View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Online Research Database In Technology

Technical University of Denmark

Code Generation for Protocols from CPN models Annotated with Pragmatics

Simonsen, Kent Inge; Kristensen, Lars Michael ; Kindler, Ekkart

Publication date:
2012

Link back to DTU Orbit

Citation (APA):
Simonsen, K. I., Kristensen, L. M., & Kindler, E. (2012). Code Generation for Protocols from CPN models Annotated with Pragmatics. Paper presented at 24th Nordic Workshop on Programming Theory (NWPT 2012), Bergen, Norway.

DTU Library
Technical Information Center of Denmark

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Code Generation for Protocols from CPN models Annotated with Pragmatics

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

[1] Department of Computer Engineering, Bergen University College, Norway
Email: {lmkr,kifs}@hib.no
[2] DTU Informatics, Technical University of Denmark, Denmark
Email: {kisi,eki}@imm.dtu.dk

Introduction. Model-driven software engineering (MDE) [3] provides a foundation for automatically generating software based on models. Models allow software designs to be specified focusing on the problem domain and abstracting from the details of underlying implementation platforms. When applied in the context of formal modelling languages, MDE further has the advantage that models are amenable to model checking [1] which allows key behavioural properties of the software design to be verified. The combination of formally verified models and automated code generation contributes to a high degree of assurance that the resulting software implementation is correct according to the verified properties.

Coloured Petri Nets (CPNs) [2] have been widely used to model and verify protocol software [4], but limited work exists on using CPN models of protocol software as a basis for automated code generation. In this paper we present an approach for generating protocol software from a restricted class of CPN models. The class of CPN models considered aims at being *descriptive* in that the models are intended to be helpful in understanding and conveying the operation of the protocol while at the same time being close to a verifiable version of the same model and sufficiently detailed to serve as a basis for automated code generation when annotated with *code generation pragmatics*. The purpose of the pragmatics is to address the problem that models with enough details for generating code from them tend to be verbose and cluttered.

Our code generation approach consists of two main steps starting from a CPN model annotated with pragmatics: the first step is to construct an *abstract template tree* (ATT). The ATT then directs the code generation in the second step in that each node of the ATT has associated code templates that are invoked to generate code. In the following we first introduce pragmatics and the class of CPN models considered using a small example of a unidirectional data framing protocol. Next, we use the framing protocol example to illustrate the two main steps of the code generation.

CPN model structure and pragmatics. A protocol system consists of a set of *principals* communicating over *channels*. This structure is reflected in the modular structure of the class of CPN models considered: there is a module for each principal and each channel, and our generation approach generates the code for each of the principals. The unidirectional data framing protocol used as an example in this paper consists of a sender principal and a receiver principal communicating over a FIFO channel. Each principal, in turn contains sub-modules for each of the service primitives (methods) that the principal provides to upper layer protocols or applications. To enable code generation, we introduce the concept of *pragmatics* which are syntactical annotations that can be associated with CPN model elements (e.g., places, transitions, and inscriptions) and are indicated between << >>. The pragmatics are either added *explicitly* by the modeller or are *implicit* in that they are inferred automatically derived from the structure of the CPN model. The pragmatics used in this paper fall into two categories: operation pragmatics and control flow pragmatics.

Figure 1 shows the CPN module representing the send method primitive provided by the sender in the framing protocol. Due to space limitations, we only sketch how code generation is performed at the method level and omit the principal and API levels (which are comparably simpler). The transition Send (top) annotated with an <<external>> pragmatic represent the entry point of the send method. The msg parameter specifies that the send method takes a message (msg) as parameter. Using CPNs for modelling protocols typically follows some general principles which clearly separate between control flow elements and data elements. Automatically identifying the control flow elements, however, is sometimes a bit tricky. Therefore, our approach assumes that the control flow pragmatics <<ID>> provides this information, which the modeller had in mind when designing the CPN model.

In the CPN model from Fig. 1, each incoming message is first partitioned (transition Partition) into a set of Outgoing Messages (fragments). Next, the control flow enters a loop (place Start) where each of the fragments are sent together with a flag indicating whether or not it is the last frame of the message. The loop terminates (transition AllSent) after the last frame has been sent and the send method returns. The latter is modelled by the Completed transition annotated with the <<return>> pragmatic.

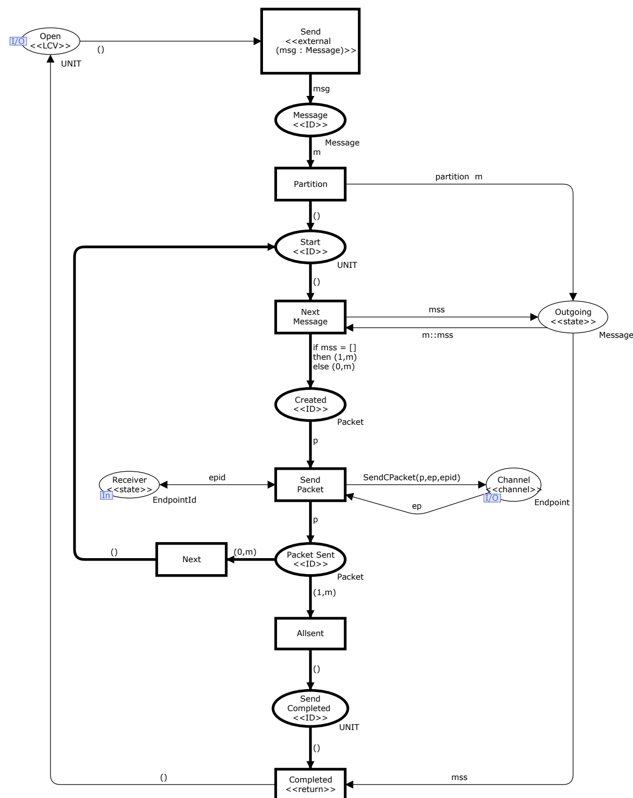

Figure 1: Send CPN submodule (sender principal).

Abstract template tree construction. The abstract syntax tree (ATT) for a CPN module is an ordered tree that has nodes corresponding to explicit and inferred pragmatics. The ATT is derived from the CPN model based on the pragmatics. The ATT represents the control structure of the CPN independently from a specific programming language. The nodes of the ATT is bound to code templates which are then applied during code generation. When constructing the ATT representing the part of the protocol shown in Fig. 1, the control flow path as determined by the <<ID>> pragmatic is traversed starting from the transition annotated with the <<external>> pragmatic. The <<external>> pragmatic is translated into a *container node* in the ATT which will contain all nodes for this module.

The implicit pragmatics are computed in a preprocessing stage prior to the actual ATT construction. The send module has some implicit pragmatics that are inferred during the traversal of the CPN module corresponding to operations that are to executed at different points along the control flow path of the principal. One example is <<partition>> which is determined by the `partition` function used in the arc expression on the arc from transition Partition to Outgoing Messages. Another example is a <<send>> pragmatic inferred from the model pattern used at the SendPacket transition for accessing the place Channel for sending a packet.

While traversing the CPN module for inferring derived pragmatics, the *block* structure of the CPN model is computed. This block structure characterises the control flow structures of the CPN, such as *branches*, *loops*, and *sequences*. This decomposition is, on the on hand side, used for inferring implict control flow pragmatics – e. g. tagging places with start and end loop pragmatics, where the end loop pragmatics holds a boolean condition for continuing the loop. On the other hand side, the block structure is represented as the ATT, which captures the control flow independently from any specific programming language or platform.

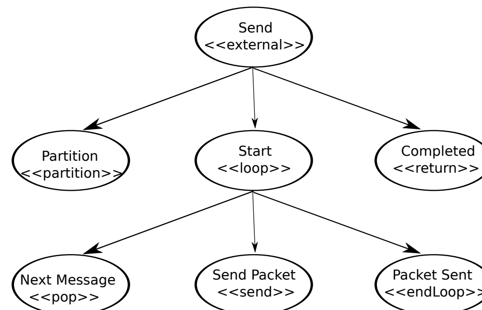

Figure 2: ATT for the send CPN submodule.

Figure 2 shows a simplified representation of the the ATT corresponding to the send CPN module in Fig. 1 (right). The <<external>> root node of the ATT contains nodes for partitioning the message, executing

a loop, and returning. It should be noted that sequences are not explicitly represented in the ATTs but given by the order of child nodes. The <<loop>> node contains nodes for extracting the next fragment, sending a packet, and ending the loop.

Code Generation. After the ATT has been created, code is generated by applying templates for each pragmatic on each node in the ATT. Below we illustrate how code can be generated implementing the framing protocol in the Groovy programming language. In this context, template texts are combined replacing the `%%yield%%` directive in container templates with the code generated for its child nodes.

As an example of a container template, the template for a Loop pragmatic for the Groovy language is given in Listing 1. The template creates a while-loop which continues while the `__LOOP_VAR__` variable is true. The body of the Loop is populated by replacing the `%%yield%%` directive with the code generated by the templates of the sub-nodes in the ATT. The `__LOOP_VAR__` is updated at the end of the loop by the <<endLoop>> pragmatic which is always present as the last child element of a loop. The <<send>> pragmatic is an example of an operation and is used to send a message over a channel. Listing 2 show the template for the Send pragmatic which requires two parameters: one is the name of the socket that the message should be sent on, and the other is the variable that holds the message to be sent.

Listing 3 shows the generated Groovy code corresponding to the loop in the send method. The code generated for the send pragmatic in the example above is highlighted in Listing 3.

Listing 1: Template for loops in Groovy.

```
%%VARS:__LOOP_VAR__%%
__LOOP_VAR__ = true
while(__LOOP_VAR__){
        %%yield%%
}
```

Listing 2: Template for sending a message.

```
${params[0]}.getOutputStream()
        .newObjectOutputStream()
        .writeObject(${params[1]})
%%VARS:${params[1]}%%
```

Listing 3: Generated code for the loop.

```
__LOOP_VAR__ = true
while(__LOOP_VAR__){
  def  m = OutgoingMessage.remove(0)
  if(OutgoingMessage.size() > 0){
    __TOKEN__ = [1,m]
  } else {
    __TOKEN__ = [0,m]
  }
  Receiver.getOutputStream()
         .newObjectOutputStream()
         .writeObject(__TOKEN__)
  __TOKEN__
  __LOOP_VAR__ = 1 == __TOKEN__[0]
}
```

Conclusion. The code generation approach presented in this paper is still work in progress. Prototypes have been implemented that have allowed initial application of the approach on the data framing protocol and a security protocol. Next steps in the development of the approach will be a formalisation of the concepts and application to larger examples of protocols. A key aspect of this is also the mapping from operations in the CPN models (which are written in Standard ML) into pragmatics and operations on the underlying platform. Here we proposes that operations and corresponding pragmatics can be grouped into packages according to their function in a manner similar to libraries in conventional programming languages. We propose to predefine a set of essential packages with operations that are common to most protocols. However, we acknowledge that it will not be possible to create a full set of operations for the protocol software domain. Therefore, we also provide a way for the modeller to define protocol specific operations. This provides our approach with a high degree of flexibility.

References

[1] C. Baier and J-P Katoen. *Principles of Model Checking.* MIT Press, 2008.

[2] K. Jensen and L.M. Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems.* Springer, 2009.

[3] S. Kent. Model Driven Engineering. In *Proc. of Integrated Formal Methods*, volume 2335 of *LNCS*, pages 286–298. Springer, 2002.

[4] L.M. Kristensen and K. Simonsen. Applications of Coloured Petri Nets for Functional Validation of Protocol Designs. In *Proc. of Advanced Course on Petri Nets*, LNCS. Springer, 2012. To appear.