PetriCode: A Tool for Template-based Code Generation from CPN Models

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
Email: {kisi}@imm.dtu.dk

Abstract. Code generation is an important part of model driven methodologies. In this paper, we present PetriCode, a software tool for generating protocol software from a subclass of Coloured Petri Nets (CPNs). The CPN subclass is comprised of hierarchical CPN models describing a protocol system at different levels of abstraction. The elements of the models are annotated with code generation pragmatics enabling PetriCode to use a template based approach to generate code while keeping the models uncluttered.

1 Introduction

This paper describes the tool PetriCode which is a platform implementing the approach presented previous works [16]. In contrast to previous works [16, 18, 17], this paper focuses on the techincal software realization of our approach whereas earlier work has focused on the conceptual and theoretical aspects of our modelling and code generation methods. The intended use of PetriCode is to generate software for network protocols based on annotated and descriptive protocol models [18] in a flexible way and for different target languages and platforms.

PetriCode takes a template-based approach to code generation from CPN models [6] annotated with pragmatics. Pragmatics are syntactic annotations on CPN elements that are used to guide and provide hints to the code generation procedure. Pragmatics are associated with code templates that are invoked for code generation. Our code generation approach outlined in previous work [**?**] consists of three main steps. The first step of the code generation is to parse the CPN model and automatically derive additional pragmatics for the CPN model. The derived pragmatics are used to provide the code generator with additional information of what is represented by the various CPN structures. The second step is to construct an Abstract Template Tree (ATT)which is used as an intermediary structure for code generation. The ATT provides a flexible and platform independent data structure that simplifies the final step of the code generation. The third and final step is the actual code generation where the ATT, using a series of visitors and templates, is transformed into code.

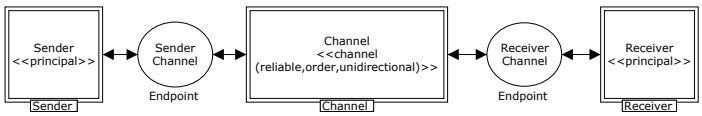

Fig. 1. The protocol system level

The rest of this paper is organized as follows. Section 2 shows, by an example, how PetriCode can be used to generate code for a simple framing protocol. Section 3 describes how the tool has been implemented. Section 4 contains a discussion of related works. Concluding remarks and future work are presented in 5. Details on how to download operate PetriCode and a screen capture showing how the tool can be used are available at the PetriCode project website[15]. This paper assumes that the reader is familiar with hierarchical CPNs.

PetriCode is freely available together with documentation [15]. A binary version together with the example in this article is provided for reviewers at http://kentis.github.io/nppn-cli/files/petriCode.tar.gz.

2 Example

In order to present the workings of PetriCode, we use a simple framing protocol as a running example. The protocol is described in the technical report [16]. The model is divided into three hierarchical layers: the protocol system, principal, and service layers. The protocol system layer, depicted in Fig. 1, shows the principal agents of the protocol as well as the connections between them. In the example, those are the Sender, Receiver and the Channel between them. In Fig. 1 the substitution transitions Sender and Receiver are both annotated with a $\langle\langle \mathsf{principal} \rangle\rangle$ pragmatic. This conveys to the code generator that the sub-modules represented by each of these substitution transitions represent principal agents of the system. The third substitution transition in the protocol system module, Channel, is annotated with the pragmatic $\langle\langle \mathsf{channel} \rangle\rangle$ specifying that the underlying module defines the channel. In the rest of this paper, we focus on the Sender agent of the protocol. Figure 2 shows the principal level of the sender. The Open and Close Open and Close services the channel to the Receiver while Send sends a message over the channel. In this paper, we go into details with the Send service. The principal level contains the services provided by each principal as well as life cycle variables which controls when the various services can be called and places which hold global data for the principal. The Send service, shown in Fig. 3, shows the sending part of the protocol. The Send service divides a message into smaller fragments called frames. Each frame is sent together with a bit that is set if the current frame is the last frame of the message, and unset otherwise. In the model, the message, which is a parameter to the Send service, is broken up into frames by the transition Partition. Then the fragments are sent one by one in a loop until all the fragments have been sent.

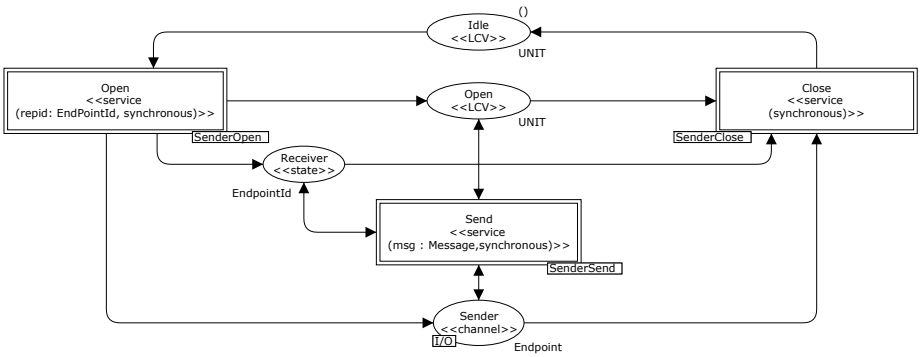

Fig. 2. Example of a principal level module: The Sender module

In order to generate code from the CPN model, PetriCode is invoked with appropriate arguments. An example of such an invocation is shown in Listing 1. Here the last argument is the protocol model FrameingProtocol.cpn. The first step of the program is to parse the model and add derived pragmatics. It is also possible, as part of the command-line arguments, to give further pragmatics and rules for deriving them as will be discussed in Section 3.2. The second step is to generate the ATT which is discussed further in Section 3.3. The third and final phase is the code generation where the -o option provides the output directory where the generated code is placed and the -b option takes a *binding descriptor* file as an argument. The binding descriptor file provides a set bindings of pragmatics to code generation templates for the specific platform under consideration. These bindings, known as *template bindings* are described in further detail in Section 3.4. One thing that is not in the listing is a reference to pragmatics descriptors which describes the available pragmatics. This is because a core set of pragmatics, which contains most of the pragmatics used in this particular example, are defined in the tool and available by default.

Listing 1. Command to run PetriCode for the simple framing protocol example.

```
petriCode -o . -b ./groovy.bindings ./FrameingProtocol.cpn
```

After running the command shown in Listing 1 two files will be generated in the output directory. Each of these files will contain a single Groovy class, one for the Sender principal and one for the Receiver. For Sender class there will be exactly three methods, one for each of the services the principal provides (see Fig. 2). The code for the Send service is shown in Listing. 2. The first line of the code defines the method implements contain the service. Lines two to three check that preconditions, that are defined at the principal level and are translated to class fields, are fulfilled and then setting of a precondition field (denoted as a place annotated with $\langle\langle \mathsf{LCV} \rangle\rangle$). The next few lines initializes variable for the method. Then, in line ten and eleven, the $\langle\langle \mathsf{partition} \rangle\rangle$ pragmatic, which is generated using a custom template, is shown. In lines thirteen and fourteen, the loop is

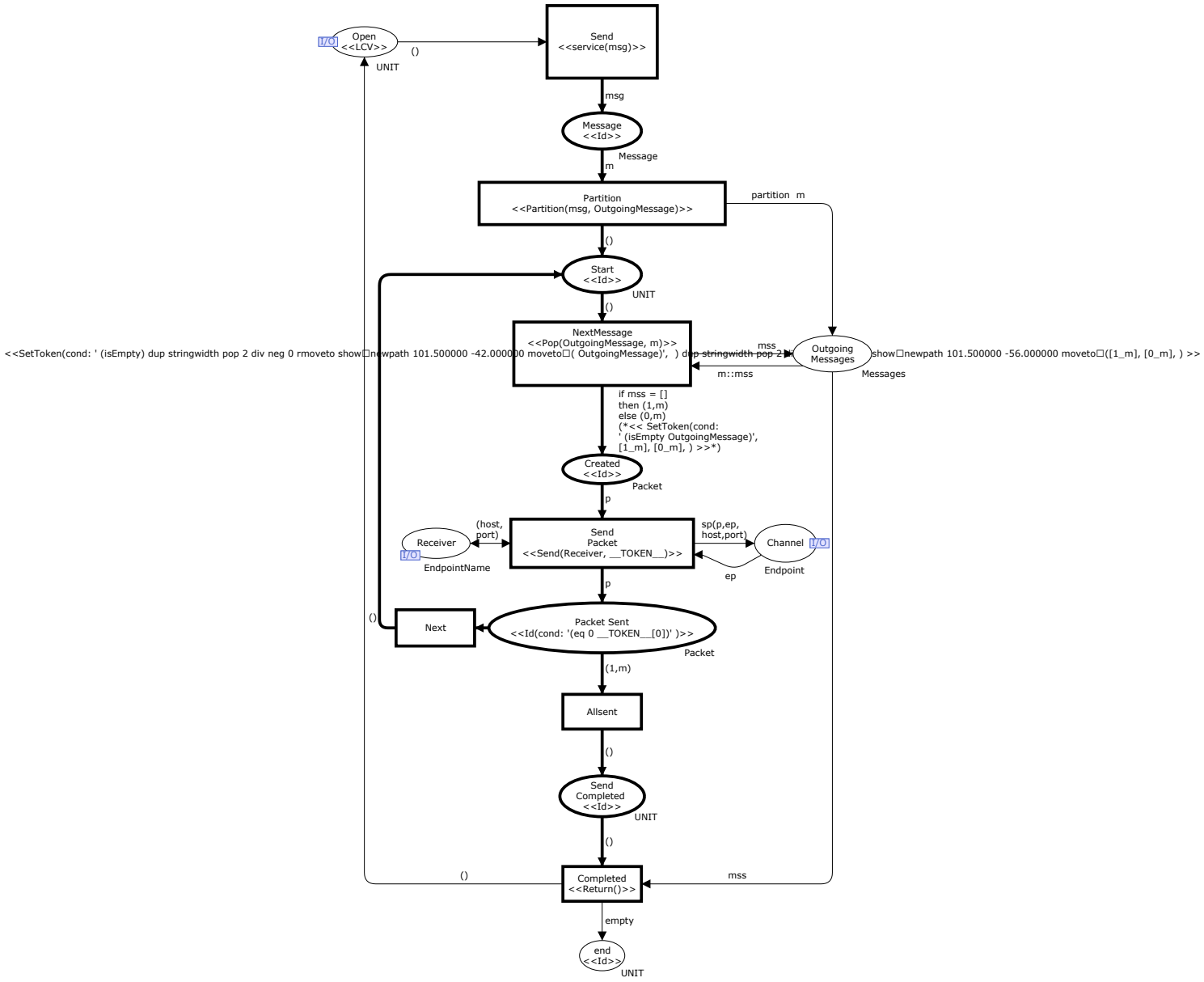

Fig. 3. The Sender Send module

started by setting a loop variable named "__LOOP_VAR__" to true and entering a while loop. Inside the loop, first, a variable is initialized in line sixteen and then the code emitted by the template for ⟨⟨pop⟩⟩ is shown in line seventeen. The next lines determine whether the current frame is the final frame of the message

Listing 2. Generated code for Send service of Sender principal.

```
1  def Send(msg){
2    if(!Open) throw new Exception('unfulfilled precondition: Open')
3    Open = false
4    /*vars: [__TOKEN__, msg, OutgoingMessage, __LOOP_VAR__]*/
5
6    def __TOKEN__
7    def OutgoingMessage
8    def __LOOP_VAR__
9
10   OutgoingMessage = msg.getChars().toList()
11     .collate(5).collect{ new String(it.toArray(new char[0])) }
12
13   __LOOP_VAR__ = true
14   while(__LOOP_VAR__){
15     /*vars: [m]*/
16     def m
17     m = OutgoingMessage.remove(0)
18     if(OutgoingMessage.size() == 0){
19       __TOKEN__ = [1,m]
20     } else {
21       __TOKEN__ = [0,m]
22     }
23     Receiver.getOutputStream().newObjectOutputStream().writeObject( __TOKEN__)
24     __LOOP_VAR__ = ( 0 == __TOKEN__[0] )
25   }
26   Open = true
27   return
28 }
```

and append the appropriate integer. Finally, in line twenty-three, the message is transmitted and the loop variable is updated in line twenty-four. Outside the loop the life-cycle variables are updated and the service terminates.

3 Architecture and Design of PetriCode

PetriCode is divided into three functional modules corresponding to the three main steps in our code generation approach. The modules are the Pragmatics, ATT and Code generation modules. These modules will be discussed in detail in later sections. When invoked, PetriCode invokes each of the modules in sequence. Each module is passed all relevant options and parameters as well as the output of previous modules as needed.

When designing and implementing PetriCode there was a number of key requirements that needed to be addressed and which affected the choice of software technologies used for the implementation. An important feature of PetriCode is

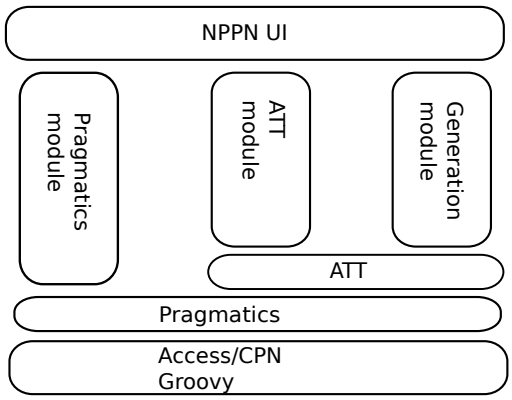

Fig. 4. Architectural overview of PetriCode

the ability to read, parse and write CPN models stored in the format of CPN Tools [7]. A Java library Access/CPN [20] provides this capability for the Java platform. Therefore, in order to use Access/CPN it is necessary to choose a platform with good integration with Java libraries. It is also useful to be able to alter the Access/CPN model slightly by adding a `pragmatics` field to the model elements avoiding an overly complicated translation layer. Another important requirement was to easily be able to create Domain Specific Languages (DSLs) for defining pragmatics descriptions and template binding. The Groovy programming language [4], which runs on the Java Virtual Machine, was chosen since it has a seamless integration with all Java libraries including Access/CPN. Groovy also has a simple mechanism, not available to Java, to manipulate classes at runtime and also has good support for many types of DSLs. Groovy also has other useful features such as a command-line interface options builder and an easy to use yet powerful template engine.

3.1 Overall Architecture

change main class name in code and here.

Figure 4 provides an architectural overview of PetriCode. PetriCode is controlled by its main class PetriCode in the UI module. PetriCode parses the command-line arguments and calls the modules shown directly below the UI model in Fig. 4 as appropriate. PetriCode uses the CliBuilder included in Groovy to parse command line arguments. All the modules depend on Access/CPN for reading and manipulating CPN models. In addition, PetriCode is implemented using the Groovy language and builds upon the Groovy and Java platforms. All the modules are also dependent on the data model for pragmatics. The ATT and Generation modules also share a data model for ATTs.

The overall program flow of PetriCode is shown in Fig. 5. Each column of the flow chart represents a module of PetriCode. The left column is the pragmatics module, the middle column is the ATT module and the right column are operation of the generation module. The process of code generation begins with

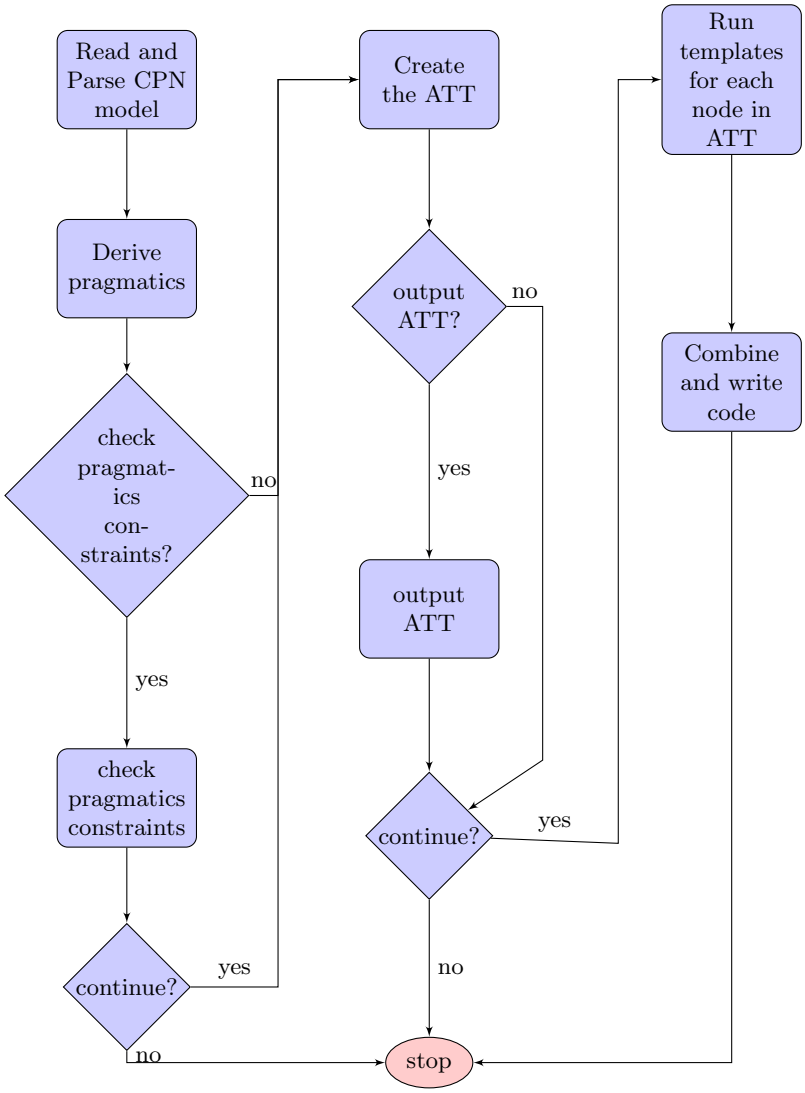

Fig. 5. Control flow of PetriCode

reading and parsing a CPN model including parsing explicit pragmatics. The next step in the code generation process is to derive pragmatics. This is done since pragmatics do not have to be added manually, but instead they can be automatically derived from the structure of the model. After all the derived pragmatics have been added to the CPN, the generation process optionally checks the consistency of the pragmatics. This is optional because, in some circumstances, the check requires all pragmatics to be defined in a pragmatics description, and this may slow down software development in early stages of a project. After op-

tionally checking the pragmatics, the code generation process turns to the ATT module. The first part here is to generate the ATT, which is used internally by the code generation module to generate code. After generating the ATT, the ATT can optionally by output in either an XML format or as a picture. Based on the command-line arguments given, the process optionally terminates after generating and outputting the ATT. If the generation process is to continue, the code generator, using a series of visitors on the ATT, will generate code appropriate for each node of the ATT. After generating code for each node in the ATT, the generated code fragmants are recombined in a bottom up fashion until the code for each principal has been completely generated. Finally, the code is written to files and the process is ended.

3.2 Pragmatics Module

The Pragmatics module has three main responsibilities: reading and parsing CPN models, parsing *pragmatics descriptors* and adding derived pragmatics to CPN models. The pragmatics derivation process is driven by a DSL which is used to parse the pragmatics descriptor files containing information about the pragmatics used in a model.A class diagram showing the meta-model for pragmatics is shown in Fig. 6. In the diagram pragmatics are separated in two dimensions. One dimension is the explicit/derived dimension, where explicit pragmatics must be explicitly added to the CPN model by the modeller, whereas derived pragmatics may be derived based on structural patterns. The second dimension is whether the pragmatic is supplied by the user (a custom pragmatic) or is part of the built-in core pragmatics of PetriCode. In the class diagram the different types of pragmatics are represented using an inheritance hierarchy. This approach has not been chosen in the code because it would increase the complexity of the implementation. Instead PetriCode uses an object derived from the pragmatics descriptor to contain, in the form of fields, the information contained in the inheritance hierarchy.

The pragmatics description language is a simple builder language that describes the available pragmatics. A core set of pragmatics (see Listings 3 and 4) is provided by PetriCode while others may be provided by the user using the pragmatics description language. The language consists of *descriptors* that each describe a pragmatic. Each descriptor consists of a name, which is the name of the pragmatic, followed by a pair of parenthesis. Inside the parenthesis, the parameters of the pragmatics definition are given in the form of key-value pairs. The possible parameters for a pragmatics descriptor are `origin` and `derivationRules`. The `origin` parameter indicates whether the pragmatic is explicitly given by the modeller or should be automatically derived. The first pragmatic descriptor in Listing 3 describes the ⟨⟨Principal⟩⟩ pragmatic. The origin field of ⟨⟨Principal⟩⟩ indicates that this is an explicit pragmatic meaning that it will not be generated automatically. The `derivationRules` parameter gives the patterns that should be used to find the elements of a CPN model where a derived pragmatic should be added. The patterns for derived pragmatics are described shortly.The last pragmatic descriptor in Listing 4 is the ⟨⟨endLoop⟩⟩

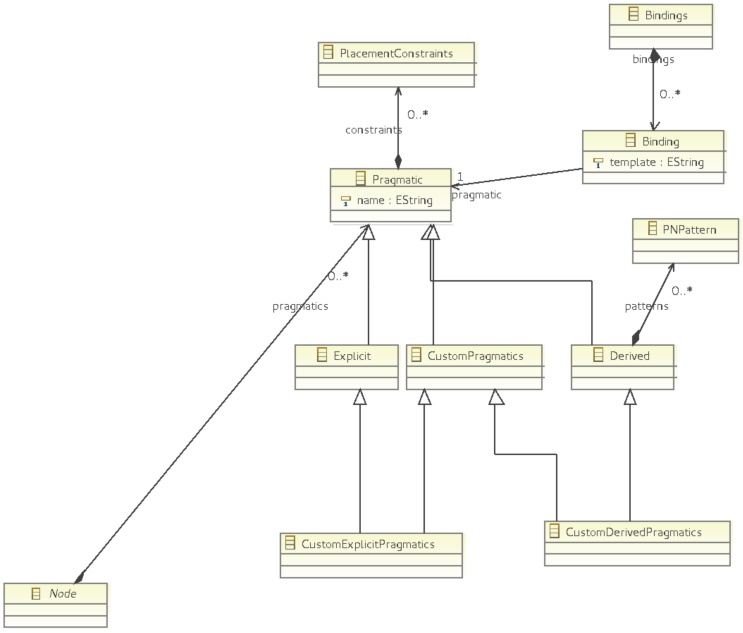

Fig. 6. Data model for the Pragmatics module

pragmatic. The ⟨⟨endLoop⟩⟩ pragmatic has derivation rules that states that it should be added to nodes already annotated with ⟨⟨Id⟩⟩ , with at leat two outgoing edges and exactly one backlink, that is a connection to somewhere earlier in the control flow path (see Section 3.2). In addition both ⟨⟨Principal⟩⟩ and ⟨⟨endLoop⟩⟩ have some constraints indicated by the `constraints` field. This means that the pragmatics should only recide on places where the constraint is fullfilled. In the case of the ⟨⟨Principal⟩⟩ pragmatic this means that it should only be used to annotate nodes on the protocol system level which are substitution transitions.

Listing 3. The explicit core pragmatics for PetriCode

```
principal(origin: 'explicit', constraints: [levels: 'protocol',
          connectedTypes: 'SubstitutionTransition'])

channel(origin: 'explicit')

id(origin: 'explicit', controlFlow: true, constraints:
           [levels: 'service', connectedTypes: 'Place'])
lcv(origin: 'explicit')
```

```
service(origin: 'explicit', constraints:
        [[levels: 'principal', connectedTypes:
        'SubstitutionTransition'],[levels: 'service',
        connectedTypes:'Transition']])

state(origin: 'explicit', constraints: [levels:
      ['principal','service'], connectedTypes:'Place'])

_return(origin: 'explicit', constraints:
        [levels: 'service',connectedTypes:'Transition'] )
```

Listing 4. The derived core pragmatics for PetriCode

```
branch(origin: 'derived', derviationRules:
       ['new PNPattern(pragmatics: [\'Id\'], minOutEdges: 2,
       backLinks: 0, forwardLinks: 0)'],block:
       [type: "branch", ends: "merge"],
       constraints: [levels: 'service', connectedTypes: 'Place'])

merge(origin: 'derived', derviationRules:
      ['new PNPattern(pragmatics: [\'Id\'], minInEdges: 2,
      backLinks: 0, forwardLinks: 0)'],
      constraints: [levels: 'service', connectedTypes: 'Place'])

startLoop(origin: 'derived', derviationRules:
          ['new PNPattern(pragmatics: [\'Id\'],
          minInEdges: 2, forwardLinks: 1)'],
          block: [type: "Loop", ends: "endLoop"],
          constraints: [levels: 'service', connectedTypes:'Place'])

endLoop(origin: 'derived', derviationRules:
        ['new PNPattern(pragmatics: [\'Id\'],
          minOutEdges: 2, backLinks: 1)'],
        constraints: [levels: 'service', connectedTypes:'Place'])
```

The following table shows the current set of core pragmatics included in PetriCode. Each pragmatic in the table has a name, a short description, the model levels it can be used on and the type of CPN elements it can be used to annotate.

Pragmatic	Description	Level	Type
principal	Denotes a principal agent.	Protocol system	Substitution transitions
channel	Denotes a channel	all	Substitution transitions and places
id	Denotes a place in the control-flow path	service	Place
LCV	Life cycle variable	principal	Place
service	A service	principal and service	Substitution transition or transition
state	A data holder	Principal and service levels	Places
return	Ending a service	Service level	Transitions
branch	Starts a conditional	Service level	Places
merge	Ends a conditional	Service level	Places
startLoop	Starts a loop	Service level	Places
endLoop	Ends a loop	Service level	Places

Pragmatics Derivation. The method for deriving pragmatics is based on traversing each service module and checking each node, i.e, place or transition, against structural patterns described by the pragmatic descriptors. The pragmatics are described in a pragmatics descriptor file as detailes in the following. An important concept for for pragmatics derivation and indeed the entire code geneation approach is the *control flow path*. The control flow path consits of all the nodes annotated with ⟨⟨ls⟩⟩ where the first node would be the node of a service annotated with ⟨⟨service⟩⟩ pragmatic and the last is annotated with ⟨⟨return⟩⟩ and ignoring loops. For derived pragmatics, a list of patterns are supplied. Each pattern, will be matched against each node on the control-flow path. If a pattern matches, the corresponding pragmatic is added to the node.

A pattern for matching Petri Net nodes (called a PNPattern) specifies a set of conditions that must be satisfied for a node (place or transition) to be matched. The conditions currently available are shown in Table 1. The current set of conditions, shown in Table 1 is a result of what conditions that has been identified as needed in the cases we have studied until now. It is, however, simple to add conditions.

The past- and future-link conditions are used to distinguish between the loop and conditional block structures. The progress concept inherent in these terms alludes to the order of nodes on the control-flow path. As an example, Listing 4 shows the derivation rules for the pragmatics used to describe loops (⟨⟨startLoop⟩⟩ and ⟨⟨endLoop⟩⟩) and branches (⟨⟨branch⟩⟩ and ⟨⟨merge⟩⟩). The derivation rule for ⟨⟨startLoop⟩⟩ and ⟨⟨merge⟩⟩ both require a minimum of two incoming arcs as seen by the minInEdges: 2 parameter. However, they are distinguishable by using our notion of progress though the control flow path by having a different value to the forwardLinks parameter. Similarly, both the ⟨⟨branch⟩⟩ and ⟨⟨endLoop⟩⟩ have a minimum number of outgoing arcs given by the minOutEdges:

Condition	Description
name	matches the name of the node.
pragmatics	matches the explicit pragmatics that are present on the node.
type	matches the type of the node (Transition or Place).
minInEdges	Matches against a given minimum number of incoming arcs of the node.
maxInEdges	Matches against a given maximum number of incoming arcs of the node.
minOutEdges	Matches against a given minimum number of outgoing arcs of the node.
maxOutEdges	Matches against a given maximum number of outgoing arcs of the node.
outArcInscription	Matches if at least one of the outgoing arcs contains the given arc-inscription.
adjacentPatterns	Matches adjacent nodes to a given pattern. This condition is considered fulfilled if any of the adjacent nodes fulfil the pattern.
pastLinks	Matches the number of incoming arcs that lead to a node earlier in the control flow.
futureLinks	Matches the number of incoming arcs that comes from a node later in the control flow.

Table 1. The current set of pragmatics derivation rules.

2. The difference between these two derivation rules concerns the backlinks, that is links from the node going against the direction of the control flow path, where the $\langle\langle$endLoop$\rangle\rangle$ is required to have one, while $\langle\langle$branch$\rangle\rangle$ may not have any. Using these derivation rules, PetriCode is able to automatically annotate the model in Fig. 3 with $\langle\langle$startLoop$\rangle\rangle$ at the transition Start and $\langle\langle$endLoop$\rangle\rangle$ at the transition Packet Sent. The example does not contain any examples of the $\langle\langle$branch$\rangle\rangle$ and $\langle\langle$merge$\rangle\rangle$ pragmatics.

Pragmatics Constraints. It is possible, when defining pragmatics, to add constraints on which elements a pragmatic may annotate. This is useful to ensure that pragmatics are used in the way it was intended when they were defined. The available parameters for the constraints are given in Table 2.

levels	The levels (Protocol system, principal or service level) the pragmatic is allowed on.
connectedTypes	The types of elements (i.e. Place or Transition) the pragmatic is allowed on.

Table 2. The current set of possible pragmatics constraints.

Defining Custom Pragmatics. Custom pragmatics are defined by creating a text file with lines like the ones shown in Listings 3 and 4. If a custom pragmatic is to be derived automatically, derivation rules must be supplied. In order to enforce correct use of the pragmatic it is also useful to include the proper pragmatic constraints to the pragmatic definition.

3.3 ATT Module

An ATT is an internal temporary data structure of PetriCode. Its purpose is to simplify and make more flexible the code generation process by organizing the block structures in an ordered tree. When this tree has been constructed, code generation is performed by traversing the tree. The tree is built up using the hierarchical structure of PetriCode models down to the service level. At the service level, the block structure of the service is reflected in the structure of the ATT.

The ATT module is responsible for generating ATTs and the main classes that make up the ATTs are show in Figure 7. After pragmatics are parsed and derived pragmatics have been added to the CPN model, the next step is ATT generation. The ATT generation is done by the ATTFactory class which produces an instance of the class AbstractTemplateTree. The AbstractTemplateTree has as its descendants instances of the classes Atomic, Conditional, Loop, Principal and Service. The Principal and Service classes each have a link going to the Instance class of the Access/CPN model which represents substitution transitions. The Block class has two outgoing associations with Place nodes from Access/CPN and the Atomic, in addition, has an association with transitions.

An ATT is implemented as an ordered tree. Each non-leaf element in the tree has a list of children. The root element of an ATT is an instance of the AbstractTemplateTree class. Each child of the root element is expected to be of the class Principal. The Principal class has as children the services of the principal. The Service class represents a service, its children are the blocks of the service according to the block structure introduced in [16].

There are three basic Block types in PetriCode: atomic, conditional and loop, each represented by a corresponding class. The Atomic class does not have any child elements since it is always a leaf node. Loop and Conditional however are not leafs and have children elements. The children of Conditional and Loop can be the same as for Service.

The ATT from the example in Section 2 is shown in Fig. 8. The tree has a single root representing the entire protocol system. At the next level, the principals are represented. The children of the principal nodes are the services, and their children represents the block structure of the services. Looking specifically at the Send service of the Sender principal, we see that the service has three direct descendants. These descendants represents the loop in the service and one atomic block on each side of the loop. The first of the nodes is the partition atomic which contains the partition pragmatic which is where the message sent by the framing protocol is divided into smaller smaller. The second node is the

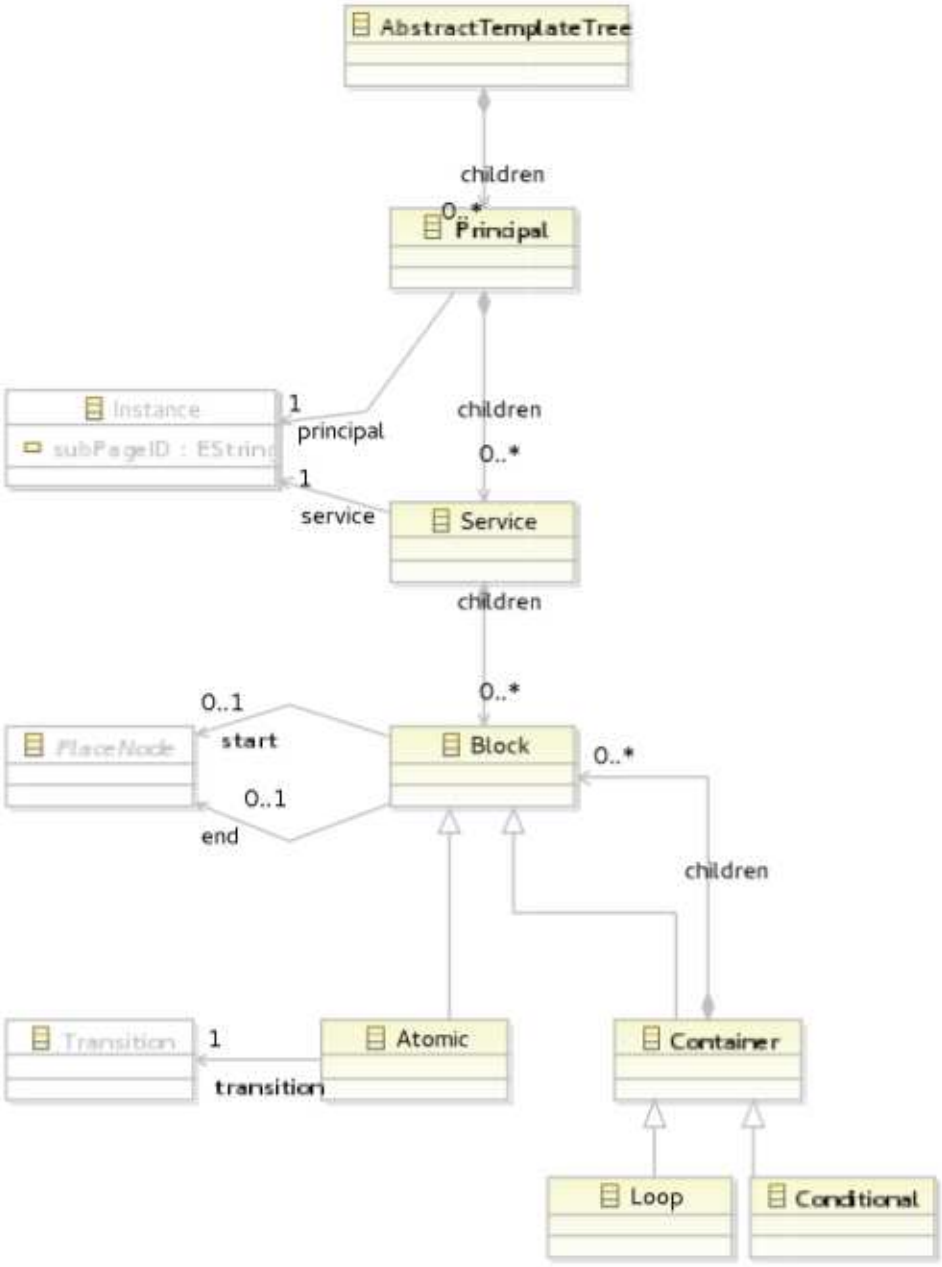

Fig. 7. Classes of the ATT

loop itself and the final node is the atomic after the loop which does not have any pragmatics and as such does not produce any code.

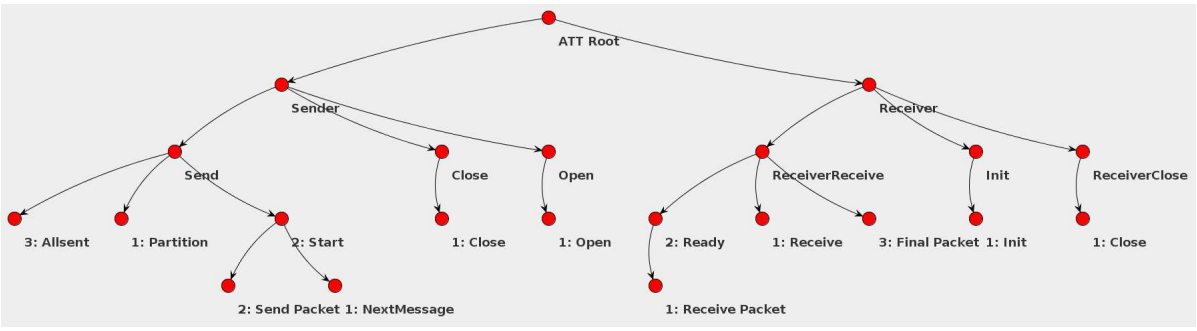

Fig. 8. ATT of the example

ATTs may be serialized and de-serialized to and from XML documents. This is done by using the XMLStream library and follows its conventions and file format. Necessary parts of the CPN model will also be serialized together with the ATT. PetriCode can also output a graphical representation of the ATT such as the one in Fig. 8.

3.4 Code Generation Module

The generation module is responsible for generating code from ATTs. In order to generate code from annotated CPNs, the pragmatics must be connected to code generation templates. This is done using the Binding class which is connected to Pragmatics(see Fig. 6). The bindings are produced by another DSL wich parses user provided template bindings and returns an object structure for the template bindings.

The code generation phase can be divided into two separate sub-phases. The first sub-phase is the code generation for each element in the ATT. A visitor visits each element in the ATT in no particular order. Each ATT node must implement the method generateCode(bindings) which finds and uses the appropriate templates to generate the corresponding code. The code is put in the field `text` in each of the elements. Other visitors makes sure that variables used are declared at the proper places for applicable platforms.

The second sub-phase in the code generation phase is to stitch together the generated code for each ATT node. This is done by a depth-first traversal of the ATT. For each node, when all the sub-nodes have been visited, the `%%yield%%` tag in the code generated for the node is replaced by the concatenation of the text field of all the immediate descendent of the node. When this has been done for each principal in the protocol, the code generation is complete and the code is written to the output directory.

Template bindings. In order to select the right code template for each pragmatic, the user supplies PetriCode with so-called template bindings. These bindings are supplied in the form of a DSL. The DSL allows the user to specify the

template and other necessary information about a template and how it should be applied. In the template bindings DSL, there are names, which can be anything as long as they are unique, followed by a pair of parenthesis. Inside the parenthesis a map is given which is the parameters of the binding. The possible parameters are described in Table 3.

Parameter	Description
pragmatic	The name of pragmatic that is bound by this binding.
template	The path to the template file referred to by this binding.
parameterStrategy	The strategy that should be used by the generator to supply the template with the appropriate parameters.

Table 3. The parameters for template bindings.

The available parameter strategies are `FROM_PRAGMATIC` and `CONDITIONALS`. The `FROM_PRAGMATIC` strategy simply splits the parameters given in a pragmatic based on ','. The resulting list is then added to the map of parameters with the key 'params'. The `CONDITIONALS` is used when a condition needs to be parsed. This strategy checks if the arguments of the pragmatics starts with the string "cond:". If it does, the first part of the arguments string, encapsulated in parenthesis, contains a boolean expression. The expression is expressed in a specialized lisp-like language language. The next two parts of the arguments should contain the outcomes for the cases where the expression evaluates to true and false.

Listing 5 shows two template bindings. The first binding is a binding for the $\langle\langle$Principal$\rangle\rangle$ pragmatic, which is used on the Sender and Receiver substitution transitions in Fig. 1. This is a container, which means that the generator should add the code generated to the principals children in the ATT to it. The other fields are pragmatic (which names the pragmatic) and template (which contains the file-name of the template). The second template binding binds $\langle\langle$send$\rangle\rangle$, which is placed on the Send Packet transition (See Fig. 3). after pragmatics derivation. This binding has the additional field parameterStratey. This field determines how the parameters to the template should be constructed.

Listing 5. Two examples of template bindings.

```
classTemplate(pragmatic: 'Principal',
    template: './groovy/mainClass.tmpl', isContainer: true)
send(pragmatic: 'send', parameterStrategy: FROM_PRAGMATIC,
    template: './groovy/sendMessageTCP.tmpl')
```

Custom Templates. When creating support for different platforms, adding new functionality through custom templates or overriding existing code generation new templates should be created and bound through template bindings.

Templates are Groovy Templates and processed by the Groovy SimpleTemplateEngine [3]. This means that the full Groovy programming language is available for the template creator, as well as a convenient notation.

4 Related Work

Many tools exist for generating software from models. Most of the tools, however, support only the generation of static parts of the code and some standard behaviour [8]. This does less than it could to help create robust software since the non-trivial parts are still written manually. However, some tools allows for generating more than structural parts of software. In this section we only consider tools that do full code generation where no manual coding is necessary.

Process-Partitioned CPNs (PP-CPNs) [9] have been used to automatically generate code for several purposes including protocol software. PP-CPNs are a restricted sub-class of CPNs. Code is generated from PP-CPNs by first translating the PP-CPN into a control flow graph (CFG), then translating the CFG into an abstract syntax tree for an intermediary language. The CFG is translated into another intermediary representation which is dependent on the target platform, and from this representation code is generated. In [9], PP-CPNs are used to model and obtain an implementation for the DYMO routing protocol using the Erlang programming language and platform. Both PP-CPNs and our modelling language are subclasses of CPNs. However, where we rely on pragmatics to control code generations PP-CPNs, rely on a restricted colour set and CPN structure to allow the generator to deduce the needed information. Our approach also models the environment of the services while PP-CPNs are geared more to just modelling the services. This allows us to represent the protocol at higher levels of abstraction on the protocol and principal levels as well as on the service level. It also allows us to define how the services should be called in a structured way.

There are several tools for modelling and generating protocol software based on the Specification and Description Language (SDL) [5, 2]. SDL is created for the purpose of modelling protocols, and is extensively used in the telecommunications industry. The IBM Rational SDL Suite (previously Tau SDL Suite and SDT) is among the most well known proprietary tools for SDL. The Rational SDL Suite supports code generation for SDL models to C and C++ code and also supports verification through model checking. Another SDL tool is Jade [14] that supports editing and analysis/verification of SDL models. Code generation for JADE is said to be in development. SDL Integrated Tool Environment (SITE) supports editing fg SDL models and code generation to Java and C++ code. SITE also supports some analysis of SDL models. SDL is a graphical language based on Finite State Machines (FSMs). This allows verification of protocols using model checking techniques. Compared to our approach, SDL is not as easily extensible as our approach.

Renew [12] is a tool that allows creation and execution of object-oriented Petri Nets. Renew supports several modelling formalisms. However, the for-

malisms are all based on various forms of Petri Nets. Renew supports Reference nets can be annotated with Java code and can be executed using a built-in simulator engine. The simulator can execute the nets incorporating the Java annotations in a headless mode so that no visualization will occur. This means that the simulations can be used as stand-alone programs. The simulation approach is in contrast to our code generation approach where code is generated and can be inspected and compiled as computer programs created with traditional programming languages.

GenERTiCA[19] is a tool aimed at embedded systems that generates code from UML models. In the generation process, the classes of the UML model are selected one by one, and templates are executed on each of them according to an XML mapping. It is not detailed in the article[19] on GenERTiCA how behaviour is modelled. However, from the description of the mapping file it is clear that there must exist templates for key behavioural actions. Many aspects of GenERTiCA are similar to our approach, although in a different domain. A key difference in the approaches is that GenERTiCA is based on an aspect oriented/object oriented approach. This limits the target platforms slightly, although the paper clearly states that aspect orientation need not be supported by the target language.

The Unified Modelling Language, and in particular state charts and sequence diagrams, has been used to model and generate code for protocols in several approaches [10, 19, 1, 13, 11]. Several tools exists for UML which support analysis and code generation in various ways. None of the methods surveyed, however, supports modelling the interface to other applications that is done in our approach in the service layer without switching to another diagram type such as class diagrams or. Also our pragmatics- and template-based approach allows us to give the user a great deal of flexibility be letting the user easily define custom pragmatics and templates.

5 Conclusions and further work

In this paper we have described a tool that can generate code from Coloured Petri Nets annotated with pragmatics. We have shown how this tools works by using the example of a simple communication protocol. The goal of our tool is to be able to generate code that is complete in the sense that no further code should be required to use the services our code provides. Another important goal has been to generate code that is readable and analysable for human programmers.

The input of the tool is an instance of a specific class of CPN models. A main goal of the tool and indeed the entire approach is that these models should be as descriptive in the sense that they can be used to describe the modelled system accurately on useful levels of abstraction.

In the future we will use the tool to evaluate our approach using a larger and more realistic examples and expand the range of available templates to other languages and platforms. Another item we plan to work on is to make our approach even more flexible by allowing the users to easily add custom pragmatic

patterns and placement conditions. We will also attempt to integrate the tool with other popular software development tools.

References

1. M. Alanen, J. Lilius, I. Porres, and D. Truscan. *On Modeling Techniques for Supporting Model Driven Development of Protocol Processing Applications*, pages 305–328. Springer, 2005.
2. F. Babich and L. Deotto. Formal methods for specification and analysis of communication protocols. *Communications Surveys Tutorials, IEEE*, 4(1):2–20, 2002.
3. Groovy. *Groovy Templates*. http://groovy.codehaus.org/Groovy+Templates.
4. Groovy. *Project Web Site*. http://groovy.codehaus.org.
5. ITU-T. Recommendation z.100 (11/99) specification and description language (sdl), 1999.
6. K. Jensen and L.M. Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009.
7. Kurt Jensen, LarsMichael Kristensen, and Lisa 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.
8. E. Kindler. Model-based software engineering; the challenges of modelling behaviour. In *Proceedings of the Second Workshop on Behavioural Modelling - Foundations and Application (BM-FA 2010)*. ACM electronic libraries, 2010.
9. 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.
10. Christian Kroiss, Nora Koch, and Alexander Knapp. Uwe4jsf: A model-driven generation approach for web applications. In *Proceedings of the 9th International Conference on Web Engineering*, ICWE '09, pages 493–496, Berlin, Heidelberg, 2009. Springer-Verlag.
11. P. Kukkala, V. Helminen, M. Hannikainen, and T.D. Hamalainen. Uml 2.0 implementation of an embedded wlan protocol. In *Personal, Indoor and Mobile Radio Communications, 2004. PIMRC 2004. 15th IEEE International Symposium on*, volume 2, pages 1158–1162 Vol.2, 2004.
12. Olaf Kummer, Frank Wienberg, Michael Duvigneau, Jörn Schumacher, Michael Köhler, Daniel Moldt, Heiko Rölke, and Rüdiger Valk. An extensible editor and simulation engine for petri nets: Renew. In Jordi Cortadella and Wolfgang Reisig, editors, *Applications and Theory of Petri Nets 2004*, volume 3099 of *Lecture Notes in Computer Science*, pages 484–493. Springer Berlin Heidelberg, 2004.
13. J. Parssinen, N. von Knorring, J. Heinonen, and M. Turunen. Uml for protocol engineering-extensions and experiences. In *Technology of Object-Oriented Languages, 2000. TOOLS 33. Proceedings. 33rd International Conference on*, pages 82–93, 2000.
14. C.L. Pereira, Jr. da Silva, D.C., R.G. Duarte, A.O. Fernandes, L.H. Canaan, C.J.N. Coelho, and L.L. Ambrosio. Jade: An embedded systems specification, code generation and optimization tool. In *Integrated Circuits and Systems Design, 2000. Proceedings. 13th Symposium on*, pages 263–268, 2000.
15. PetriCode. *Project Web Site*. http://kentis.github.io/nppn-cli/.

16. K. I.F Simonsen, L.M. Kristensen, and E. Kindler. Code Generation for Protocol Software from CPN models Annotated with Pragmatics. Technical Report IMM-Technical Reports-2013-01, Technical University of Denmark, DTU Informatics, January 2013. Available via `http://bit.ly/WwH2hf`.
17. Kent Inge Fagerland Simonsen. On the use of pragmatics for model-based development of protocol software. In *International Workshop on Petri Nets and Software Engineering*, 2011.
18. K.I.F. Simonsen and L.M. Kristensen. Towards a CPN-based modelling approach for reconciling verification and implementation of protocol models. In *Proc. of MOMPES'12*, LNCS. Springer, 2012. To appear.
19. Marco A. Wehrmeister, Edison P. Freitas, Carlos E. Pereira, and Franz Rammig. Genertica: A tool for code generation and aspects weaving. In *Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, ISORC '08, pages 234–238, 2008.
20. Michael Westergaard and LarsMichael Kristensen. The access/cpn framework: A tool for interacting with the cpn tools simulator. In Giuliana Franceschinis and Karsten Wolf, editors, *Applications and Theory of Petri Nets*, volume 5606 of *Lecture Notes in Computer Science*, pages 313–322. Springer Berlin Heidelberg, 2009.