the client sends data to the server; one where the server sends data to the client; and one where both the client and the server sends data. The table lists the number of Nodes and Arcs in the state space, the amount of Time used to generate the state space, and the number of Terminal States. For all configurations, we were able to establish the properties P0, P1 and P2 which provides confidence in the correctness of the model. During the verification process, several minor modelling errors were identified and fixed. For example, this lead to the inclusion of the clearBuffers transition (see Fig. 2) which was added to properly clean up message buffers and reduce the number of terminal states.

The major drawback with state space exploration techniques is the state explosion problem which means that the state space in many cases grows too large to be handled with the available computing power. It is interesting to observe that the size of the state space for the model described in Sect. 2 is relatively small for the configurations considered. This shows how our modelling approach makes it possible to construct models at a high-level of abstraction so that it is feasible to fully verify even industrial-sized protocols.

4 Automated Code Generation

In this section we describe the code generation process for the WebSocket protocol targeting the Groovy platform and illustrate it with examples of code generation templates and code snippets. Groovy is also the implementation language of the PetriCode tool [17] and has been chosen because it has several features that makes it easy to implement and debug templates including dynamic typing, closures and iterators.

The automatic code generation process, as implemented in the PetriCode tool, starts with a CPN model annotated with pragmatics. The model is first transformed into an intermediary representation in the form of an *abstract template tree* (ATT). The ATT reflects the hierarchical structure of the CPN model down to the service level. On the service level, the ATT contains *blocks* that are derived from the control flow path specified by the ⟨⟨id⟩⟩ pragmatics of the service level modules. The next step in the code generation process is to traverse the ATT and emit code for each node by applying code generation templates bound to the pragmatics of a node. Pragmatics are bound to templates using *template bindings* which are defined in a domain specific language (DSL). When this is done, the code is stitched together using special markers in the generated code. The details are described in [15, 17]. Our approach makes it possible to produce code for several platforms and programming languages. This is achieved by using different sets of code generating templates and binding them as appropriate to code generation pragmatics through the use of the DSL.

When the code generator, on its traversal through the ATT, encounters a node annotated with a ⟨⟨principal⟩⟩ pragmatic it executes the associated templates which, in the Groovy platform, defines a class. Then the traversal continues to the child nodes of the principal. When the generation traverses child nodes of a principal and encounters a node containing a ⟨⟨service⟩⟩ or ⟨⟨internal⟩⟩ pragmatic it executes the service template. The code generation for the principal is completed by replacing a special tag, %%yield%% with the result for the service template for all underlying services. The generated code of the client with declaration and method bodies omitted is shown in Listing 1.2. As can be seen when comparing with Fig 2, there is one method defined for each external and internal service. This comprises the API for the WebSocket client implementation with all the callable methods and their signature.

The template for the ⟨⟨service⟩⟩ pragmatic is shown in Listing 1.3. Lines 1-2 define a new method and its signature. The lines 3-12 set up preconditions (if

Listing 1.2. The generated code for the services in the client.

```
1  class Client {
2  ...
3  def MessageBroker(){ ... }
4  def ServerClose(){ ... }
5  def OpenConnection(uri){ ... }
6  def ClientSendMessage(msg){ ... }
7  def ReceivePingPong(){ ... }
8  def SendPingPong(ping){ ... }
9  def ClientClose(){ ... }
10 def getMessage(){ ... }
11 }
```

Listing 1.3. The template bound to the ⟨⟨service⟩⟩ pragmatic

```
1  def ${name}(${binding.getVariables()
2              .containsKey("params") ? params.join(", ") : ""}){
3  <%
4    if(binding.variables.containsKey('pre_conds')){
5      for(pre_cond in pre_conds){
6        %>if(!$pre_cond) throw
7            new Exception('unfulfilled precondition: $pre_cond')
8        <%
9        if(!pre_sets.contains("$pre_cond")){%>$pre_cond = false<%}
10     }
11   }
12 %>
13   %%yield_declarations%%
14   %%yield%%
15   <%if(binding.variables.containsKey('post_sets')){
16     for(post_set in post_sets){
17       %>$post_set = true<%
18      }
19  }%>
20 }
```

applicable) based on the manipulation of places at the service level that are annotated with the ⟨⟨LCV⟩⟩ pragmatic. The next two lines are place-holder tags that show where declarations and the method body will be inserted respectively. Finally, post-conditions are set and the method body ends in line 20.

Listing 1.4 shows the template for the ⟨⟨getMessage⟩⟩ pragmatic used on the transition in Fig. 4. The template takes two parameters. The name of variable to set the next message to, and the name of the buffer to retrieve the next message from. First, the template checks to see if the buffer is not empty. If it is not empty, the first message is retrieved from the buffer. Then the payload is translated into a `String` or a byte array depending on the message type and the variable given in the first parameter to the pragmatic is set to the payload of the message. If the buffer is empty the variable given in the first parameter to the pragmatic is set to *null*.

The generated code for the getMessage service is shown in 1.5. Lines 1-4 and 21 are generated by the service template. The rest of the code, except from the return line, follows the template for ⟨⟨getMessage⟩⟩ where the first and second

Listing 1.4. The template for the ⟨⟨getMessage⟩⟩ pragmatic.

```
if(${params[1]} != null &&  ${params[1]}.size() > 0){
  ${params[0]} = ${params[1]}.remove(0)
  byte[] bArr = new byte[${params[0]}.payLoad.size()]
  for(int i = 0; i < bArr.length; i++){
    bArr[i] = ${params[0]}.payLoad.get(i)
  }
  if(${params[0]}.opCode == 1){
    ${params[0]} = new String(bArr)
  }else if(${params[0]}.opCode == 2) {
    ${params[0]} = bArr
  }
}else{
  ${params[0]} = null
}
%%VARS: ${params[0]}, ${params[1]}%%
```

Listing 1.5. The generated code for the getMessage service in the client.

```
def getMessage(){
  /*vars: [__TOKEN__:, message:]*/
  def __TOKEN__
  def message
  //getMessage
  if(inBuffer != null && inBuffer.size() > 0){
    message = inBuffer.remove(0)
    byte[] bArr = new byte[message.payLoad.size()]
    for(int i = 0; i < bArr.length; i++){
      bArr[i] = message.payLoad.get(i)
    }
    if(message.opCode == 1){
      message = new String(bArr)
     }else if(message.opCode == 2) {
       message = bArr
     }
  }else{
    message = null
  }
  return message
}
```

parameters have been replaced with inBuffer and message respectively since those are the two parameters given to the pragmatic in Fig. 4.

In order to generate code for the WebSocket protocol, we reused 10 templates from the library of templates provided by PetriCode. In addition, 22 new templates were needed, including two templates that override existing templates. New templates were needed because the WebSocket protocol has many features we have not encountered with earlier examples, such as receiving and interpreting binary messages, and validating handshakes and frames.

5 Testing the Generated WebSocket Implementation

We validated the operation and interoperability of the generated code in two ways. First, we created test drivers for the generated WebSocket implementation to connect to the example chat server and client [1] that comes with the GlassFish

Listing 1.6. The code for the client runner.

```
def client = new Client()
client.OpenConnection(new URI("ws://localhost:31337/chat/websocket"))
def t = Thread.start {
  while(true){
    def msg =  client.getMessage()
    if(msg) println "RECEIVED: $msg"
    Thread.sleep(1000)
}}
client.ClientSendMessage("${args[0]} joined")
BufferedReader br = new BufferedReader(new InputStreamReader(System.in))
while(true){
  print "#: "
  def msg = br.readLine()
  if(msg == "#quit"){
    client.ClientClose()
    try{ Thread.wait(1000) }
    catch(Exception ex){  }
    System.exit(0)
  } else if(msg == "#close"){
    client.ClientClose()
  }else{
    client.ClientSendMessage("${args[0]}: $msg")
  }
}
```

Application Server [14]. Secondly, we submitted the generated implementation to the Autobahn Testsuite [18] version 0.5.5[1].

Chat Application. The code for the chat client using the generated API (cf. Listing 1.2) is shown in Listing 1.6. The chat client uses the generated WebSocket protocol as an API given the signatures of the principal level. The client begins by creating a `Client` object from the generated code and opens a WebSocket connection to the server. Then, a thread is started to receive messages which polls the client object for new messages and prints any received messages to the console. After the message receiving thread is started, the client sends a message notifying the server that the client has joined the chat. Finally, the client enters an infinite loop that listens to the console for messages and sends any messages to the server. The server is implemented in a similar way using the generated `Server` class as the server-side WebSocket API.

Listing 1.6 demonstrates that our approach is upwards compatible, i.e., that the services of the generated code can easily be used by third party software. A key feature that provides this is that we include the API in the model as the services at the principal level of the CPN model.

Figure 5 shows the chat client (upper right) and server (lower right) running together with the web-based chat client from [1]. The web-based client has only been modified to connect to the server using the generated API by changing a hard-coded server address. We also tested that the chat client is able to connect and communicate with a chat-server from [1].

[1] The test results can be seen at http://t.k1s.org/wsreport

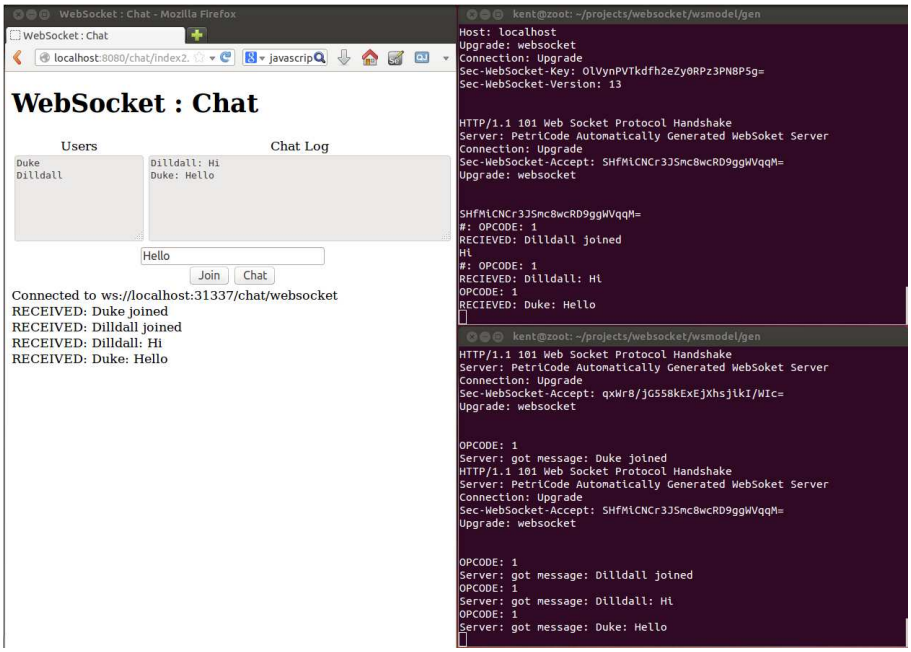

Fig. 5. Chat server and client using the generated API (right) and a web-based chat client connected to the same server (left)

Autobahn Test-suite. The Autobahn WebSocket test-suite provides comprehensive validation of server and client implementations of the WebSocket protocol. The test-suite has been used by several high-profile projects to develop and validate WebSocket implementations including the Firefox and Jetty projects. When running the Autobahn test-suite several problems with the implementation were discovered. Most of the problems were simple oversights in the code generation templates that were easily fixed once they were identified. An example of the trivial problems that were not evident when running the chat application was that the HTTP header lines were terminated with LF instead of the mandated CRLF. However, one change to the CPN model was necessary. This was related to fragmented messages where we added a buffer for temporarily storing frames of unfinished messages and a transition to distributing non-final frames. This was necessary because a WebSocket endpoint should be able to handle control messages intermingled with fragmented messages. The new elements, which can be seen in Fig. 3, are the place fragments, the transition nonFinal, and the arcs connected to those two elements.

A summary of the result for the final Autobahn tests can be seen in Table 2. The Autobahn test suite contains 301 tests cases for the client and server. For the client, 10 test cases fail and for the server, 4 test cases fail. The extra test cases that fail on the client concern performance with large messages. The test cases that fail for both the server and client are UTF-8 parser errors. This is because

the Java implementation of UTF-8 parsers is more lenient than the Autobahn test-suite expects. Therefore, we had to create our UTF-8 validator which fails to identify some UTF-8 errors.

Table 2. Results for the Autobahn tests

Tests	Server Passed	Client Passed
1. Framing (text and binary messages)	16/16	16/16
2. Pings/Pongs	11/11	11/11
3. Reserved bits	7/7	7/7
4. Opcodes	10/10	10/10
5. Fragmentation	20/20	20/20
6. UTF-8 handling	137/141	137/141
7. Close handling	38/38	38/38
9. Limits/Performance	54/54	48/54
10. Auto-Fragmentation	1/1	1/1

6 Conclusions and Related Work

In this paper we have shown that the PetriCode code generation approach can be applied to industrial sized protocols as exemplified by the WebSocket protocol. Obtaining the implementation was achieved with limited effort even though quite a few new templates were created. We have found that the template provides an effective way to force the code to be modular. This means that the templates can be developed in a certain degree of isolation, giving the developer the opportunity to concentrate on getting a single template right at a time. Therefore, even though many templates are created for only a single protocol, this is an efficient way to prototype protocols based on a CPN model.

Compared to previous examples, the WebSocket model had many more services. This makes the principal level somewhat harder to read and suggests that some kind of mechanism of grouping the services in several layers might be advantageous. At the service level, the models are approximately the same size as in previous examples. The readability of the service level modules can also be controlled by offloading behaviour to pragmatics such as we do for the $\langle\langle$receive$\rangle\rangle$ pragmatic in the message broker. All in all, the WebSocket model shows that we can make code generation models for real protocols without necessarily losing descriptiveness.

We have also showed that the code generation model can be verified by state space exploration. This highlights a major advantage of using CPN models which are directly executable. This allows us to perform analysis on high-level models and thereby keep the state spaces small. Although the verification presented only considers basic connection establishment and termination properties, other more elaborate properties including liveness properties can be checked using similar techniques. We are also working on using the sweep-line method, and advanced state space exploration method, to alleviate the state space explosion problem.

Finally, we have validated the automatically generated WebSocket implementation both by applying it to a well-known example in the form of the example chat application which is distributed with the GlassFish Application server and also by using the Autobahn test-suite which thoroughly tests most aspects of WebSocket protocol implementations.

To the best of our knowledge there does not exist any examples of using model-based techniques for generating an implementation of the WebSocket protocol in the literature. However, there exists a few examples for other industrial sized protocols. In [12] PP-CPNs, another class of Petri Nets, was used to generate code for the DYMO routing protocol for the Erlang platform. In our approach, we have a more flexible code generation approach through pragmatics that allows us to create new custom templates for new situations. Another approach to code generation is exemplified by the RENEW tool [13]. RENEW uses a simulation based approach where the implementation is a simulation of the underlying Petri Net. The direct use of simulation code makes it harder to meaningfully inspect the generated programs.

Other formalisms such the Specification and Description Language (SDL) has also been used as a starting point for code generation. For example, an early warning system for earthquake was developed using SDL in combination with UML [6, 3]. Both simulation and prototype code were generated using C++ as the target language. Our approach differs from the above mentioned approaches by the flexibility in abstraction level because of our pragmatics, by being platform independent by simply substituting templates and by the fact that we model the API explicitly at the service level and thereby easy interoperability with third-party software. Our approach also allows us to model the service interface, which is not available to the same degree in PP-CPNs and Renew. Another approach to generating software for reactive systems from UML models is the SPACE method [10] and its tool Arctis. This approach employs collaborations to compose services. The collaborations are then transformed into state machines that are executed together with Java snippets which are bound to actions of the collaborations. This approach relies on the state machines to either be translated to code or executed directly by some other tool. This is in contrast to our approach where we generate code directly instead of going through other formalisms and tools. MACE [9] is a textual state-transition language that is used to create distributes systems. It uses a compiler to compile the textual state-transition language into C++ code. This means that MACE is not as platform independent as our template-based approach is. In MetaEdit+ [19], models are mapped to some underlying formalism on which analysis is then performed. This tends to produce larger state space sizes compared with our approaches where the model is executable and allows verification at a high level of abstraction.

In the future we will apply more advanced state-space techniques, such as the sweep-line method, in order to do a more through verification of the model with larger and more complex configurations. Furthermore, we will investigate how errors in code generated by PetriCode can be traced back to the relevant pragmatics and model elements.

References

1. A. Gupta. *Chat Sever using WebSocket in GlassFish 4.* `https://blogs.oracle.com/arungupta/entry/chat_sever_using_websocket_totd`.
2. J. Billington, G.E. Gallasch, and B. Han. A Coloured Petri Net Approach to Protocol Verification. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 210–290. Springer, 2004.
3. M. Brumbulli and J. Fischer. SDL Code Generation for Network Simulators. In *Proc of SAM '10*, volume 6598 of *LNCS*, pages 144–155. Springer, 2011.
4. CPN Tools. *Home Page.* `http://cpntools.org/`.
5. I. Fette and A. Melnikov. The websocket protocol. 2011. `http://tools.ietf.org/html/rfc6455`.
6. J. Fischer, F. Kühnlenz, K. Ahrens, and I. Eveslage. Model-based Development of Self-organizing Earthquake Early Warning Systems. In *Proceedings of MATHMOD*, 2009.
7. Groovy. *Project Web Site.* `http://groovy.codehaus.org`.
8. K. Jensen, L.M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 9(3-4):213–254, 2007.
9. C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M Vahdat. Mace: language support for building distributed systems. In *ACM SIGPLAN Notices*, volume 42, pages 179–188. ACM, 2007.
10. Frank Alexander Kraemer, Rolv Bræk, and Peter Herrmann. Compositional Service Engineering with Arctis. *Telektronikk*, 105(2009.1), 2009.
11. L.M. Kristensen and K.I.F. Simonsen. Applications of Coloured Petri Nets for Functional Validation of Protocol Designs. In *Transactions on Petri Nets and Other Models of Concurrency VII*, volume 7480 of *LNCS*, pages 56–115. Springer, 2013.
12. L.M. Kristensen and M. Westergaard. Automatic Structure-Based Code Generation from Coloured Petri Nets: A Proof of Concept. In *Proc. of FMICS'10*, volume 6371 of *LNCS*, pages 215–230. Springer, 2010.
13. O. Kummer et al. An Extensible Editor and Simulation Engine for Petri Nets: Renew. In *Proc. of ICATPN '04*, volume 3099 of *LNCS*, pages 484–493. Springer, 2004.
14. Oracle Corporation. *GlassFish Application Server.* `https://glassfish.java.net/`.
15. K. I. F. Simonsen, L. M. Kristensen, and E. Kindler. Generating Protocol Software from CPN Models Annotated with Pragmatics. In *Formal Methods: Foundations and Applications*, volume 8195 of *LNCS*, pages 227–242. Springer, 2013.
16. K.I.F. Simonsen. An Evaluation of Automated Code Generation with the PetriCode Approach. In *Submitted to: PNSE'14*.
17. K.I.F. Simonsen. PetriCode: A Tool for Template-based Code Generation from CPN Models. In *SEFM 2013 Collocated Workshops: BEAT2, WS-FMDS, FM-RAIL-Bok, MoKMaSD, and OpenCert*, volume 8368 of *LNCS*, pages 151–166. Springer, 2014.
18. Tavendo GmbH. *Autobahn—Testsuite.* `http://autobahn.ws/testsuite/`.
19. Juha-Pekka Tolvanen. Metaedit+: domain-specific modeling for full code generation demonstrated. In *proc. of OOPSLA'04*, pages 39–40. ACM, 2004.