Project no.:	**ICT-FP7-STREP-214755**
Project full title:	**Quantitative System Properties in Model-Driven Design**
Project Acronym:	**QUASIMODO**

Deliverable no.: D3.6

Title of Deliverable: Code Generation from Untimed Specifications

Contractual Date of Delivery to the CEC:	**Month 24**
Actual Date of Delivery to the CEC:	**Month 24 (February 1, 2010)**
Organisation name of lead contractor for this deliverable:	**ESI/UT**
Author(s):	**Theo Ruys**
Participants(s):	**P02 ESI/UT**
Work package contributing to the deliverable:	**WP3: Implementation**
Nature:	**R+P**
Version:	**0.1**
Total number of pages:	**13**
Start date of project:	1 Jan. 2008 **Duration:** 36 month

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)

Dissemination Level

PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Abstract:

This deliverable reports on the progress of the work within W3.6: code generation from untimed specifications. We describe the P2J compiler which compiles a PROMELA model to an equivalent Java application.

Keyword list: untimed specification, code generation, PROMELA, Java, P2J

Contents

Bibliography

[1] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language (Third Edition)*. Addison-Wesley, 2000.

[2] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL — A Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of 4th DIMACS Workshop on Verification and Control of Hybrid Systems III*, LNCS 1066, pages 232–243. Springer, 1995.

[3] Mark Bijl. P2J – A Promela to Java compiler. Technical report, Formal Methods & Tools Group, University of Twente, Enschede, The Netherlands, 2004. Internal Report within the Master course 'Tool Architecture' (in Dutch).

[4] Greg Bollella, Ben Brosgol, Steve Furr, David Hardin, Peter Dibble, James Gosling, and Mark Turnbull. *The Real-Time Specification for Java (RTSJ)*. Addison-Wesley, Reading, MA, USA, 2000.

[5] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Sci. Comput. Program.*, 72(1-2), 2008.

[6] Gerard J. Holzmann. *The* SPIN *Model Checker – Primer and Reference Manual*. Addison-Wesley, Boston, Massachusetts, USA, 2004.

[7] IEEE. *IEEE Standard VHDL Language Reference Manual*, 2000. IEEE Std 1076.

[8] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language (Second Edition)*. Prentice Hall, Englewood Cliffs, NJ, 1998.

[9] Siegfried Löffler and Ahmed Serhrouchni. Creating Implementations from PROMELA Models. In *Proceedings of the 2nd Int. SPIN Workshop, Rutgers University, New Jersey, USA, August 5, 1996*, 1996. Available from `http://www.spinroot.com/`.

[10] Terence Parr. *The Definitive* ANTLR *Reference*. The Pragmatic Bookshelf, Raleigh, North Carolina, 2007.

[11] Theo C. Ruys. Concurrent Programming with Java. Slides of the 5th lecture on the course "Concurrrent and Distributed Programming", University of Twente, 8 March 2006, 2006.

[12] Don Syme, Adam Granicz, and Antonio Cisternino. *Expert F#*. Apress, 2007.

[13] Edwin Vielvoije. SPINs Promela to Java Compiler, with help from Stratego. Master's thesis, Delft University of Technology, Faculty of EEMCS, Software Engineering Research Group, September 2008.

[14] Eelco Visser. Stratego: A Language for Program Transformation Based on Rewriting Strategies. In *Proc. of the 12th Int. Conf. on Rewriting Techniques and Applications (RTA 2001), Utrecht, The Netherlands, May 2001*, LNCS 2051, pages 357–362. Springer, 2001.

[15] ANTLR.
`http://www.antlr.org/`.

[16] F# at Microsoft Research.
`http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/`.

[17] UPPAAL.
`http://www.uppaal.com/`.

Abbreviations

AAU: Aalborg University, DK

CFV: Centre Fèdèrè en Vèrification, B

CNRS: National Center for Scientific Research, FR

ESI: Embedded Systems Institute, NL

ESI/RU: Radboud University Nijmegen, NL

ESI/UT: University of Twente, Enschede, NL

RWTH: RWTH Aachen University, D

SU: Saarland University, D

1 Introduction

Model-Driven Development (MDD) is a software development technique in which the primary software artefacts are models providing a collection of views. Within MDD, programming is replaced by modelling. The question, however, is how to transform a high level model to efficient low-level programming code. We are thus interested in effective implementation mappings (code generation) from abstract models onto concrete platforms with guarantees that correctness properties established of the models also hold of the resulting implementation.

In this deliverable we sketch the context and progress of the subproject WP3.6: the code generation for untyped systems. Within the Quasimodo project, so far, only ESI (i.e., the University of Twente) has worked on WP3.6.

In Section 2 we describe the code generation problem that we try to solve. Section 3 describes our solution to this problem: the development of P2J, a compiler which translates PROMELA models to equivalent Java applications. Section 4 gives some further details on P2J. Finally, Section 5 lists several ideas for future work.

2 Problem

The objective of this subproject within WP3 is to develop a framework for code generation for untimed, concurrent specifications that

- is based on compiler generation techniques,
- is able to deal with absent information, abstractions and bottom-up constraints,
- is fully tool supported, and
- is applicable to industrially sized case studies.

3 Solution

From discussions with the Quasimodo industrial participant CHESS concerning the 'ChessWay' case study, we learned that – within this case study at least – the choice of both the source and target programming languages are not of vital importance. With respect to the target language, both F# [12, 16] and VHDL [7] have been mentioned as possible candidates. And with respect to the (source) modelling language, there was a preference for UPPAAL [2, 17], but only because the initial model for the 'Chessway' had been formulated in UPPAAL.

Given the relative indifference to source and target languages, we let the experience and expertise of the participants within this subproject (i.e. ESI/UT) be leading in the choice for both languages. As (source) modelling language we choose PROMELA, the specification language for the model checker SPIN [6]. As target language we choose Java [1]. There are several reasons for choosing Java as target language:

1. Java is a modern, simple, object-oriented programming language with automatic memory management (garbage collection).
2. Java is platform independent and is being used on many platforms including mobile telephones, small embedded systems, etc.
3. Java provides powerful standard libraries (also cross platform) for multi-threading, data structures, networking, GUIs, etc.

Given the source language PROMELA and the target language Java, we formulated additional design objectives and constraints for our translation scheme:

- The generated Java program should be semantically 'equivalent' to the PROMELA model. The set of possible runs of the Java program should be included in the set of possible runs of the PROMELA model.
- The code of the resulting Java applications should be readable in the same way as code produced by 'recursive descent parser generators' is readable. The connection with the PROMELA model should be clear. A Java programmer should be able to understand and maintain the generated Java code.
- Allow as much concurrency as possible. So not just a PROMELA simulator which randomly schedules processes. Consequently, a minimization of synchronisation and locks.
- Exploit Java's interface mechanism to allow different implementations for certain PROMELA constructs, e.g., channels, rendez-vous communication.

Related work Already in 1996, Löffler and Serhrouchni [9] developed a compiler for the PROMELA language. The target language was C. Their ambition of the compiler was modest: they simply wanted to ease the creation of test scenarios and a tool for rapid prototyping of valid protocol implementations. Instead of generating a stand-alone C application, their application

simply added C code to the `pan.c` verifier as generated by the SPIN program. In this way, the resulting `pan` program acted both as a verifier and implementation. The resulting C program was not multi-threaded: there was a single scheduler UNIX process. For the switching between the PROMELA proctypes a scheduler was used which selected the next active proctype.

Following the approach sketched in [11], Vielvoije [13] developed a prototype of a compiler for PROMELA to Java, using the Stratego toolset [14, 5]. Unfortunately, the complexity of the Stratego toolset hampered a productive and effective implementation trajectory. The resulting compiler was buggy and crashed on most PROMELA models. And for the small PROMELA models for which the compiler produced code, the resulting Java programs were quite restrictive: the application was simply a scheduler that executed statements from the different processes in an interleaving fashion. In this way the simulator SPIN was simulated.

Another attempt to compile PROMELA to Java was done by Mark Bijl [3] in an internal Master project at the University of Twente. Instead of using the scheduler approach as sketched above, his compiler translated each PROMELA proctype to a separate Java `Thread` object which would run independently of each other. Unfortunately, the scope of the Master project was rather limited and only resulted in a proof-of-concept: a small subset of PROMELA was supported and several translation schemes were implemented in an ad-hoc manner. The approach was promising, though.

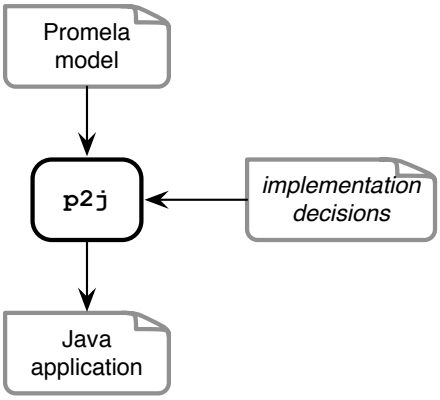

Figure 1: Toplevel architecture of P2J.

4 Design

We have named our compiler P2J, which stands for "PROMELA to Java".
Figure 1 gives a toplevel architecture of the P2J compiler. The input 'implementation decisions' relates to additional information that is needed to map the high-level, often non-deterministic choices of the PROMELA model to specific Java code. The P2J compiler is built using ANTLR [10, 15], a powerful and popular LL(*) compiler generator.

As described in the previous section, both [9] and [13] used the *simulator* approach to implement the PROMELA model: there is only a single process: a scheduler processes which executes the statements from the different proctypes. For a prototype for the application this might be acceptable, but this approach fails when the application is to be deployed in a multi-threaded and/or distributed environment. For such modern computing environments, the PROMELA proctypes should be mapped upon separate threads which run concurrently or even distributed. For P2J we therefore follow the same approach as was pioneered in [3]: each PROMELA proctype is mapped upon a Java `Thread`.

Figure 2 shows an example of the translation by P2J. The `init` process is mapped upon a class `Main` which starts all processes in its `main` method. The proctype `Foo` is mapped upon a class `Foo` which is a subclass of `Thread`. The method `run` contains the translation of the behaviour of the proctype `Foo`. The translator also generates a class `Globals` which contains instance variables for all global variables of the original PROMELA model.

Challenges It seems that the translation from PROMELA to Java should be rather straightforward. Both PROMELA and Java are related to the programming language C [8]. Most of PROMELA datatypes have equivalent datatypes in Java. Each PROMELA proctype can be represented by a Java thread. There are a few challenges, though.

- *granularity of languages*. In PROMELA, the statements (transitions) are atomic. In Java, the bytecode instructions are atomic. In this context, the assignment statement is problematic

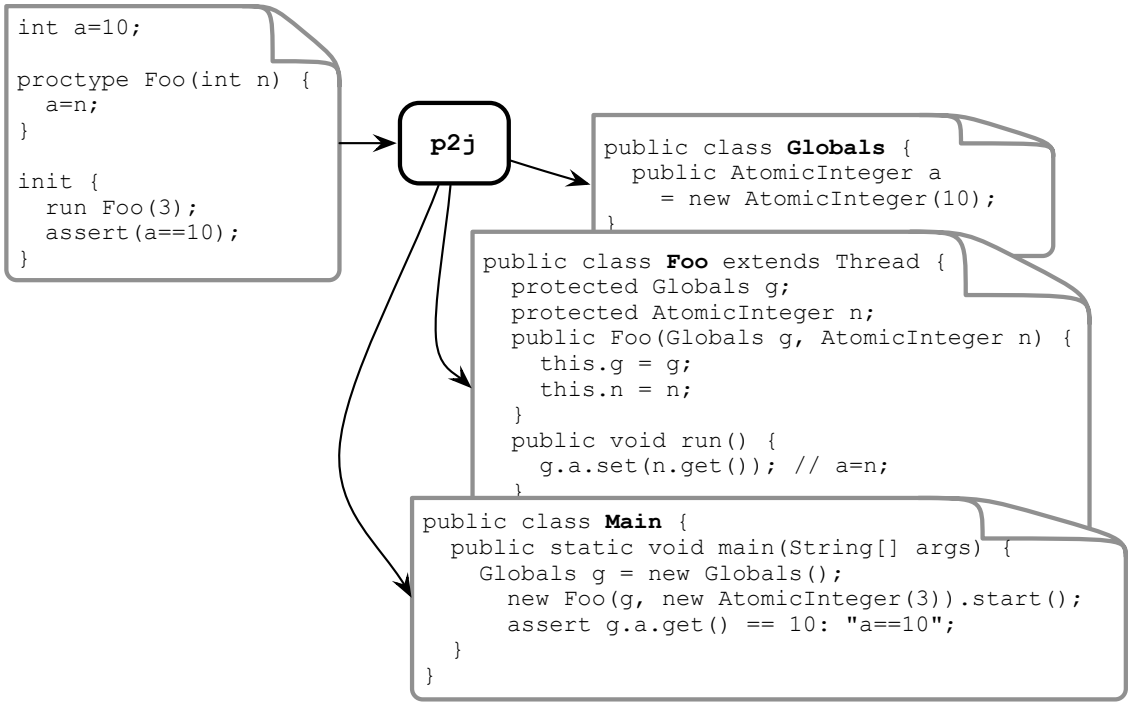

Figure 2: Example of the translation process of P2J.

due to the possibility of a data-race. It is no option to protect each assignment to a global variable with a global synchronized lock on the `Global` object, as this would seriously hamper the independent concurrency of the threads.

- *non-deterministic statements*. PROMELA supports two non-deterministic constructs: `if` and `do`. When a process is enabled, one of the executable guards of `if` (or `do`) is chosen non-deterministicly. While evaluating the guards, no other process/thread should be able to change the executability of the guards. Again it is no option to protect the beginning of each `if`/`do` statement using a global synchronized lock.

- *blocking of statements*. PROMELA statements that are not executable are blocked. A possible solution would be to use a busy-wait-loop, but this is not acceptable for obvious reasons. A better approach is to use Java's `wait` and `notify` methods to communicate the status of threads. However, now the question arrises: which synchronisation object should be used for this communication?

- *atomic and d_step*. PROMELA has two powerful constructs to combine several transitions into a single transition: `atomic` and `d_step`. Static analysis of the atomic sequences is needed to conclude whether other threads should be stopped while executing the atomic sequence.

It is clear that to maximize the concurrency of the threads we should be able to tune the level of synchronisation between the threads: not just a global wait/notify on a single object. Detailed (data-flow) information on the variables is needed to achieve this. And all with an important challenge in mind: how to make sure that the resulting Java code is still readable?

Java 5 to the rescue Since version 5, Java offers several powerful additional features that can be used for programming concurrent applications:

- *explicit locking*: instead of using synchronised blocks, an implementation of the interface `Lock` (and its methods) can be used to protect critical sections;
- *condition variables* allows for selective targeting of individual threads;
- *atomic variables*: data is read from or written to main memory in atomic operations: no explicit synchronisation locks are needed;
- `java.util.concurrent.*` does for threads what `java.util.Collection` did for data structures, e.g. the classes `BlockingQueue` and `Exchanger`.

Clearly these features are of great benefit when generating concurrent Java code from PROMELA models.

Status Due to a lack of manpower, we did not succeed in completing a working prototype of the P2J compiler yet. The front end (i.e. lexical analyser and parser for PROMELA) of the P2J compiler has been developed using ANTLR [10] and is feature complete. For walking the resulting AST, an ANTLR tree walker has been specified. The back end of P2J (i.e., the code generator) is still missing though. The current version of the P2J compiler can be retrieved from `http://ewi.utwente.nl/~ruys/p2j/`

5 Future Work

There still is a lot of work to be done before arriving at a practical compiler for full PROMELA. We foresee at least the following steps:

- Translation rules from the PROMELA language to Java should be formulated (1 man month).
- The code generator for the full PROMELA language has to be completed (2 man months).
- The compiler should be tested on industrial size PROMELA examples, especially on the Quasimodo's case studies (1 man month).

After the implementation of the compiler, there are several interesting directions for this work:

- Design an intermediate, process oriented programming language. Of course, this language should have features of both PROMELA and modern multi-threaded programming languages like Java. Let us call this language PIL, which stands for Process Intermediate Language.
- Design and implement a front end to compile PROMELA to PIL.
- Design and implemant a back end to compile PIL to Java.
- Design and implement back ends for other concrete target languages, e.g., F#, VHDL, etc.
- Add Java code extensions to PROMELA, similarly to PROMELA's C code extensions, e.g., `j_code`.

In a next phase we would also like to adopt Real Time Specifications for Java (RTSJ) [4] as a target langage for the code generation process. This would ease the generation of code for *timed* specifications.