Automatic Structure-Based Code Generation from Coloured Petri Nets: A Proof of Concept

Lars Michael Kristensen¹ and Michael Westergaard²

¹ Department of Computer Engineering, Bergen University College, Norway
lmkr@hib.no

² Department of Mathematics and Computer Science,
Eindhoven University of Technology, The Netherlands
m.westergaard@tue.nl

Abstract. Automatic code generation based on Coloured Petri Net (CPN) models is challenging because CPNs allow for the construction of abstract models that intermix control flow and data processing, making translation into conventional programming constructs difficult. We introduce Process-Partitioned CPNs (PP-CPNs) which is a subclass of CPNs equipped with an explicit separation of process control flow, message passing, and access to shared and local data. We show how PP-CPNs caters for a four phase structure-based automatic code generation process directed by the control flow of processes. The viability of our approach is demonstrated by applying it to automatically generate an Erlang implementation of the Dynamic MANET On-demand (DYMO) routing protocol specified by the Internet Engineering Task Force (IETF).

1 Introduction

The development of concurrent software systems is complex due to the rich behaviour introduced by concurrency, communication, and non-determinism. Coloured Petri Nets (CPNs) [9] and CPN Tools [2] (and formal modelling in general) have been widely used to address these challenges and construct formal and executable models of system designs with the aim of validating functional and performance properties prior to implementation [7]. Constructing a formal model yields important insight into the system design, and is a very helpful reference artefact when conducting a manual implementation of a software design. Even so, manual implementation is error-prone and time-consuming, making automatic code generation [8, Chap. 21] preferable in order to reduce the risk of introducing errors and to exploit the resources invested in model construction.

Despite the wide use of CPNs and high-level Petri Nets for modelling and design validation, we are aware of relatively few examples where CPNs have been used to automatically obtain an implementation of the final software system. This is in contrast to, e.g., the area of hardware design, where low-level Petri nets have been widely used to synthesise hardware circuits [17]. A simulation-based approach to automatic code generation from CPNs has been used in the projects reported in [13] and [10]. Here, the simulation code for the CPN model
generated by CPN Tools is extracted, and after undergoing automatic modifications, e.g., linking the code to external libraries, the generated simulation code is used as the system implementation. A simulation-based approach is also used in [14] to generate Java code from a high-level Petri net. The idea of [14] is to make a class diagram which outlines the classes and method signatures of the program. From this diagram, classes are generated where the method bodies are filled with simulator code. The advantage of a simulation-based approach is that it does not put any additional limitations on the class of models for which code can be generated. Furthermore, the direct use of the simulation code automatically ensures that the implementation is behaviourally equivalent to the underlying model. A main disadvantage is performance. Firstly, the execution speed is affected because each step in the execution of the program involves the computation and execution of enabled transitions (as done by a simulator) in order to determine the next state. Secondly, the approach ties the target platform to that of the simulator which may make the approach impractical for certain application domains due to resource consumption. As an example, the SML/NJ compiler used for the simulator in CPN Tools has a large memory footprint making it ill-suited for the domain of embedded systems. These disadvantages can to some extent be overcome using a state-based approach. Here, the state space of the model is used to control the execution of the program and determine the next state. This approach assumes that the state space is finite and small.

The disadvantages of simulation- and state-based approaches to code generation motivate our work on a structure-based approach. The key idea is to exploit structure in the CPN model, which can be naturally mapped to conventional programming language constructs. This has the advantage that the structure of the CPN model becomes clearly recognisable in the generated code, and that the generated code has a structure closer to code written by a human programmer. Furthermore, the code generated using a structure-based approach contains no simulator scheduler to control the execution, thereby improving performance, and the approach can be made target language independent. Exploiting structure in CPNs for code generation purposes is challenging since CPNs makes it possible to model control flow structures, message passing, and data access more abstractly than supported directly in most programming languages.

To address this, we introduce Process-Partitioned CPNs (PP-CPNs) which constitute a subclass of CPNs. PP-CPNs contain additional syntactical information and semantic restrictions that provide an explicit separation of process control flow, message passing, and access to data. This is used in a four phase code generation approach, where the choice of target language is deferred to the last two phases. Figure 1 shows the four phases in our structure-based code generation approach for translating a PP-CPN model into code in a target programming language. The first phase (1) translates a PP-CPN into a control flow graph (CFG) for each process subnet, extracting the control flow from the model. Nodes in a CFG represent statements and directed edges represent jumps in the control flow. The CFG may also be subject to static analysis, e.g., dead code elimination. In the second phase (2) the CFG is translated into an abstract