An Evaluation of Automated Code Generation with the PetriCode Approach

Kent Inge Fagerland Simonsen[1,2]

[1] Department of Computing, Bergen University College, Norway
Email: kifs@hib.no
[2] DTU Compute, Technical University of Denmark, Denmark

Abstract. Automated code generation is an important element of model driven development methodologies. We have previously proposed an approach for code generation based on Coloured Petri Net models annotated with textual pragmatics for the network protocol domain. In this paper, we present and evaluate three important properties of our approach: platform independence, code integratability, and code readability. The evaluation shows that our approach can generate code for a wide range of platforms which is integratable and readable.

1 Introduction

Coloured Petri Nets (CPNs) [5] is a general purpose formal modelling language for concurrent systems based on Petri Nets and the Standard ML programming language. CPNs and CPN Tools have been widely used to model and validate network protocol models [6]. In previous works [14], we have proposed an approach to automatically generate network protocol implementations based on a subclass of CPN models. We have implemented the approach in the PetriCode tool [13]. In this approach, CPN models are annotated with syntactical annotations called *pragmatics* that guide the code generation process and have no other impact on the CPN model. Code is then generated based on the pragmatics and code generation templates that are bound to each pragmatic through template bindings. This paper presents an evaluation of the PetriCode code generation approach and tool.

The four main objectives of our approach are: platform independence, code integratability, code readability, and verifiability of the CPN models. The contribution of this paper is an evaluation of the first three of these objectives. In this study, we used the PetriCode [13] tool to evaluate our code generation approach. Platform independence, i.e., the ability to generate code for several platforms, is an important feature of our approach. For the purposes of this study, a platform is a programming language and adjoining APIs. Being able to generate protocol implementations for several platforms allows us to automatically obtain implementations for many platforms based on the same underlying CPN model. Platform independence also contributes to making sure that implementations for different platforms are interoperable. Another aspect is to have

models that are independent of platform specific details. Integrateability, i.e., the ability to integrate generated code with third-party code, is important since the protocols must be used by other software components written for the platform under consideration (upwards integratability). It is also important to be able to support different underlying libraries so that the generated code can be referred to by other components (downwards integrateability). Readability is important in order to gain confidence that the implementation of a protocol is as expected. While being able to verify the formal protocol models also contribute to this, inspecting and reviewing the final code further strengthens confidence in the correctness of the implementations. The ability to manually inspect the generated code is useful since, in our approach, we only verify the model which is not sufficient to remove local errors in the code.

The rest of this paper is organized as follows. Section 2 describes the example protocol used throughout this paper, and illustrates the code generation process for the Groovy platform. Section 3 evaluates platform independence by considering the Java, Clojure, and Python platforms. Section 4 evaluates integrateability, and Section 5 evaluates readability of the code generated by our approach. Section 6 presents related work, sums up conclusions and discusses directions for future work. Due to space limitation we provide little on CPNs and Petri Nets. The reader is referred to [5] for details on CPNs and Petri Nets. The PetriCode tool as well as the model, template and bindings used in this paper are available at [10].

2 Example and Code Generation

In this section, we present an example CPN model which is an extension of the protocol model we have used in previous work [14]. The example allows us to introduce the concepts and foundations of our approach and the PetriCode tool as well PA-CPNs [14], the CPN sub-class that has been defined for this approach. The example is a well established and used in the literature to describe CPNs [5]. It is also a natural extension of the example we have been using in previous works [14].

This example is a simple framing protocol which is tolerant to packet loss, reordering and allows a limited number of retransmissions. The top level of the CPN model is shown in Fig. 1. The model consists of three sub-modules. Sender and Receiver represents each of the principal actors of the protocol, and Channel connects the two principals.

The protocol uses sequence numbers and a flag to indicate the last message of a frame. After a frame has been sent, the receiver, if it receives the frame, sends an acknowledgement consisting of the sequence number of the frame expected next. If the acknowledgement is not received, the sender will retransmit the frame until an acknowledgement is received or the protocol fails sending the message.

In the Sender module, shown in Fig.2, there are two sub-modules. The send sub-module is annotated with a $\langle\langle$service$\rangle\rangle$[1] pragmatic and represents a service

[1] Pragmatics in the model and in the text are by convention written inside $\langle\langle\rangle\rangle$.

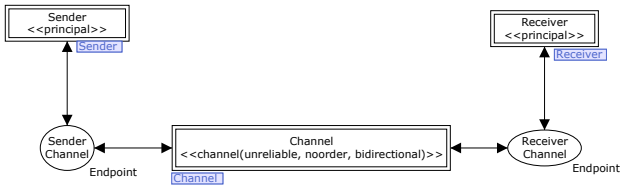

Fig. 1: The protocol system level

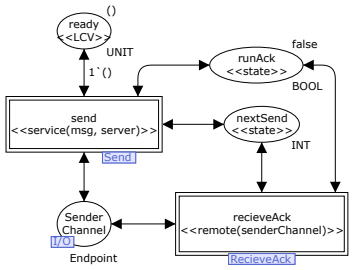

Fig. 2: The Sender principal module

provided by this principal for sending a message. The other substitution transition receiveAck, annotated with an ⟨⟨internal⟩⟩ pragmatic, represents an internal service which is to be invoked by another service of the principal. In this example, the receiveAck service is invoked from the send service.

The Sender module also contains two places, runAck and nextSend, annotated with a ⟨⟨state⟩⟩ pragmatic which contains shared data between the two services. The ready place, annotated with a ⟨⟨LCV⟩⟩ pragmatic, is used to model the life-cycle of the Sender principal and makes sure that only a single message is sent at a time.

The send service, shown in Fig. 3, starts at the transition send which opens the channel, initializes the content of the message to be sent and the sequence number. Also, at this transition, the receiveAck internal service is started by placing a token with the colour true at the ⟨⟨state⟩⟩ place runAck. The service continues from send to enter a loop at the start place. Inside the loop, the sendFrame transition retrieves the next frame to be sent based on the sequence number of the frame which is matched against the sequence number incoming from the place start. The limit place is updated with the sequence number of the current frame, and the number of times the frame has been retransmitted. Then, the current frame is sent. Due to the ⟨⟨wait⟩⟩ pragmatic at the sendFrame transition, the system waits in order to allow acknowledgements to be received. The loop ends at place frameSent. If a token is present on the place frameSent the loop will either continue with the transition nextFrame firing or end by firing the return transition. At the return transition, state places and the channel are cleared and

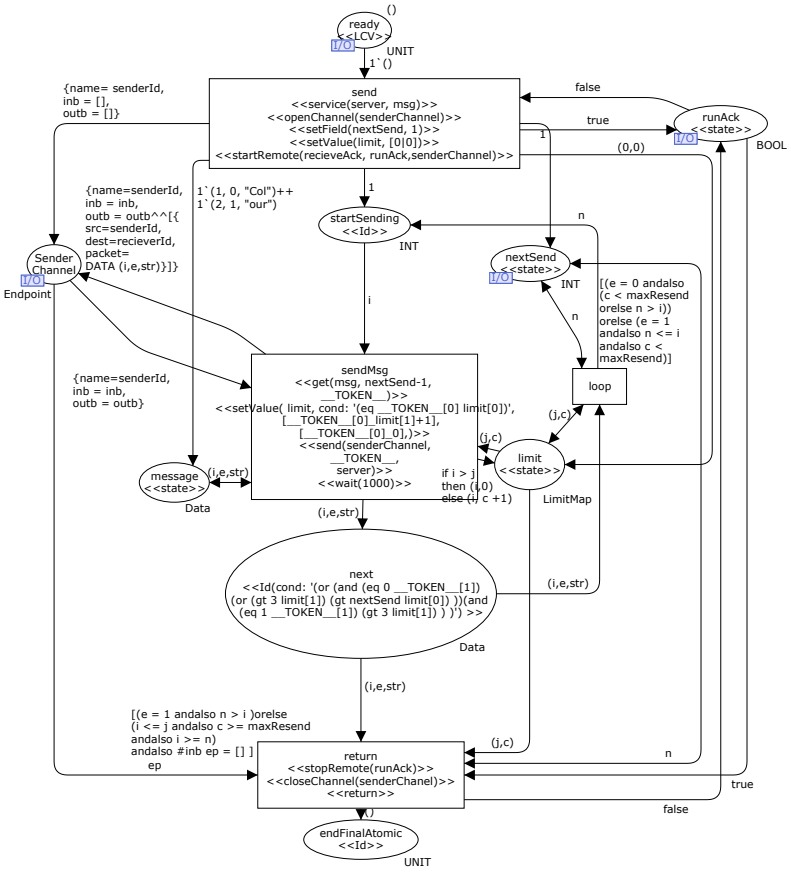

Fig. 3: The Send service module

the service terminates. In the model, we have not shown the pragmatics that can be automatically derived from the CPN model structure, see [14] for details.

The code generation approach is template-based and uses pragmatics to guide the code generation in two ways. The first way is by having structural pragmatics define the principals, services, and control-flow path within each service. The ⟨⟨principal⟩⟩, ⟨⟨service⟩⟩, and ⟨⟨Id⟩⟩ pragmatics in Figs. 1-3. The second way is to define the operations that should occur at each transition. The pragmatics are described in a domain specific language (DSL) and can often be derived from the CPN model structure. Structural pragmatics are used to generate the Abstract Template Tree (ATT), an intermediary representation of the pragmatics annotated CPN model. Each node in the ATT has pragmatics attached. Pragmatics are bound to code generation templates by template bindings. The generation

Listing 1: The Groovy template for ⟨⟨service⟩⟩ (left) and for ⟨⟨send⟩⟩ (right).

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

```
def bos = new ByteArrayOutputStream()
def oos = new ObjectOutputStream(bos)
oos.writeObject(${params[1]})
_msg_ = bos.toByteArray()
DatagramPacket pack =
  new DatagramPacket(_msg_, _msg_.length,
  InetAddress.getByName(${params[2]}.host),
  ${params[2]}.port)
${params[0]}.send pack
%%VARS:_msg_%%
```

uses these bindings to generate code for each pragmatic at each ATT node. Finally, the code is stitched together using special tags in the templates.

In order to give an overview of the code generation process, we use two templates as examples. The templates are the template for the ⟨⟨service⟩⟩ pragmatic (Listing 1 (left)) and the ⟨⟨send⟩⟩ pragmatic (Listing 1 (right)).

The service template for the Groovy platform is shown in Listing 1 (left). The first line of the template creates the signature of a method what will implement the service. Lines 4 to 10 iterates over preconditions to the ⟨⟨service⟩⟩. Each precondition is checked to make sure that the service may execute. In lines 11-12 two special tags %%yield%% and %%yield_declarations%% indicates the places where the method body and the declarations will be inserted from nodes coming from the sub-nodes in the ATT.

The template for ⟨⟨send⟩⟩ is shown in Listing 1 (right). The template first creates a byte array from the data to be sent and then creates an appropriate data packet and, finally, sends the datagram packet. The template uses UDP as the underlying transport protocol, which is why the packet is created in the form of a `DatagramPacket`.

The Groovy code shown here provides a baseline implementation for the protocol. In the next section we show ho we can generate code from the same model for three other platforms.

3 Evaluating Platform Independence

In order to demonstrate the platform independence of our approach, we have generated code for the Java, Clojure and Python platforms in addition to the Groovy platform. The platforms have been chosen in order to cover three main programming languages and paradigms. Java is an imperative and object oriented programming language. Clojure is a Lisp dialect for the Java Virtual Machine (JVM). It is a functional language, however it is able to utilize Java objects

and the Java API. Python is a multi-paradigm language and, as the only language in this survey, does not rely on the JVM. Python also uses significant white-spaces which makes Python unique in this evaluation in both respects. For each of the platforms, we show selected templates corresponding to the ones shown for the Groovy platform in Sect. 2. In addition, we show an exerpt of the generated code for the Java platform since this was used as part of the evaluation of readability presented in Sect. 5

Listing 2: The Java template for ⟨⟨service⟩⟩ (left) and for ⟨⟨send⟩⟩ (right).

```
public Object ${name}(<%
def paramsVal = ""
def params2 = []
if(binding.getVariables()
   .containsKey("params")){
  params.each{
    if(it.trim() != "")
      params2 << "Object $it"
  }
  paramsVal = params2.join(", ")
}%>$paramsVal) throws Exception {
<%if(binding.variables
    .containsKey('pre_conds')){
 for(pre_cond in pre_conds){
 %>if(!$pre_cond)
  throw new RuntimeException("...");
  <%if(!pre_sets
      .contains("$pre_cond"))
    {%>$pre_cond=false;<%}
 }}%>
  %%yield_declarations%%
  %%yield%% }
```

```
1  ByteArrayOutputStream bos = new
2      ByteArrayOutputStream();
3  ObjectOutputStream oos = new
4        ObjectOutputStream(bos);
5  oos.writeObject(${params[1]});
6  byte[] _msg_ = bos.toByteArray();
7  DatagramPacket pack = new
8    DatagramPacket(_msg_, _msg_.length,
9    InetAddress.getByName((String)
10   ((Map)${params[2]}).get("host")),
11   (Integer) ((Map)${params[2]})
12   .get("port"));
13 ((DatagramSocket)${params[0]})
14 .send(pack);
```

The Java Platform. The ⟨⟨service⟩⟩ template for the Java platform is shown in Listing 2 (left). The main difference from the Groovy service template is that, in the first line, the return type and visibility protection is explicit.

The ⟨⟨send⟩⟩ template (see Listing 2 (right)) is similar to the Groovy ⟨⟨send⟩⟩ template. The differences are mainly caused by the fact that Java is explicitly typed and, at times, requires explicit casts.

Excerpts of the Java code for the Sender principal is shown in Listing 3. The first part is generated from the service template. Lines 1-5 are generated by the ⟨⟨service⟩⟩ template (Listing 2 (left)) and lines 10-17 are generated by the ⟨⟨send⟩⟩ template (Listing 2 (right)).

The Clojure Platform. The Clojure ⟨⟨service⟩⟩ template is shown in Listing 4 (left). It begins by defining a function with the name set to the name parameter. Then it creates a vector which holds incoming variables. Finally, it yields for declarations and the body of the function.

The networking templates for Clojure uses the Java networking API and the ⟨⟨send⟩⟩ template (see Listing 4 (right)) and is therefore reminiscent of Groovy

Listing 3: The Java code for the send service.

```
1  public Object send(Object msg, Object  server) throws
2        Exception { /*[msg,  server]*/ /*[Object msg, Object  server]*/
3    if(!ready) throw new RuntimeException(
4                "unfulfilled precondition: ready");
5    ready = false;
6    ...
7    __LOOP_VAR__ = true;
8    do{
9      ...
10     ByteArrayOutputStream bos = new ByteArrayOutputStream();
11     ObjectOutputStream oos = new ObjectOutputStream(bos);
12     oos.writeObject(__TOKEN__);
13     byte[] _msg_ = bos.toByteArray();
14     DatagramPacket pack = new DatagramPacket(_msg_, _msg_.length,
15         InetAddress.getByName((String)((Map)
16         server).get("host")),(Integer) ((Map)server).get("port"));
17     ((DatagramSocket)senderChannel).send(pack);
18     ...
19   }while(__LOOP_VAR__);
20   ...
21 }
```

Listing 4: The Clojure template for ⟨⟨service⟩⟩ (left) and for ⟨⟨send⟩⟩ (right).

```
(defn ${name} <%
def paramsVal = ""
def params2 = []
if(binding.getVariables().
 containsKey("params")){
 params.each{
   if(it.trim() != "") params2 << "$it"
 }
 paramsVal = params2.join(", ")
%>[$paramsVal]<%}%>
(%%yield_declarations%%
%%yield%%))
```

```
(def bos (ByteArrayOutputStream.))
(.writeObject
 (ObjectOutputStream. bos)
 @${params[1]})
(def _msg_ (.toByteArray bos))
(.send  ${params[0]}
  (DatagramPacket.
    _msg_ (alength _msg_)
    (InetAddress/getByName
    (.get ${params[2]} "host") )
    (.get ${params[2]} "port"))
)
```

and Java templates. First, the message is converted into a byte array using `java.io` streams. Then a data packet is constructed and sent using the socket given as a parameter.

The Python Platform. The Python ⟨⟨service⟩⟩ template is shown in Listing 5 (left). The template defines the method in line 2 and adds parameters, given by the template variable paramsVal in line 10. Finally, the template yields for declarations and the method body in lines 12-13.

The Python template for ⟨⟨send⟩⟩ is shown in Listing 5(right). The data using Python is a simple call to the sendto function of a socket given as params[0] with the serialized data given in params[1] and the host and port from params[2] in a tuple.

Discussion. The examples above demonstrate that our approach allows us to generate code for several platforms by providing a selection of templates for

each platform. The platforms considered, spanning several popular paradigms, gives us confidence that our approach and tool can also be applied to generate code for many other platforms. Furthermore, we are able to generate the code for each of the platforms using the same model with the same annotations and the same code generator while only varying the code generation templates and the mappings between the pragmatics and mappings between pragmatics and code templates.

Adapting the Groovy templates to Java was, for the most part simple since the two languages are similar in several respects. However, whereas Groovy is optionally typed, Java is statically typed and requires all variables to be typed or to be cast to specific types when accessing methods. Fulfilling Java's requirements for explicit types requires functionality from PetriCode so that the templates are aware of the type of variables.

Clojure is a functional language with a different control flow from languages such as Java. The main issue, compared with Groovy and Java, was related to using immutable data-structures. In Clojure all data types are, in principle, immutable. However, there is an Atom type in which values may be swapped. This was challenging because Atom values must be treated differently from pure values and lead to somewhat more verbose code than what could otherwise have been written. Also, Clojure allows the use of Java data structures, which are mutable and thus easier to work with in this case.

Python, as Groovy, is a multi-paradigm language combining the features of object oriented and functional paradigms. Creating the templates of the Python code was, although being the only language in this survey not based on the JVM, no more difficult than for the other languages. The main challenge was to handle the significant white-spaces of the Python syntax. To support this, PetriCode contains functionality to keep track of the current indentation level. This required no special treatment and was not strictly necessary, but allowed for much cleaner templates.

Table 1 shows the sizes of the Sender and Receiver principal code (measured in code lines) for each of the platforms considered. As can be seen, the code for

Listing 5: The Python template for ⟨⟨service⟩⟩ (left) and for ⟨⟨send⟩⟩ (right).

```
<%import static org.k1s.petriCode.
        generation.CodeGenerator.indent
%>${indent(indentLevel)}def ${name}(self,<%
def paramsVal = ""
def params2 = []
if(binding.getVariables()
   .containsKey("params")){
  params.each{
    if(it.trim() != "") params2 << "$it"
  }
  paramsVal = params2.join(", ")
%>$paramsVal<%}%>):
%%yield_declarations%%
%%yield%%
```

```
<%import static org.k1s.petriCode.
   generation.CodeGenerator.indent
%>${indent(indentLevel)}
   ${params[0]}.sendto(
     pickle.dumps(${params[1]}),
     (${params[2]}["host"],
     ${params[2]}["port"]))
```

Python is much smaller than the others. This is due to the efficient libraries in Python and that the Python code, for technical reasons, have much fewer blank lines which is also reflected in the templates. Table 2 shows the sizes, in lines, for selected templates and all the templates for each platform. The sizes reported are the sizes in the actual code and may not correspond to the templates as they are formatted in this paper. In this example, there was the same number of templates for each platform, but this is not necessarily always the case. As can be seen in Table 2, there is not a perfect correlation between the size of templates and the size of the generated code. This is due to, in part, some templates being more complex for some languages than others and template reuse being possible for some languages. An example is the Clojure templates, where the templates for the ⟨⟨setField⟩⟩ and ⟨⟨setValue⟩⟩ pragmatics are the same, but since the ⟨⟨setValue⟩⟩ template has more functionality than the ⟨⟨setField⟩⟩ template for all platforms, this results in a higher total number of template lines for Clojure. For each of the languages eleven new templates were constructed while ten templates were already provided as part of the PetriCode tool. The new templates were templates that are specific to the pragmatics applied for the protocol considered.

4 Evaluating Intergrateability

It should be possible to integrate code generated by our approach with existing software. We evaluate two types of integration with other software. The first type can be exemplified by having our generated cod use another library for sending and recieving data from the network. We call this type of integration downwards integration (i.e, generated code can use different third-party libraries). The other type can be exemplified by creating a runner program that employs the generated protocol for sending a message to a server. This type is called upwards integration (i.e, applications can use services provided by the generated code). We have evaluated integratability using the code generated for the Java platform based on the example in Sect. 2. However, the results are applicable for other platforms as well.

Downwards Integration. We have already shown that by changing templates, our approach can be used to generated code for different platforms. The same technique can be used to employ various libraries on the same platform to perform the same task. We illustrate this by changing the network library from the standard `java.net` library to Netty [16]. This example was chosen because

Language	Groovy	Java	Clojure	Python
Sender	131	132	119	66
Receiver	81	78	68	38
Total	212	210	187	104

Table 1: Sizes of the generated code.

Language	Groovy	Java	Clojure	Python
service	19	28	15	15
runInternal	4	10	4	3
send	9	9	8	2
All templates	154	219	251	112

Table 2: Size of code generation templates.

Listing 6: The Java template for ⟨⟨send⟩⟩ with Netty (left) and the runner for the generated Java code (right).

```
1  ByteArrayOutputStream bos =
2    new ByteArrayOutputStream();
3  ObjectOutputStream oos =
4    new ObjectOutputStream(bos);
5  oos.writeObject(${params[1]});
6  byte[] _msg_ = bos.toByteArray();
7  ((io.netty.channel.Channel)
8    ${params[0]}[0]).writeAndFlush(
9      new io.netty.channel.socket
10       .DatagramPacket(
11       io.netty.buffer.Unpooled
12       .copiedBuffer(_msg_),
13     new InetSocketAddress(InetAddress
14       .getByName((String)((Map)
15         ${params[2]}).get("host"))
16     ,(Integer)((Map)${params[2]})
17     .get("port")))).sync();
```

```
1  def sender = new Sender.Sender()
2  def reciever =
3      new Receiver.Receiver()
4  t = new Thread().start {
5    def ret = reciever.receive(31339)
6    println "Recieved: ${ret}"
7  }
8  def msg = [
9   [1,0,'Col'],[2,0,'our'],[3,0,'ed '],
10  [4,0,'Pet'],[5,0,'ri '],[6,0,'Net'],
11  [7,1,'s']
12 ]
13 sender.send(
14   msg,[host:"localhost", port:31339])
```

networking is an important function of the network protocol domain that we consider, and because Netty is substantially different from `java.net` as it is an event driven library.

Three out of twenty-one templates had to be altered to accommodate Netty as the network library for the sender principal. These were the templates that generate code for sending and receiving data from the network. We show the Netty variant of the send template from Listing 2 (right) in Listing 6 (left). The main differences is the call to the socket (or channel in the terminology of Netty) to send the message (lines 6-12).

Upwards Integration. The ability to call the generated code is necessary for the code to be useful in many instances. Our approach allows this by explicitly modelling the API in the CPN protocol model in the form of services which defines the class and method names. To demonstrate upwards integration, we have created runners for the generated implementations for each of the platforms considered. The runner for the Java platform can be seen in Listing 6 (right). This demonstrates that it is possible to use the generated services from third party software. It is worth noticing that the explicit modelling of services in the CPN model implies that it is simple to invoke the generated code.

5 Evaluating Readability

We have evaluated the readability of code generated by PetriCode in two ways. One way is that we applied a code readability metric [2] to selected snippets of the generated classes from the example described in Sect. 2, and the example described in previous works [14]. Furthermore, we have conducted a field study

The Buse-Weimer experiment (BWE)	The experiment conducted by Buse and Weimer to create the BWM. The snippets were selected from open source experiments.
The metric experiment (ME)	Our experiment to validate the results from BWE for professional developers. This experiment evaluated the twenty first snippets evaluated in the *BWE*.
The code generation experiment (CGE)	Our experiment to evaluate readability of generated code compared to non-generated code. Eight snippets were randomly selected from generated code and twelve from the open source projects in the network protocol domain.

Table 3: Overview of the experiments conducted and discussed in this section

where software engineers were asked to evaluate the readability of the generated code. This study was also used to evaluate the code readability metric.

We use the *Buse-Weimer metric* [2] (BWM) as a code readability metric. This metric was constructed by Buse and Weimer based on an experiment (the *Buse-Weimer experiment* (BWE), see Table 3) asking students at the University of Virginia to evaluate short code snippets with regards to readability on a scale of one to five. The experiment was used to construct the metric using machine learning methods to compute weights on various factors that have an impact on code readability. The final metric scores code snippets on a scale from zero to one where values close to zero indicates low readability and values close to one indicates a high degree of readability.

Our field study with software engineers took place at the JavaZone software developer conference in Oslo, Norway in September 2013. The experiment was organized into two parts. One part (*the metric experiment* (ME), see Table 3) evaluated the BWM. The other part (the *code generation experiment* (CGE), see Table 3) evaluated the readability of the generated code compared to non-generated code. Both experiments were conducted by asking software developers to evaluate twenty small code snippets with regards to readability by assigning values, on a scale from one to five, to each code snippet. The experimental set-up was created to mimic the BWE. The main advantage of our experiment over the BWE is that the dominating majority of the participants were professional software developers instead of students. The ME had 33 participants while the CGE had 30 participants.

For the CGE, we randomly selected code snippets from code generated for the Java platform based on the example described in Section 2, and the example described in [14]. We use code for the Java platform because it was used in the BWE, and the subjects of our experiments knew Java. Also, there exist several Open Source projects from which to obtain snippets for our experiments. In addition to the generated snippets, we selected, as controls, snippets from three Open Source projects in the network protocol domain. These were the Apache FtpServer, HttpCore and Commons Net [15]. All three are part of the Apache project, and we consider them to be high quality projects within the network protocol domain.

In the ME, we used the first twenty snippets from the BWE. Since we did our experiment at a conference, we could not redo the experiment with all the

Snippet	1	2	3	4	5	6	7	8	Mean	Median
Score	0,14	0,03	0,19	0,28	1,00	1,00	1,00	0,99	0,58	0,63

Table 4: The results for the BWM on generated code

Snippet	1	2	3	4	5	6	7	8	9	10	11	12	Mean	Median
Score	0,54	0,95	0,15	0,79	0,01	0,40	0,26	0,04	0,00	0,01	0,96	0,65	0,40	0,33

Table 5: The results for the BWM on selected hand-written protocol software snippets

one hundred snippets from the BWE and still expect enough software engineers to participate.

Applying the Buse-Weimer Metric. The BWM is based on the scores of hundred small code snippets. Even though the size of the snippets are not scored directly, some of the factors are highly correlated with the snippet size [11]. This makes it inappropriate to measure entire applications. Therefore, we applied the metric to the snippets selected for the CGE.

Table 4 shows the results of running the BWM tool on each of the generated code snippets. The mean and median score is above 0.5, indicating that the code is fairly readable. Also the mean and median of the generated code is higher than for the non-generated protocol-code as can be seen in Table 5.

Although the scores of the BWM on the generated code are highly encouraging, the scores are either very high or very low. This motivates an independent evaluation of the readability of the generated code (see below), as we have done with the CGE, and to validate the BWM, as we have done with the ME (see below).

The Metric Experiment: Validating the Buse-Weimer Metric. The ME was conducted to validate the BWM and the BWE. This experiment measured twenty of the code snippets that were measured in the BWE in a similar manner. The goal was to determine whether the results of the BWE holds for professional software developers.

Figure 4 shows the means of the BWE (blue/solid) and our repeat (red/-dashed) for the selected snippets. The figure suggests significant covariance even if the students in the BWE tended to judge snippets higher than the software developers in our ME. We computed three statistical tests on the correlation between the means of the two experiments (see Table 6). The correlation tests show that there is strong to medium correlations between the means and that the correlation is statistically significant($p<0.05$).The correlation tests were carried

Method	**Corrolation**	**P-value**
Pearson	cor = 0,82	$9{,}28 \cdot 10^{-06}$
Spearman	rho = 0,79	$2{,}94 \cdot 10^{-05}$
Kendall	tau = 0,61	$1{,}65 \cdot 10^{-04}$

Table 6: Correlations between the means of the ME and the BWE

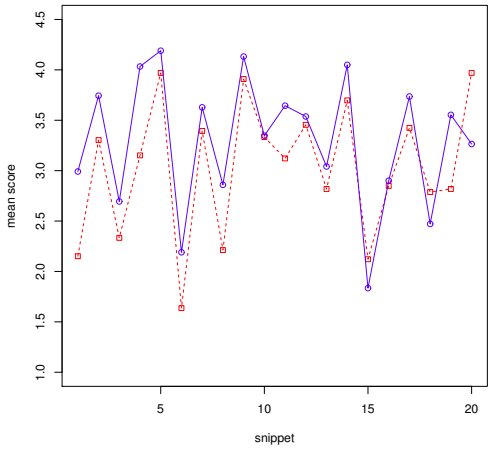

Fig. 4: The values of the selected snippets for the BWE and the ME.

out using the R [12] tool and the standard correlation test call, `corr.test()`, with all the methods available for the call. These results indicate that the BWE is relevant to professional software developers.

Mean values for the original BWE and for the ME are shown in Table 7. As can be seen, *the ME* resulted in somewhat lower scores than that of the BWE, in fact it is lower in 17 out of 20 instances. In order to determine the significance of this observation we conducted a T-test. The results of the T-test does not allow us to rule out that the means are not equal (p=0,21), although it does not give us statistically significant results on the repeat always being higher either (p=0,10), although that may be more likely.

The ME showed that there is a significant correlation between the results of the BWE (conducted with students) and the *ME* (conducted with software development professionals). This can be interpreted as evidence that the results from BWE also has validity for professional developers, although the metric based on it might be in need of some minor adjustments.

Snippet	1	2	3	4	5	6	7	8	9	10
Metric Experiment	2,15	3,30	2,33	3,15	3,97	1,64	3,39	2,21	3,91	3,33
Buse-Weimer Experiment	3,02	3,78	2,72	4,07	4,23	2,21	3,66	2,88	4,17	3,38
	11	12	13	14	15	16	17	18	19	20
	3,12	3,45	2,82	3,70	2,12	2,85	3,42	2,79	2,82	3,97
	3,68	3,57	3,07	4,08	1,85	2,93	3,77	2,49	3,58	3,29

Table 7: Snippet means for the metric and BWE.

The Code Generation Experiment: Comparing Generated and Handwritten Code. We expected that the generated code would not do quite as well as the hand-written high-quality code used as control. Therefore, our hypothesis was that the generated code would be within the standard deviation of the hand-written written code. The mean score for each of the snippets in the CGE are shown in Table 8. Snippets one to eight are generated code while snippets eight to twenty are hand-written. To check our hypothesis, we ran Welch's T-test on the results which is useful for determining the difference between the two samples. The first hypothesis we checked was whether the generated snippets are less readable than the hand-written ones. The results of a Welch's Two Sample t-test showed that the generated code-snippets scored below that of the hand-written code ($p=4{,}81 \cdot 10^{-05}$).

Then we checked the hypothesis when reducing the score of the measurements from the Apache projects by the standard deviation of those measurements. The Welch's Two Sample t-test indicates that the generated code scores better than one standard deviation below the hand-written code (p=0,03). This indicates that our goal of being readable within a standard deviation of non-generated code is met both by measuring via the BWM and experimentally.

Table 9 shows normalized means for each snippet from the CGE and the results of running the BWM on the corresponding snippets. As can be seen, the correlation is less than strong as confirmed by running correlation tests (see Table 10). Even though the results of the ME indicates that the BWE, which the BWM is based on, is valid even for professional software developers, we content that the results of the CGE are more reliable than the BWM. This is because the BWM is derived from software from different domains and that it is sensitive to snippet length. This indicates that the BWM is not relevant to code for network protocols.

Assessment of Validity of Our Results. As with most experimental approaches, this evaluation has some threats to the validity of the results. These are issues we have identified that might skew our results. One such threat to validity for the original BWE was that they used student as subjects who may or may not disagree with professional software developers on the readability of code. We have tried to alleviate this threat in the ME by repeating part of the BWE with professional developers. Further threats to validity to the experiments and results described in this section are discussed in the following.

Small sample size and limited number of participants may skew the results. Since we conducted this experiment at a software developer conference where people tended to be on their way to some lecture, we had to limit the number

Snippet	1	2	3	4	5	6	7	8	9	10
Mean	2,40	2,10	3,83	3,23	2,67	3,13	2,97	2,73	3,90	3,97
	11	12	13	14	15	16	17	18	19	20
	2,67	3,13	2,73	3,43	3,67	3,07	3,83	2,00	3,20	3,93

Table 8: Means of results for generated code (1-8) and Apache projects code (9-20).

Snippet	1	2	3	4	5	6	7	8	9	10
Experiment Score	0,48	0,42	0,77	0,65	0,53	0,63	0,59	0,55	0,78	0,79
Metric Score	0,14	0,03	0,19	0,28	1,00	1,00	1,00	0,99	0,54	0,95

11	12	13	14	15	16	171	18	19	20
0,53	0,63	0,55	0,69	0,73	0,61	0,77	0,40	0,64	0,79
0,15	0,79	0,01	0,40	0,26	0,04	0,00	0,01	0,96	0,65

Table 9: Normalized means from the CGE and results from applying the BWM

Method	**Corrolation**	**P-value**
Pearson	cor = 0,21	0,37
Spearman	rho = 0,20	0,40
Kendall	tau = 0,14	0,40

Table 10: Correlation between normalized experimental scores and the BWM applied to the same snippets

of snippets we asked each participant to evaluate. Also, because professional software developers are harder to recruit than students, the number of participants was limited. Furthermore, it is possible, albeit unlikely, that the people participating in the experiment are not representative for software developers as a whole. These threats can be alleviated by conducting broader studies on larger groups of developers and using interviews.

In our experiments, we used small randomly selected code snippets as proxies for code readability. We do this both for practical and conceptual reasons. The practical reasons revolve around what we realistically could expect participants to score. If they had to read entire classes or software projects in order to score the code, this would have taken to much time and could have resulted in getting too few participants in our experiments. Furthermore, we wanted to evaluate the BWM since it is the only implemented metric we could find in the literature. The more conceptual reason is that if each snippet is readable, then the whole code is likely to be readable as well. In our approach, high-level understanding is based more on the CPN models of the protocols than on the implementation, so it makes sense for us to concentrate on low level, snippet-sized readability, since readability in the large is intended to be considered at the level of the model.

6 Conclusions and Related Work

In this paper, we have evaluated our code generation approach and supporting software, with respect to platform independence, the integratability of the generated code as well as the readability of the generated code.

Platform independence was evaluated by generating code for a protocol for three platforms in addition to the Groovy platform from a single CPN model. The number of and differences between the platforms gives us confidence that our approach and the PetriCode tool can be used to generate protocol implementations for many target platforms. All the platforms considered have automatic memory management in the form of garbage collection. This is convenient, but

we intend to support platforms without automatic memory management in the future.

Platform independence is especially important for network protocols since they are used to communicate between two or more hosts that often run on different underlying platforms. Although there exists many tools that allow generating code from models claiming to be platform independent, few studies seem to have been made actually generating code for several platforms.

MDA [8] and associated tools rely on different platform specific models (PSM) to be derived for platforms before generating code for each platform. This adds an extra modelling step compared to our approach and may require somewhat different PSMs for different platforms. The Eclipse Model To Text (M2T) [3] project provides several template languages for code generation from Ecore models. In general, M2T languages can generate code for several platforms. However, to go beyond pure structural features and standard behaviour, the developer must create customized code generators. In [9] code is for protocol is generated using UML stereotypes and various UML diagram types. The UML diagrams, annotated with stereotypes according to a custom made UML profile, combined with a textual language named GAEL are used to obtain protocol specification in the Specification and Description Language (SDL) [1, 4]. The authors also conjecture that the approach can be used to generate code for any platform. The use of stereotypes in the approach presented in [9] is similar to the pragmatics that our approach uses. However, a difference is that several diagram types are used in the UML based approach in contrast to our approach where we use CPNs to describe both structure and behaviour.

MetaEdit+ [17] allows code generation of visual Domain Specific Modelling Languages (DSMLs). MetaEdit+ and the DSML approach is similar to the PetriCode approach since CPNs and pragmatics constitute a DSML. A main difference is that MetaEdit+ allows users to generate custom graphical languages while PetriCode uses CPN, but extends CPNs using pragmatics. This allows us to use the properties of CPNs for verification and validation, and also to use a single syntax for different domains.

The Renew [7] tool uses a simulation-based approach where annotated Petri Nets can be run as stand-alone applications. The simulation-based approach is fundamentally different from our approach where the generated code can be inspected and compiled in the same way as computer programs created with traditional programming languages. A detailed comparison between these two approaches would be an interesting avenue for future work.

We evaluated the integratability of the generated code in two directions: upwards and downwards integratability. Upwards integratability was evaluated by showing that the generated protocol software can be called by programs running the protocols. Downwards integratability was evaluated by showing how we can change the network API for the Java platform by binding different templates to some of the pragmatics.

Readability of the generated code was evaluated by an automatic metric and an experiment. According to the BWM, the generated code is as, or possibly

even more, readable than the samples of high quality code in the same domain that we used for comparison. Based on our experiment with software developers, however, the generated code is somewhat less readable but within a standard deviation of the non-generated code. A contribution of this paper is also to provide evidence that the experimental results from the BWE are relevant to professional software developers in addition to the students. However, based on the discrepancy between the experimental evaluation, it seems that the BWM may not be applicable to code in the network protocol domain. To the best of our knowledge, there are no previous work evaluating intergrateability and readability of automatically generated software.

In the future we will evaluate the verifiability of the models used in our approach by applying verification techniques to example protocols. We also intend to develop a set of template libraries that can be used for code generation as well as procedures for testing code generation templates. Another possible direction for future work is to apply our code generation approach to other domain.

References

1. F. Babich and L. Deotto. Formal methods for specification and analysis of communication protocols. *Communications Surveys Tutorials, IEEE*, 4(1):2–20, 2002.
2. R.P.L. Buse and W.R. Weimer. A metric for software readability. In *Proc. of ISSTA'08*, pages 121–130, NY, USA, 2008. ACM.
3. IBM. *Eclipse Model To Text (M2T).* `http://www.eclipse.org/modeling/m2t/`.
4. ITU-T. Recommendation z.100 (11/99) specification and description language (sdl), 1999.
5. K. Jensen and L.M. Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009.
6. L.M. Kristensen and K.I.F. Simonsen. Applications of Coloured Petri Nets for Functional Validation of Protocol Designs. In *ToPNoc VII*, volume 7480 of *LNCS*, pages 56–115. Springer, 2013.
7. 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.
8. Object Management Group. *MDA Guide*, June 2003. `http://www.omg.org/cgi-bin/doc?omg/03-06-01`.
9. J. Parssinen, N. von Knorring, J. Heinonen, and M. Turunen. UML for protocol engineering-extensions and experiences. In *Proc. of TOOLS '00*, pages 82–93, 2000.
10. PetriCode. *Example protocol.* `http://bit.ly/19HU8U4`.
11. D. Posnett, A. Hindle, and P. Devanbu. A simpler model of software readability. In *Proc. of MSR '11*, pages 73–82. ACM, 2011.
12. R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, 2013.
13. K. I. F. Simonsen. Petricode: A tool for template-based code generation from cpn models. In S. Counsell and M. Núñez, editors, *Software Engineering and Formal Methods*, volume 8368 of *LNCS*, pages 151–163. Springer, 2014.
14. 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.

15. The Apache Software Foundation. FtpServer http://mina.apache.org/ftpserver-project/, HttpCore https://hc.apache.org/, Commons Net http://commons.apache.org/proper/commons-net/.
16. The Netty project. *Netty*. http://netty.io.
17. J.P. Tolvanen. MetaEdit+: domain-specific modeling for full code generation demonstrated. In *Proc of SIGPLAN '04*, pages 39–40. ACM, 2004.