Grant Agreement: 644047

INtegrated TOol chain for model-based design of CPSs

Design Principles for Code Generators

Technical Note Number: D5.1d

Version: 1.1

Date: Date: 2015

Public Document

http://into-cps.au.dk

Contributors:

Miran Hasanagić, AU
Peter Gorm Larsen, AU
Marcel Groothuis, CLP
Despina Davoudani, CLP
Adrian Pop, LIU
Kenneth Lausdahl, AU
Victor Bandur, AU

Editors:

Miran Hasanagić, AU

Reviewers:

Francois Hantry, CLE
Etienne Brosse, ST
Claes Dühring Jaeger, AI

Consortium:

Aarhus University	AU	Newcastle University	UNEW
University of York	UY	Linköping University	LIU
Verified Systems International GmbH	VSI	Controllab Products	CLP
ClearSy	CLE	TWT GmbH	TWT
Agro Intelligence	AI	United Technologies	UTRC
Softeam	ST		

Document History

Ver	Date	Author	Description
0.1	18-05-2015	Miran Hasanagić	Initial document version
0.2	18-05-2015	Miran Hasanagić	Initial document structure
0.3	25-08-2015	Miran Hasanagić	Initial description of the use cases
0.4	01-09-2015	Miran Hasanagić	Initial description of the design principles for the code generator in Overture
0.41	07-09-2015	Marcel Groothuis	Added the design principles of the 20-sim code generator
0.42	11-09-2015	Despina Davoudani	Write overview section for 20-sim (4C) & add flowcharts
0.43	15-09-2015	Adrian Pop	Add OpenModelica text
0.5	24-09-2015	Miran Hasanagić	New structure of document
0.51	25-09-2015	Miran Hasanagić	Added introduction, requirements and industrial needs
0.52	30-09-2015	Miran Hasanagić	Added design principles of Overture
0.53	01-10-2015	Adrian Pop	Added section about Modelica
0.54	15-10-2015	Victor Bandur	Added initial Related Work for projects
0.55	16-10-2015	Miran Hasanagić	Added initial Related Work for articles
0.56	18-10-2015	Miran Hasanagić	Added abstract and minor changes to the document
0.57	22-10-2015	Marcel Groothuis	Updated design principles 20-sim code generator
0.58	23-10-2015	Despina Davoudani	Added the section about the 20-sim modelling notation
0.59	26-10-2015	Marcel Groothuis	Finish the 20-sim requirements section
0.60	30-10-2015	Adrian Pop	Added OpenModelica text to Design Principles of the OpenModelica Code Generator
0.61	31-10-2015	Adrian Pop	Added OpenModelica text to Requirements
1.00	31-10-2015	Miran Hasanagić	Ready for internal review

1.01	05-12-2015	Miran Hasanagić	Changed parts based on internal review from CLP and ST
1.02	14-12-2015	Adrian Pop	Changed OpenModelica parts based on internal review comments
1.1	14-12-2015	Miran Hasanagić	Final version

Abstract

This deliverable describes the high level principles of the code generators which will be developed as part of the INTO-CPS project. Each of the tools Overture, 20-sim and OpenModelica will be extended with code generation capabilities from their respective modelling notations. Additionally the focus of this deliverable is on generating code which is runnable on platforms specified by the INTO-CPS case studies. Finally, an overview of related work is provided in the context of these INTO-CPS code generators.

Contents

1 Introduction **8**

2 Requirements and Industrial Needs **10**
2.1 INTO-CPS high level requirements 10
2.2 Industrial Needs 13

3 Modelling notations **16**
3.1 Vienna Development Method 16
3.2 20-sim 17
3.3 Modelica 19

4 Overview of the Code Generator Engines **23**
4.1 Overture 23
4.2 20-sim 25
4.3 OpenModelica 26

5 Design Principles of the Overture Code Generator **27**
5.1 Introduction 27
5.2 Main Goal 28
5.3 Mapping input and output 28
5.4 Computation and Communication elements 29
5.5 CPU code generation - Computational Element 29
5.6 BUS code generation 33

6 Design Principles of the 20-sim Code Generator **35**
6.1 Introduction 35
6.2 20-sim 35
6.3 20-sim 4C 38
6.4 INTO-CPS extensions 40

7 Design Principles of the OpenModelica Code Generator **41**
7.1 Introduction 41
7.2 OpenModelica Code Generation 41
7.3 INTO-CPS extensions 42

8 Related Work **44**
8.1 Code Generation in MBD 44
8.2 Model to PIC transformation 45
8.3 Model to PDC transformation 45

8.4 Software in CPSs 47
8.5 Formal methods - B method 47
8.6 Related Projects 48

A List of Acronyms **58**

1 Introduction

This deliverable describes the Code Generators (CGs) that will be developed as part of the INTO-CPS tool chain; a CG transforms a source language into a target language, while preserving the semantics of the source language. CGs are typically used to implement a model on an embedded system, e.g. a model is transformed to code written in the target language that can be compiled to run on an embedded system. Furthermore, the purpose of these INTO-CPS CGs with respect to the INTO-CPS tool chain, work-flow and industrial needs is explained.

The main goal of INTO-CPS is to support the Model-Based Development (MBD) of Cyber-Physical Systems (CPSs). Such systems include, for example, trains and cars, in which software is used to control actuators based on inputs from different sensors. From a modelling perspective a CPS can be divided into Discrete Event (DE) models, typically representing the software, and Continuous Time (CT) models, typically representing the physical system. The INTO-CPS tool chain will support the whole development process of CPSs, going from requirements, through the co-simulation validation phase, to the final implementation, as shown in Figure 1, where the focus of the CGs is marked with a red box.

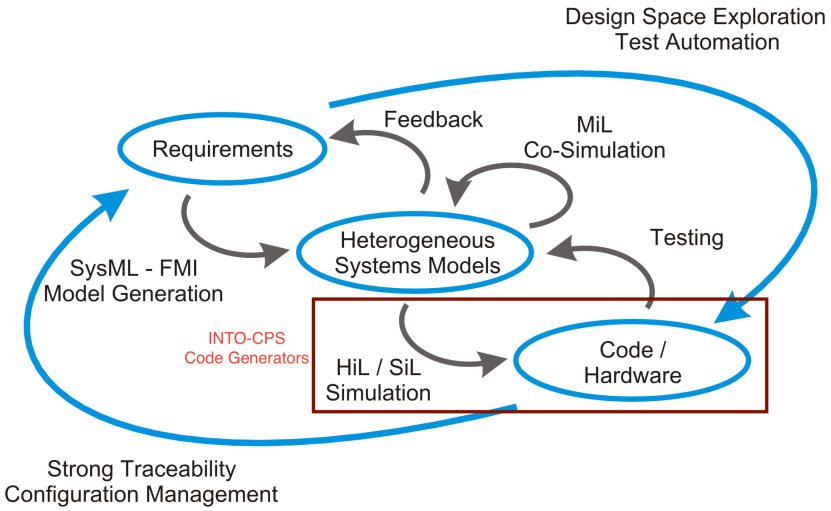

Figure 1: INTO-CPS tool chain work flow with the marking of the INTO-CPS CGs (in red)

In order for the INTO-CPS project to achieve its main goal, these CGs will play a key role by automating the process of implementing a software

model; implementing a model refers to that the code can be executed on a specific embedded system platform. The goal of the INTO-CPS CGs is to support the deployment of a software model to a hardware execution platform, as marked in Figure 1. Hence these CGs will support the MBD work flow of CPSs in INTO-CPS. So the INTO-CPSs CG can be used as part of realising the DE models of a CPS, in order to be executed on an embedded system. For this reason the INTO-CPS CGs described in the deliverable, focus on how the modelling languages of the INTO-CPS baseline tools, which are used for the discrete part of a CPS, can be translated to executable code for specific platforms. Hence these INTO-CPS CGs are developed as enhancements of the INTO-CPS baseline tools Overture [OCT07], 20-sim [Con13] and OpenModelica [Fri04].

The advantage of using code generators as a means of implementing a model is twofold. First, it automates the transformation between a validated and/or verified model, which may help to avoid introducing semantical differences between the model and the code, when compared to implementing the model manually. Second, development time can be reduced, since the corresponding code can be generated automatically from the model, enabling fast prototyping. A CG should ideally be proved/verified, or at least double cross checked/verified. It shall be noted that the transformations will not be formally verified for the INTO-CPS CGs. However, it will be argued why the INTO-CPS CGs, when generating code from a discrete model, will preserve the semantics by support of both test cases and providing a connection with the semantics work of INTO-CPS carried out as part of INTO-CPS Work Package (WP) 2.

In order to address the details of both the INTO-CPS requirements for the CGs, as well as the details of the high level design principles of the INTO-CPS CGs, the remainder of this document is structured as follows. First section 2 presents both the high level requirements and the specific industrial needs for the INTO-CPS CGs. Afterwards, sections 3 and 4 provide a brief overview of the modelling languages and the existing three code generators for the involved baseline tools, respectively. Then sections 5, 6 and 7 describe the design principles of each of the code generators separately. Finally, section 8 provides an overview of related work.

2 Requirements and Industrial Needs

This section refers to all the requirements and industrial needs for the INTO-CPS CGs. First section 2.1 provides an overview of all relevant INTO-CPS high level requirements from the INTO-CPS requirements report D7.3 [LPH+15], with respect to the baseline tools which have to satisfy them. Afterwards, section 2.2 describes the industrial needs with respect to the INTO-CPS high level requirements.

2.1 INTO-CPS high level requirements

The high level requirements from the INTO-CPS requirements report D7.3 [LPH+15] with focus on the INTO-CPS CGs are presented below and divided between the different baseline tools in order to clarify the requirements for each of the INTO-CPS baseline tools. However, *all the INTO-CPS code generators will fulfil the requirement **0089**, which states that requirements must be linked down to the code.*

2.1.1 Overture

*The code generator for the Overture platform will fulfil the requirements **0038** and **0086**.*

Requirement **0038** states that the Overture CG must generate code for a subset of VDM-RT models; the supported subset will be identified as a balance between the industrial needs identified by the INTO-CPS case studies and technological possibilities. The specific target code and platforms are described below with respect to the case study requirements.

Requirement **0086** states that the Overture CG must be capable of exporting implementations which conform to the INTO-CPS FMU format. The Overture CG will fulfil this requirement by wrapping the generated code in the Functional-Mock-up Interface (FMI) [Blo14], which also is described in section 2 in deliverable D4.1d [LLW+15]. This will be achieved by automatically generating the model description definition as required by the FMI standard. Hence this requirement enables the generated code to be used as part of the Co-simulation INTO-CPS Co-simulation Orchestration Engine (COE), which is described in the deliverable D4.1d [LLW+15], in order to support Software-in-the-Loop (SiL) simulation. Additionally, it will

be investigated how to use 20-sim 4C in order to support running models on different hardware platforms, hence the code generator should agree with the requirements of 20-sim 4C.

2.1.2 20-sim

*The 20-sim code generation will fulfil the requirements **0040**, **0042**, **0087** and partly **0010**. 20-sim 4C code generation should fulfil requirement **0042**.*

Requirement **0040** states that it must be possible to generate code (e.g. C or C++) from the discrete notation used in the 20-sim tool. This means that 20-sim should provide two code generation templates: ANSI-C and C++. Furthermore, 20-sim should support all allowed SIDOPS+ language elements for discrete submodels in the generated code.

Requirement **0010** states that the 20-sim tool must provide an INTO-CPS FMI tool wrapper that is compliant with the COE. Part of this requirement is the option to export the model as a standalone FMU for co-simulation.

Requirement **0087** states that code generated from the INTO-CPS simulation tool 20-sim must be exportable in the INTO-CPS FMU format. This requirement also requires support for exporting a model as a standalone FMU. The FMI export option to generate a standalone FMU is implemented in 20-sim as a code generation template to fulfil this part of the requirement. The toolwrapper approach still requires an exported FMU with model-specific information. A modified version of the existing 20-sim code generation feature will be used as basis for the toolwrapper FMU export.

Requirement **0042** states that it must be possible to generate a HiL configured FMU from an existing 20-sim model FMU using 20-sim 4C. Hardware-in-the-Loop (HiL) simulation refers to running part of the simulation on real hardware. It is typically used to run a simulation model of a real system on a PC connected to a real control system (hardware) to test this control system. Timing of the simulation model is important since it is connected to the real control system. The simulation model should run in real-time. To achieve this, these simulation models are typically exported as C/C++ code with real hardware I/O interfaces to communicate with the real control system. This means that the generated code must support real-time execution and that the code can interface with drivers. The generated 20-sim / 20-sim 4C code should therefore not contain any constructions that prevent real-time execution or make the execution of the code undeterministic.

Code generation requirements from the 20-sim perspective
This section explains the guidelines for code generation from a 20-sim model. 20-sim does currently not support model-to-code export for all possible 20-sim models. Only a subset of the supported 20-sim modeling language elements can be exported as code.

The original goal for the 20-sim code generator was to export control systems into ANSI-C code to run the control system under a real-time operating system. As a consequence, 20-sim currently only allows code generation for discrete time submodels or continuous time submodels using a fixed step integration method. Other language features that are not or only partly supported for code generation are: file I/O, calls to external code (DLL, Matlab), variable delay blocks, event functions.

2.1.3 OpenModelica

*The OpenModelica code generator will fulfil the requirements **0039**, **0088** and partly **0009**.*

For the requirement **0009** OpenModelica will provide an FMI wrapper to access its capabilities. Modelica models can be loaded and code can be generated from them. Code generation with debugging capabilities can be selected and then OpenModelica can be used for debugging.

For requirement **0039**, for embedded systems or real-time systems, one needs to be able to generate ANSI-C code from a Modelica model or a part of the Modelica model. While currently OpenModelica generates C code by default there are a lot of things to consider when generating code for real-time or embedded systems.

For requirement **0088**, in order to enable code generated from simulation models to be included in a co-simulation (i.e Software in the Loop (SIL)) it is required that it adheres to the INTO-CPS FMI standard.

The OpenModelica tool must provide an INTO-CPS standalone FMUs and an FMI tool wrapper that is compliant with the COE. OpenModelica will use 20-sim 4C to support running models on different hardware platforms so the C code generation should agree with the requirements of 20-sim 4C.

2.2 Industrial Needs

This section describes the relationship between the industrial needs and the high-level INTO-CPS requirements described above.

The industrial needs described below are related to high level INTO-CPS requirements described above, in accordance with the case study owners. These industrial needs are described in the deliverable D1.1a [HLB+15]. This ensures that the dependencies of each case study on the INTO-CPS high-level requirements are satisfied.

These industrial needs have been investigated with a focus on the requirements for automating the translation from model to implementation. In order to clarify the different requirements that the code generators in this delivery must fulfil, the relevant INTO-CPS case study owners have been interviewed with respect to the following needs: target platform, target language, need for distribution of the computation and additional non-functional requirements of the generated code. However, for more details on each of needs presented below can be found in the case study deliverable D1.1a [HLB+15].

Three of the four case studies plan to use code generation for their case study. The following can be summarised below from the case study deliverable D1.1a [HLB+15]. Subsequently the needs of each case study owner is described separately in order to provide an overview of the concrete needs for the INTO-CPS CGs for each case study.

2.2.1 Agro Intelligence - Agricultural case study

Overview of industrial needs:

- **Target Platform:** This case study has the need for support for two hardware platforms. In need **AI_1**, which is for the Overture CG, the target platform is an embedded Linux system. Additionally, in need **AI_2**, which is for the 20-sim CG, the target platform is a B&R PLC.
- **Target language:** From need **AI_1**, the target language is C++, structured in the Gang-of-Four (GoF) state pattern [GHJV95]. In addition, from need **AI_2**, the target language is C/C++ software following the PackML standard implemented in BRDK_MU.
- **Distribution modelling:** This case study may model distribution using VDM-RT. As a result, it may be investigated how the distribution features of VDM-RT can be supported by the Overture CG.

- **Additional comments:** **AI_6** requires that requirements can be linked down to the code.

Reference to high level requirements: Needs **AI_1**, **AI_2** and **AI_6** are related to requirements **0038**, **0039** and **0089**, respectively.

2.2.2 ClearSy - Railways case study

Overview of industrial needs:

- **Target Platform:** In need **Cle_12** the target platform is set to be the Pic 32 micro controller.
- **Target language:** From need **Cle_12** the target language is either C code or a binary (HEX).
- **Distribution modelling:** This case study plans to model distribution using VDM-RT. As a result, it may be investigated how the distribution features of VDM-RT can be supported by the Overture CG.
- **Other notes:** The generated code has to comply with safety critical standards in accordance with need **Cle_12**. Additionally, this case study has timing requirements, hence timing requirements between a model and implementation may be investigated.

Reference to high level requirements: Need **Cle_12** is related to requirements **0038**, **0039** and **0040**.

2.2.3 United Technology Research Center - Building case study

Overview of industrial needs:

- **Target Platform:** No platform is specified currently.
- **Target language:** The target language is determined to be C/Embedded C code in requirement **UTRC-Req-012**.
- **Distribution modelling:** This case study plan to model distribution using VDM-RT. As a result, it may be investigated how the distribution features of VDM-RT can be supported by the Overture CG.
- **Other notes:** No further notes.

Reference to high level requirements: The requirements **UTRC-Req-012** is related to the requirements **0038**, **0039** and **0040**.

2.2.4 TWT - Automotive case study

Currently the automotive case study does not plan to use code generators.

3 Modelling notations

This section introduces the modelling notations which are the inputs for the INTO-CPS CGs. Subsequently the subsections 3.1, 3.2 and 3.3 present the modelling notations of Overture, 20-sim and OpenModelica, respectively.

3.1 Vienna Development Method

The Vienna Development Method (VDM) is a formal method that enables the specification, modelling and evaluation of software systems. This formal method evolved from the formal notation VDL (Vienna Definition Language) [FLV08], which was named Vienna Development Method in 1973 [Jon99], and which has been validated in industrial projects [FLS08]. A model of a system to be developed is expressed as a formal VDM model, and its validity is ensured by tool support. Currently there exist three dialects of VDM, *VDM-SL*, *VDM++* and *VDM-RT*, which are introduced below. For the INTO-CPS CG for Overture the input notation is the dialect VDM-RT. However, VDM-RT is an extension to the other two dialects, hence all three dialect are introduced below.

The *VDM-SL* dialect enables the specification of functional aspects of sequential systems, and the formal definition of the language's semantics is ISO standardised [PL92, LH$^+$96, FL09].

The *VDM++* dialect is an extension of VDM-SL which enables object-oriented modelling, which means that a system is modelled as a collection of classes [FLM$^+$05]. Furthermore, it introduces the possibility to model active classes through annotations consisting of concurrency and synchronisation elements [FLM$^+$05]. Later, VDM++ was extended with functionality to analyse real-time behaviour of models with respect to time. The extension was called "VDM++ In a Constrained Environment" (VICE).

The *VDM-RT* dialect is an extension of VDM++ which enables the modelling of *distributed* real time systems [LFW09]. The VICE extension was not sufficient for modelling distributed systems [Ver05]. The extensions in VDM-RT were proposed to enable modelling of distributed systems by introducing notions of computational resources, communication lines, global time, time delay and asynchronous operations [VLH06, HV10]. This method has been validated by several case studies [FLTV07, VL07, SN07, Ver09, Wol08]. This

extension also introduces the `system` class definition, in which distributed objects can be created and deployed to computational elements called `CPU`s. In order for these `CPU`s to communicate they need to be connected by a `BUS` inside the `system` definition.

3.2 20-sim

Systems can be modelled in 20-sim using a variety of modelling formalisms:

- Block diagrams
- Bond graphs
- Iconic diagrams
- Mathematical equations
- System descriptions (state space, transfer function)

Different formalisms can be freely combined within one model (mixed model).

Graphical models in 20-sim are composed from prebuild library blocks or custom-made blocks. These blocks are called submodels. The submodels are implemented either using a graphical representation or as an equation implementation.

Using graphical implementations inside submodels allows for hierarchical modelling. 20-sim supports unlimited levels of hierarchy in the model. The highest hierarchical levels in the model typically consist of graphical models (state space models, block diagrams, bond graphs or components). The lowest level in the hierarchy is always formed by equation models written in the SIDOPS+ language.

3.2.1 The SIDOPS Language

SIDOPS (Structure Interdisciplinary Description Of Physical Systems) is a computer language developed for the description of models and submodels of physical systems [Bro90]. It is designed to express bond-graph models that describe domain-indenpendent engineering systems. 20-sim uses the SIDOPS+ version of the language, the key features of which are discussed below.

SIDOPS+ [BB97] enhances the support for organising complex systems as a hierarchy of submodels, by separating the interface of a model from its

specification. This enables the creation of different specifications for one interface.

In addition, SIDOPS+ supports different representations of model descriptions within three abstraction levels [BB97]:

- At the *technical component* level, models describe networks of devices which are represented by component graphs.
- At the *physical concept* level, models capture the physical processes of a system and can be expressed by using graphical formalisms.
- At the *mathematical* level, models provide the quantitative description of the physical processes, written in form of acausal equations or sequential statements (computer code) that calculate the output variables from the input variables.

All representations are port-based networks, meaning the connection points between the model elements is the location where exchange of information (signals) or power happens. As a result, it is possible to map one representation to another without losing consistency.

Moreover, SIDOPS+ provides support for systems that consist of a continuous-time part and a discrete-time part by offering special functions to determine the sample interval for discrete-time variables that are linked through equations and to create a continuous signal out of a discrete input signal.

The general layout of an equation model in the SIDOPS+ language is shown in listing 1. Listing 2 shows an implementation in SIDOPS+ code of the Lotka Volterra model, described in section 3.3.1.

Listing 1: SIDOPS+ equation model layout

```
constants
        "do not change during or in between simulation runs"
parameters
        "can only be changed after the simulation has been
           stopped"
variables
        "change during simulation runs"
initialequations
        "are calculated once, before the simulation run"
code
        "equations calculated sequentially at every
           simulation step"
equations
        "standard equations calculated at every simulation
           step"
```

```
finalequations
        "are calculated once, after the simulation run"
```

Listing 2: Example SIDOPS+ model

```
parameters
  real g_r = 0.04;  "Natural growth rate for rabbits"
  real d_rf = 5e-5; "Death rate of rabbits due to foxes"
  real d_f = 0.09;  "Natural death rate for"
  real g_fr = 0.1;  "Efficiency in growing foxes from
     rabbits variables"
  real rabbits;     "Rabbits with start population 700"
  real foxes;       "Foxes with start population 10
     equations"
equations
  ddt(rabbits,700) = g_r * rabbits - d_rf * rabbits * foxes;
  ddt(foxes,10) = g_fr * d_rf * rabbits * foxes - d_f *
     foxes;
```

3.3 Modelica

Modelica [FE98], [Fri04] is an object-oriented, equation based language to conveniently model complex physical systems containing, e.g., mechanical, electrical, electronic, hydraulic, thermal, control, electric power or process-oriented subcomponents. The Modelica language supports continuous, discrete and hybrid time simulations.

The Modelica language has been designed to allow tools to automatically generate efficient simulation code with the main objective of facilitating exchange of models, model libraries, and simulation specifications. The definition of simulation models is expressed in a declarative manner, modularly and hierarchically. Various formalisms can be expressed in the more general Modelica formalism. In this respect Modelica has a multi-domain modeling capability which gives the user the possibility to combine electrical, mechanical, hydraulic, thermodynamic, etc., model components within the same application model. Compared to most other modeling languages available today, Modelica offers several important advantages from the simulation practitioners point of view:

- *Object-oriented mathematical modeling.* This technique makes it possible to create model components, which are employed to support hierarchical structuring, reuse, and evolution of large and complex models covering multiple technology domains. A general type system that unifies objectorientation, multiple inheritance, and generics templates

within a single class construct. This facilitates reuse of components and evolution of models.

- *Acausal modeling* based on ordinary differential equations (ODE) and differential algebraic equations (DAE) together with discrete equations forming a hybrid DAE. There is also ongoing research to include partial differential equations (PDE) in the language syntax and semantics.
- *Multi-domain modeling* capability, which gives the user the possibility to combine electrical, mechanical, thermodynamic, hydraulic etc., model components within the same application model.
- *A strong software component model*, with constructs for creating and connecting components. Thus the language is ideally suited as an architectural description language for complex physical systems, and to some extent for software systems.
- Visual *drag & drop and connect composition of models* from components present in different libraries targeted to different domains (electrical, mechanical, etc).

The language is strongly typed and declarative. The Modelica component model includes the following three items: a) components, b) a connection mechanism, and c) a component framework. *Components* are connected via the *connection mechanism* realized by the Modelica system, which can be visualized in connection diagrams. The *component framework* realizes components and connections, and ensures that communication works over via the connections. For systems composed of *acausal* components with behavior specified by equations, the direction of data flow, i.e., the *causality* is initially unspecified for connections between those components. Instead the causality is automatically deduced by the compiler when needed. Components have well-defined *interfaces* consisting of ports, also known as *connectors*, to the external world. A component may internally consist of other connected components, i.e., *hierarchical* modeling is possible. Figure 2 shows a hierarchical component-based modeling of an industry robot.

3.3.1 An example Modelica model

The following is an example Lotka Volterra Modelica model containing two differential equations relating the sizes of rabbit and fox populations which are represented by the variables rabbits and foxes: The model was independently developed by Alfred J Lotka (1925) and Vito Volterra (1926): The

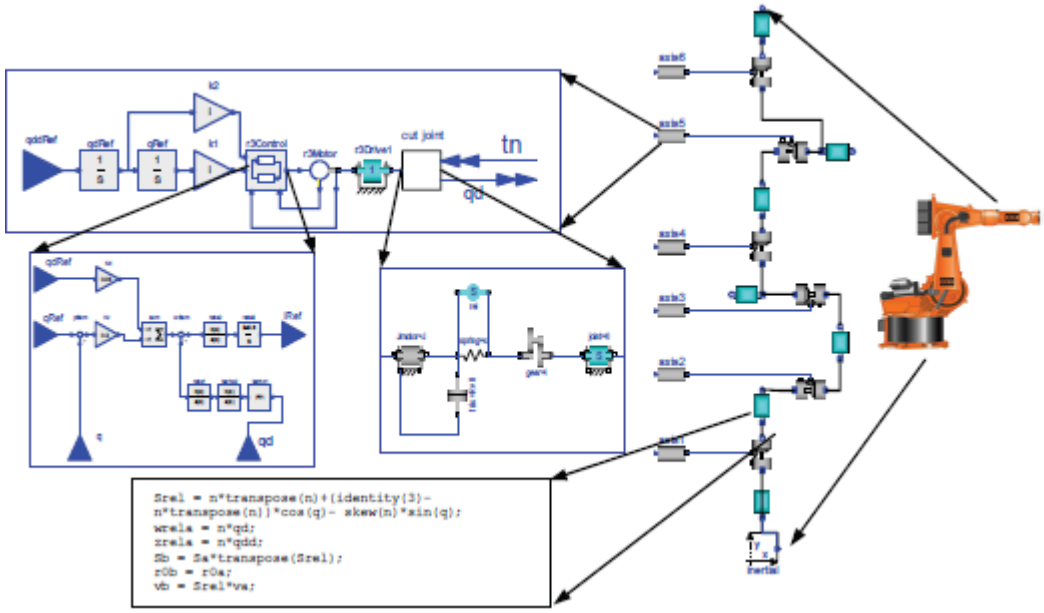

Figure 2: Hierarchical model of an industrial robot, including components such as motors, bearings, control software, etc. At the lowest (class) level, equations are typically found.

rabbits multiply (by breeding); the foxes eat rabbits. Eventually there are enough foxes eating rabbits causing a decrease in the rabbit population, etc., causing cyclic population sizes. The model is simulated and the sizes of the rabbit and fox populations are plotted in Figure 3 as a function of time.

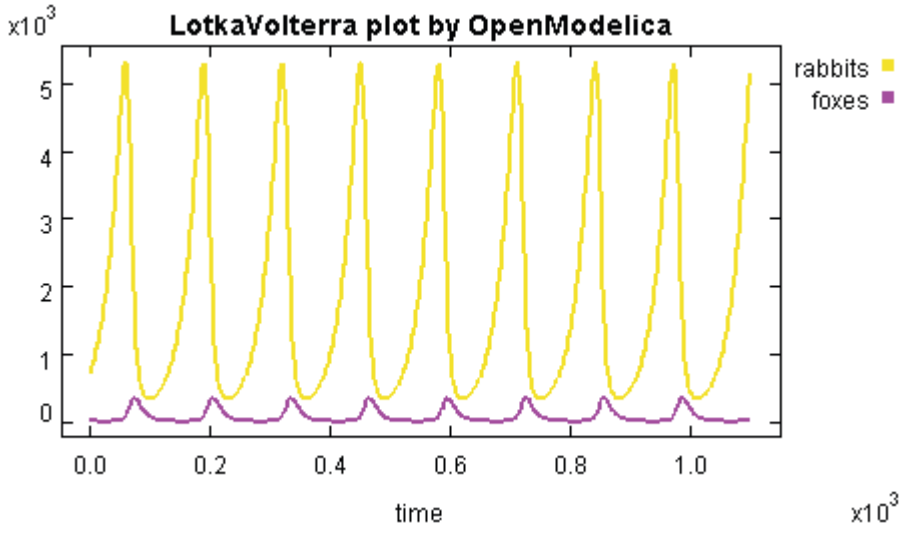

Figure 3: Number of rabbits prey animals, and foxes predators, as a function of time simulated from the predator-prey LotkaVolterra model.

The LotkaVolterra model as Modelica code is given below. The notation `der(rabbits)` means time derivative of the rabbits (population) variable.

```
model LotkaVolterra
  parameter Real g_r =0.04 "Natural growth rate for
      rabbits";
  parameter Real d_rf=5e-5 "Death rate of rabbits
     due to foxes";
  parameter Real d_f =0.09 "Natural death rate for
     foxes";
  parameter Real g_fr=0.1 "Efficiency in growing
     foxes from rabbits";
  Real rabbits(start=700) "Rabbits with start
     population 700";
  Real foxes(start=10) "Foxes, with start
     population 10";
equation
  der(rabbits) = g_r*rabbits - d_rf*rabbits*foxes;
  der(foxes) = g_fr*d_rf*rabbits*foxes - d_f*foxes;
end LotkaVolterra;
```

4 Overview of the Code Generator Engines

The following three subsections 4.1, 4.2 and 4.3 provide an overview of the three platforms Overture, 20-sim and OpenModelica, respectively. Additionally, each subsection provides a general description of the current code generation capabilities of each platform.

4.1 Overture

Overture [LBF+10] is a open-source platform which supports the construction and analysis of formal models described in VDM. It has a syntax and type checker, an interpreter to evaluate executable VDM models and a debugger. This tool is open-source and based on Eclipse, and the goal of the effort is to provide an industrial-strength tool [LL09, JLL13]. Since it is open-source, it enables researchers to extend and experiment with both the tool itself and the VDM language dialects [LBF+10], which are described in section 3.1. Subsequently subsection 4.1.1 presents a platform for code generation in Overture, which will be used as part of developing the INTO-CPS Overture CG. Afterwards subsection 4.1.2 summarises the current code generation possibilities in Overture using this platform for code generation.

4.1.1 Code Generation Platform

A VDM-RT model is described as text. Overture represents such a VDM-RT text model internally as an Abstract Syntax Tree (AST), and it will be referred to as a VDM AST subsequently. In order to support reusability when code generating a VDM AST to different language semantics, a Code Generator Platform (CGP) has been developed.

An overview of the CGP is shown in Figure 4. As the figure illustrates, the CGP takes a VDM AST as input and creates an Intermediate Representation (IR) tree. This IR preserves the semantics of the VDM AST, but is independent of both the VDM AST and the target language. So as described and required in the Description of Action (DoA) [Con14], a VDM-RT model is transformed to an IR.

Transformations, as seen in the Figure 4, can be applied to the IR tree in order to make it straightforward for the code generation backend to target a specific language. An relevant observation in [JLC15] is the possibility

to reuse parts of the transformation, when target languages face some of the same challenges with respect to code generation, which is discussed in [JLC15] for Java and C++, which both are part of the Object-Oriented imperative programming language. As an example of transformation and reuse of it is that a VDM-RT model has `operations` and `functions`, while Java and C++ only have methods. Therefore a transformation exists that converts `operations` and `functions` into methods in the IR, which both Java and C++ backends can be use for code generation. This is an example of a transformation of the IR shown in Figure 4. The IR undergoes an ordered sequence of such transformations to transform the IR to the IR' shown in Figure 4. Hence the IR transformations are used to eliminate constructs that are difficult to code generate by replacing them with other constructs that are easier to code generate.

The current backend in the CGP uses a template-based syntax transformation language to transform the IR to Java code. Essentially, the IR is transformed to make it closer to the target language representation, while preserving the semantical meaning of the VDM AST. More details about the general functionality of the CGP is described in [JLC15].

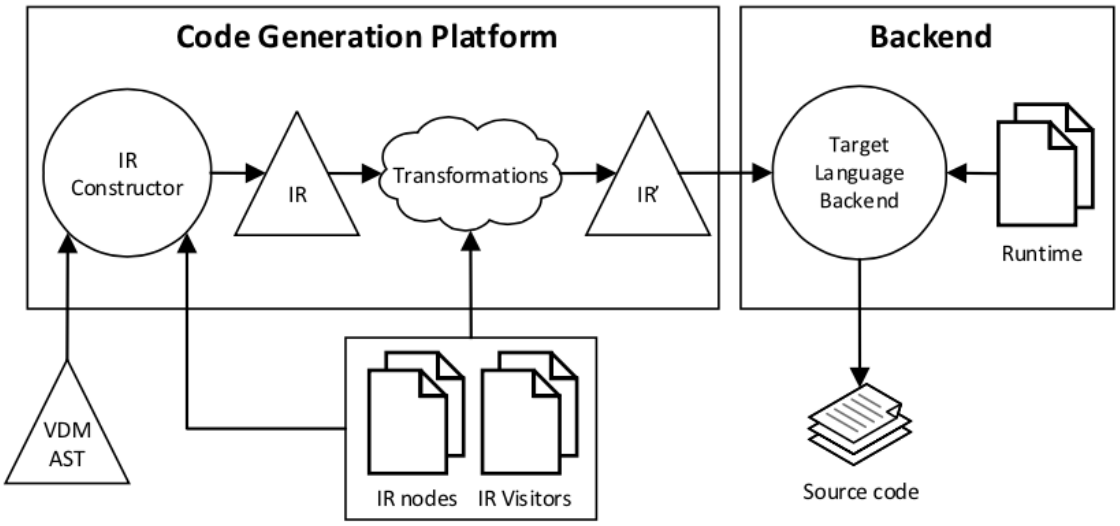

Figure 4: Code Generator Platform Architecture overview

4.1.2 Current Code Generation Capabilities

The CGP has been used to developed the current code generator capabilities supported in Overture. Currently, it is possible to generate Java code

from both a VDM-SL and VDM++ model in Overture. Because VDM-SL and VDM++ share common constructs it was possible the reuse some of the transformation for both VDM-SL and VDM++ code generation. Additionally, a prototype for generating code from a VDM++ model to C++ has been developed by using the CGP. For this C++ CG it was possible to reuse transformation from the Java CG, because they have share some similar language constructs, which was also exemplified above.

4.2 20-sim

The 20-sim application is a modelling and simulation program for multi-domain dynamic systems. It provides a series of tool boxes for building models, running simulations and analysing their behaviour. These models can be exported as C code for use in C and C++ programs or for execution on hardware. The code generation toolbox consists of two separate tools: 20-sim and 20-sim 4C.

4.2.1 20-sim Modelling & Simulation

20-sim has a hierarchical model structure. At the highest level, a system can be modelled graphically, similar to drawing an engineering scheme. At the lowest level, a system is represented by writing equations described in the language SIDOPS+, following the standard mathematical notation.

A 20-sim model without an interface (inputs, outputs) has no hierarchy and is thus automatically an equation model. Equations may be entered in random form. The correct order of execution is determined during processing of the model, when 20-sim automatically tries to rewrite equations into a causal form, i.e. a form where all output variables are written as a function of input variables. The resulting code is then used by 20-sim to perform simulation runs and to generate *model C code*, i.e. template-based code for ANSI C, C++ and Matlab.

4.2.2 20-sim 4C

20-sim 4C is a rapid prototyping tool for real-time control systems, such as PCs and embedded Linux processor boards. 20-sim 4C extends the model C code generated from 20-sim and runs it on a target platform. The name

4C stands for the four tasks required in order to get code running on the target:

1. *Configure* the target by selecting the appropriate template and specifying the network address where it is accessible.
2. *Connect* the inputs and outputs of the model to the appropriate target devices, such as onboard sensors or external actuators.
3. *Compile* the target-specific C code to create executable code.
4. *Command* 20-sim 4C to upload and run the executable code on the target with the option to modify, monitor and log model parameters.

Supported targets include any Linux-based platform that runs a real-time framework (RTAI or Xenomai),[1] and the industrial controller system Bachmann M1.[2] It is possible to extend support for additional targets with the use of templates.

4.3 OpenModelica

OpenModelica [Fri04] is an open-source Modelica-based modeling and simulation environment. Modelica [FE98] is an object-oriented, equation based language to conveniently model and simulate complex multidomain physical systems. The OpenModelica environment supports graphical composition of Modelica models. Models are simulated via translation to FMU, C or C++ code.

Compilation of Modelica models in OpenModelica happens in several phases [Sjö15], see also Figure 5:

- frontend - removes object orientation structure and build the hybrid Differential Algebraic Equations (DAE) system to be solved
- backend - the hybrid DAE system is index reduced, transformed to causal form (sorted), and optimized
- codegen - the optimized system of equation is transformed to FMU, C or C++ code using a template language

[1] https://www.rtai.org, https://xenomai.org
[2] http://www.bachmann.info/en/products/controller-system/

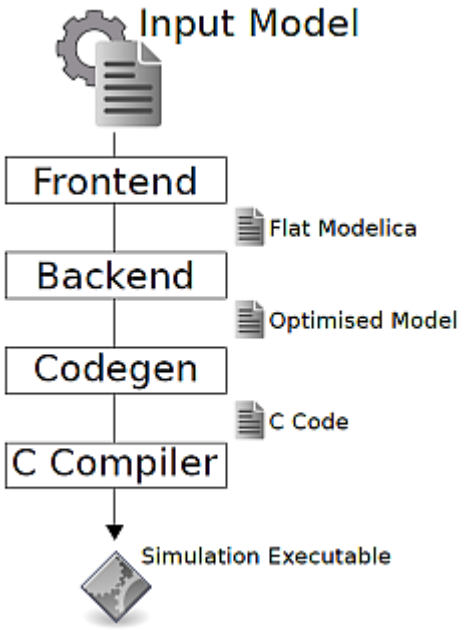

Figure 5: OpenModelica compilation phases

5 Design Principles of the Overture Code Generator

5.1 Introduction

This section presents the main principles of the Overture CG for VDM-RT models that will be developed as part of the INTO-CPS tool chain. The modelling principles of VDM-RT and the current possibilities of code generation support in Overture have been introduced in sections 3.1 and 4.1, respectively.

Existing formal development methods (VDM [Jon99], Z [WD96], Event-B [Abr10]) advocate a gradual transformation of model toward implementation, where implementation-specific changes are made to the model in successive, semantics-preserving steps. In the case of automatic code generation, this human-guided process of refinement can be collapsed to a single, automated transformation step, because the starting model is sufficiently concrete with respect to the implementation.

The remaining part of this section is structured as follows, in order to describe

the main design principles of this Overture CG. Subsection 5.2 presents the main goal of this CG. Afterwards, subsection 5.3 describes some additional challenges related to generating code from VDM-RT models in a INTO-CPS context. Then subsection 5.4 argues that generating code for the computation and communication elements can be addressed separately. Finally, subsections 5.5 and 5.6 present the design of a CG for the computation element and the communication element, respectively.

5.2 Main Goal

As described in section 2.1.1 the Overture CG will fulfil the requirements **0038** and **0086**. Furthermore, a VDM-RT model will need to be generated to target C++-code and C-code in order to fulfil the industrial needs, and this is related to the requirement **0038**. Afterwards, this CG will fulfil requirement **0086** by wrapping the generated code to make it compliant with the FMI specification.

One aspect needs to be clarified with respect to the main goal of the Overture CG; only VDM-RT supports modelling of a distributed system architecture compared to the two other baseline modelling languages/tools. Each CPU in a VDM-RT model is viewed as a separate platform with its own executable (e.g. compiled code) running. Hence a model deployed to a single CPU is generated to specific code in a similar manner as the two other INTO-CPS CGs. However, if a system is modelled using the distributed aspects, then distributed technologies have to be used in order to support network communication in the implementation. This separation will further be addressed below.

5.3 Mapping input and output

It is important to stated that a VDM-RT model does not have a notion of external input and output ports. Hence a VDM-RT model can be viewed as an isolated software model. However, in order to be useful in the development of CPSs, it has to support/introduce the notion of external ports, in order to connect it to other models, e.g. either DE or CT. As a consequence this plays a key role when generating code from a VDM-RT model, since it is more a Crescendo VDM-RT model, and not a stand-alone VDM-RT model. Hence it has to be investigated how this mapping of ports is achieved in the generated code.

The notion of inputs and outputs is also relevant when the code generated from a VDM-RT model is packed as an FMU and is to be connected to other FMUs in order to support SiL simulation. This challenge is also related to the toolwrapper for FMI for Overture described in the delivery D4.1b [PBLG15]. For the toolwrapper the input and output of the FMI standard have to be introduced in a VDM-RT model, before they can be supported by the toolwrapper. Therefore, a clear definition of inputs and outputs is required when a VDM-RT model is exported as an FMU.

5.4 Computation and Communication elements

The two main features of VDM-RT used to model an overall system architecture are the CPU and the BUS, which are used for computation and network communication, respectively. From a code generation perspective, these two components can be separated during the code generation process.

Basically a model deployed to a CPU corresponds to a VDM++ model, while the BUS models network communication between CPUs. The approach of dividing the code generation process was addressed in [Has14, HLTJ15]. In [Has14] the existing Java CG was used in order to generate the model deployed to each CPU, while the distribution technology Java Remote Method Invocation (RMI) [Sun00] was added in order to support distributed aspects of VDM-RT.

Following the discussion above, code generation can be separated in two main phases:

- code generation of the model deployed to a CPU, e.g. the computation element.
- code generation of the communication between CPUs, e.g. the communication element.

The main principles of code generating these parts will be discussed in the following two subsections, respectively.

5.5 CPU code generation - Computational Element

In VDM-RT, a computing element is modelled as a CPU. The model deployed to a CPU corresponds to a specific platform on which the software model is intended to execute. Hence this part of the code generation corresponds to

generating code for a subset of VDM-RT, in which only one CPU is used to model the system. Basically this subset can be viewed as a VDM++ model with support for modelling time and asynchronous operation calls.

The code generation process of the computation element can be further divided into two separate concerns which, to some extent, are connected, namely:

- **Functional**: Preserving the semantics of the functionality in a VDM-RT model in the target language, e.g. this is without ensuring the simulation time aspects of VDM-RT.
- **Non-functional**: Requirements when transforming the semantics of the model to the actual code, which include real-time requirements, safety critical behaviour, error handling/traceability and traceability between a model and the generated code.

The subsections below address these two concerns. However, the main goal of this Overture CG will be to support the first concern. The second concern is generally harder to provide guarantees for in a CG context, so a best effort approach is needed.

5.5.1 Functional aspects

The main goal of this part is to transform the VDM-RT model to an target language while preserving a refinement relationship between the model and the generated implementation, as shown in Figure 6. This code generator will only focus on generating toward a Intermediate Language Representation (ILR), which is the target language, as shown in Figure 6, while existing compilers will be used in order to target specific hardware platforms. Especially, it will be investigated how 20-sim 4C can be used in order to generated platform specific code from the platform independent code support by the Overture CG.

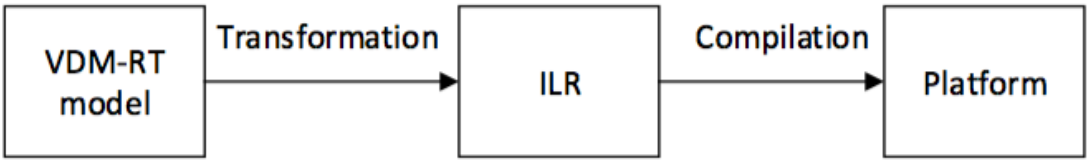

Figure 6: Main overview of the Overture CG

The target ILR can be used by different programming languages in accordance with the industrial needs. An overview of the possible ILR is shown in Figure 7 together with VDM AST, IR and the implementation. The possible ILR, both with respect to the industrial needs and in general, are discussed below.

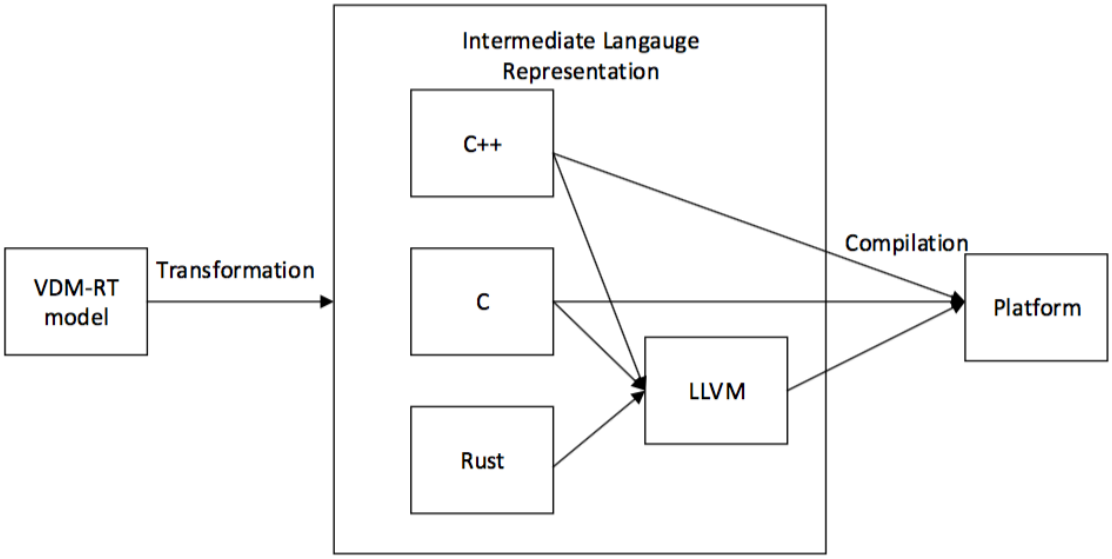

Figure 7: Possibilities for the Overture CG

Figure 7 provides an overview of possible ILR which possible may be researched: C++-code [ES90], C-code, Rust[3] and LLVM[4]. In order to describe the main principles, no further explanation of the specific constructs of the ILR will be given, since they are not relevant for the design principles. All these possible ILR are target languages for which possible translation have been investigated.

As mentioned in section 4.1 a prototype has been made for generating VDM++ models to C++-code. This prototype, however, is just a proof of concept CG in order to illustrate the reusability of the Overture CGP. It is important to stress that C++ and C, both which are required by the INTO-CPS case studies, share many common constructs. Since C++ was based on C and extended C with classes (e.g. "C with classes"), they share many similar language constructs. C++ is more similar to a VDM-RT model, since both introduce the notion of classes. However, this indicates that transformations

[3]Homepage: `www.rust-lang.org`
[4]Homepage: `www.llvm.org`

for specific constructs which C and C++ share, in the Overture CG made for targeting C++ can be reused when targeting the C language. As a consequence it will not be required to develop two separate code generators for targeting C and C++. It will require to apply different transformations for VDM-RT construct that are different for C and C++.

It has also been investigated how to transform a subset of VDM++ to the Rust programming language in collaboration with the European Space Agency (ESA), because they are also interested in transforming a VDM model to a specific embedded system. The following is claimed about Rust, on the official homepage, which is a motivation for using Rust: "Rust is a systems programming language that runs blazingly fast, prevents nearly all segfaults, and guarantees thread safety.". This claim is a consequence of the restrictions enforced by the semantics of Rust and the compiler[5]. What makes Rust especially relevant for the INTO-CPS project, is that a compiler exists for Rust that can transform it to a LLVM IR. Afterwards, this LLVM IR can target a specific platform by using an existing target specific LLVM compiler.

As Figure 7 indicates LLVM is more low-level then the other ILR possibilities. A point from the figure is that all the possible ILR can be generated to LLVM. LLVM is a IR, before the code is compiled to machine code for a specific platform. Additionally, there exists a compiler from C++, C and Rust to LLVM. Hence constructs from both C and C++ can be supported by LLVM constructs. So platform specific code is obtained by using existing compilers for LLVM. However, both C and C++ can be compiled to target specific code directly, without going through the LLVM IR. Currently it has been researched how to use a LLVM API[6] in order to make the transformation between VDM-RT and LLVM.

5.5.2 Non-Functional aspects

Using the time semantics from VDM-RT allows for modelling real time systems. Different literature has addressed the aspects of the real time, such as in [Tit06]. In this delivery real time is defined as a systems capability to meet deadlines, which can be analysed using a VDM-RT model. As a consequence limitations in the VDM-RT model itself may be introduced, because it may be possible in a model, but not in the real implementation in order to ensure

[5]Reference Manual: `doc.rust-lang.org/stable/book/`

[6]Homepage: `pauladamsmith.com/blog/2015/01/how-to-get-started-with-llvm-c-api.html`

the real time performance. Ensuring the same functional semantics between a model and the implementation may necessitate additional requirements as a consequence of differences between a VDM-RT model and the ILR.

One nontrivial aspect is the mapping of time notions introduced in a VDM-RT model to the features of the final implementation language. In VDM-RT it is possible to model time estimates using the `duration` and `cycles` language constructs. However, the ILR does do have a notion of time. Moreover, the timing characteristics of an implementation are a consequence of the software design, as well as of the hardware platform. However, in [LJL15] it has been already investigated how to map time between a VDM-RT model and implementation on a specific target platform.

As a consequence as part of the VDM-RT CG design, it may be investigated which VDM-RT constructs can be supported by this code generator when targeting ILR with real time support. Additionally it can be investigated which restrictions have to be set on the modeller when targeting an embedded platform. Such restrictions have been addressed by [Tit06], as an example, which identify constructs that are not suitable when targeting reliable embedded platforms.

So as part of the non-functional aspects for the implementation, the Overture CG may include the following consideration for research for the code generation:

- No dynamic memory allocation
- Statical memory (e.g. code size)
- power consumption
- performance (speed)
- safety critical behaviour

5.6 BUS code generation

This section focus on the distributed aspects of VDM-RT, and how a code generator can be used in order to support this in the implementation. Currently two INTO-CPS case studies plan to use code generation of the distributed aspects as part of their implementation.

VDM-RT supports modelling of distributed systems, but VDM-RT does not support any specific network technology, it is just an abstract network. How-

ever, the communication paradigm for the distributed aspects in VDM-RT is RMI; an object can invoke methods of an object deployed on another CPU as if it runs in the same address space. Code generating the distributed aspects raises both some general challenges related to distributed systems, and some specific to the code generation of a VDM-RT model to a distributed implementation. These challenges have been addressed by [Has14], which include the following:

- Transforming the communication paradigm of the VDM-RT model to the communication paradigm of the distributed technology while preserving the semantics from the VDM-RT model in the implementation.
- Initialising the whole distributed system in the implementation with respect to the VDM-RT model.
- Enable objects located on different CPUs to communicate with respect to the VDM-RT model.

A prototype for generating the distribution aspects of VDM-RT has been developed as a prototype in [Has14], by using Java RMI in order to support the distributed aspects in the implementation. Since the case studies can model distribution in VDM-RT, different distribution technologies may be investigated, which the code generator can use. For the UTRC case study it may be investigated how automatic support can be provided for the BACnet[7] distributed technology, because they used BACnet for network communication. For ClearSy it still is not decided which distribution technology will to used in the implementation. However, different aspects, related to the challenges described above, will need to be taken into account during the research.

[7]Homepage: `www.bacnet.org`

6 Design Principles of the 20-sim Code Generator

6.1 Introduction

This chapter explains the design principles behind the code generation toolbox in 20-sim and how 20-sim 4C extends the generated model code with target code. The result is a standalone program that can run on (industrial) computers, PLC systems and various embedded boards like ARM, AVR or PIC microcontroller-based boards.

20-sim is a modelling and simulation package that can generate C-code from a graphical or equation model. 20-sim 4C is a rapid prototyping application that takes the generated model code as an input, combines it with target-specific code and prepares it for running on a real-time target.

The main design principle behind the separation between 20-sim and 20-sim 4C is that a model should be independent of the actual target it should run on. A model should contain only the necessary information of the target relevant for the simulation and no details for code generation. Therefore, a typical 20-sim model contains no information about the target it will run on. The model can contain behavioural details about the target I/O like the accuracy of an analog-to-digital convertor, but detailed knowledge about the actual chip used and how to read its value is not necessary for the simulation and therefore not part of the model. As a consequence, 20-sim is not able on its own to produce standalone C-code that can access specific hardware. It can only generate standalone C-code that includes the model behaviour.

Section 6.2 describes the 20-sim code generation process in more detail. Section 6.3 describes how 20-sim 4C takes the input from 20-sim and how it is extended to run on the target platform. Section 6.4 summarizes the implemented and planned extensions to 20-sim and 20-sim 4C for INTO-CPS.

6.2 20-sim

The 20-sim ANSI-C/C++ code is generated based on all SIDOPS+ equations inside the model. Figure 8 shows the flowchart of the 20-sim code generation process. The processing phase in 20-sim takes the graphical -or-

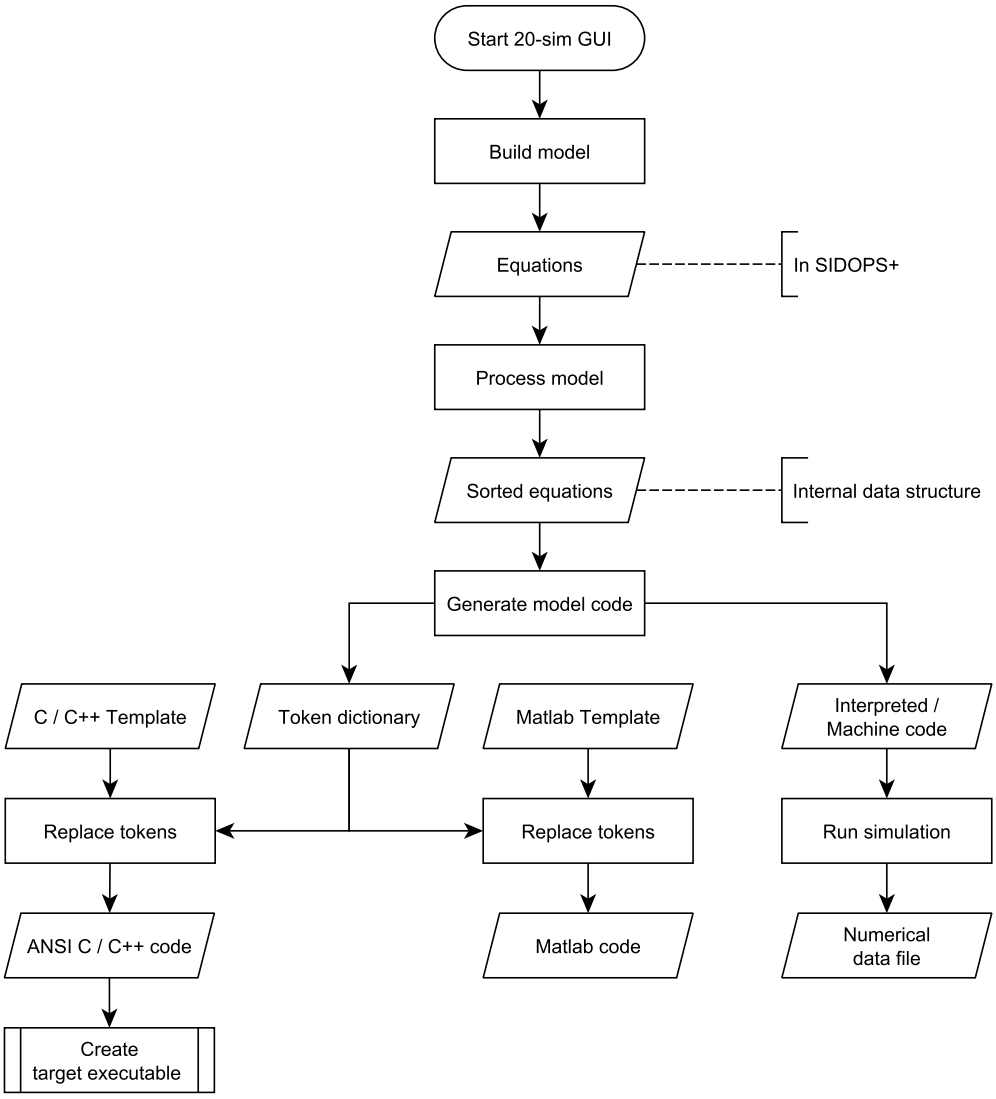

Figure 8: Flowchart of code generation from 20-sim.

equation model, flattens the model and translates it into an Hybrid Differential Algebraic Equations (DAE) system. This DAE system is transformed into a causal form (set of sorted equations). These sorted equations are then further optimized for both simulation and code generation purposes.

20-sim uses code generation templates to generate code for different purposes. Examples of these code generation templates are: ANSI-C code, C++ class, Matlab/Simulink export and FMU export. These templates contain special

tokens that 20-sim will replace by the corresponding parts of the model. 20-sim translates the optimized sorted equations into several blocks of ANSI-C code (e.g. initialization code, static equations, dynamic equations). These blocks are all stored in a token dictionary. Based on the token dictionary and the selected code generation template, the actual code is produced by means of a token replacement step.

As an example, the 20-sim equations block shown in Listing 3 is translated by 20-sim into the ANSI-C code shown in Listing 4. Note that 20-sim generates the original SIDOPS+ code line as a comment above each generated code line.

Listing 3: 20-sm SIDOPS+ equation model

```
parameters
  real A[2,2] = [1,2;3,4];
  real B = 2;
variables
  real C[2,2];
  boolean d;

equations
  C = A * (B + 1) + time;

  d = if (C[1,1] > 4) then
    true
  else
    false
  end;
```

Listing 4: Corresponding ANSI-C code snippets

```
XXDouble xx_P[5];                /* parameters A, B */
XXMatrix xx_M[3];                /* matrices */
XXDouble xx_V[5];                /* variables C, d */

void XXModelInitialize (void)
{
  /* set the parameters */
  xx_P[0] = 1.0;                 /* A */
  xx_P[1] = 2.0;
  xx_P[2] = 3.0;
  xx_P[3] = 4.0;
  xx_P[4] = 2.0;                 /* B */

  /* set the matrices */
  xx_M[0].mat = &xx_P[0];                /* A */
  xx_M[0].rows = 2;
  xx_M[0].columns = 2;
```

```
  xx_M[1].mat = &xx_V[0];                    /* C */
  xx_M[1].rows = 2;
  xx_M[1].columns = 2;
  xx_M[2].mat = &xx_U[0];                    /* xx_U1 */
  xx_M[2].rows = 2;
  xx_M[2].columns = 2;
}

void XXCalculateDynamic (void)
{
  /* xx_U1 = (A * (B + 1)) */
  XXMatrixScalarMul (&xx_M[2], &xx_M[0], (xx_P[4] + 1.0));
  /* C = xx_U1 + time; */
  XXMatrixScalarAdd (&xx_M[1], &xx_M[2], xx_time);

  /* d = if (C[1,1] > 4)... ; */
  xx_V[4] = (xx_M[1].mat[0] > 4.0) ?
        /* 1.0 */
        XXTRUE
  :
        /* 0.0 */
        XXFALSE
  ;
}
```

The presented example uses matrices, which are not directly supported in ANSI-C. Therefore, the generated code includes some helper functions like *XXMatrixScalarMul()* and *XXMatrixScalarAdd()*. Note that the implementation of all helper functions is included in the generated code. The next section explains how the 20-sim 4C extends the generated 20-sim code to run the model on a target.

6.3 20-sim 4C

The input for 20-sim 4C is external C-code (e.g. generated from 20-sim) together with an XML model description with the available parameters and variables in the code. The full process for creating target executables from the input code and running it on the target is shown in Figure 9. The details of each step are described below.

In 20-sim 4C, the user can select a specific target from a list of target templates (similar to code generation templates in 20-sim). A 20-sim 4C target template contains all information necessary to extend the input code into an executable ready to run on the target. Typical information found in a target

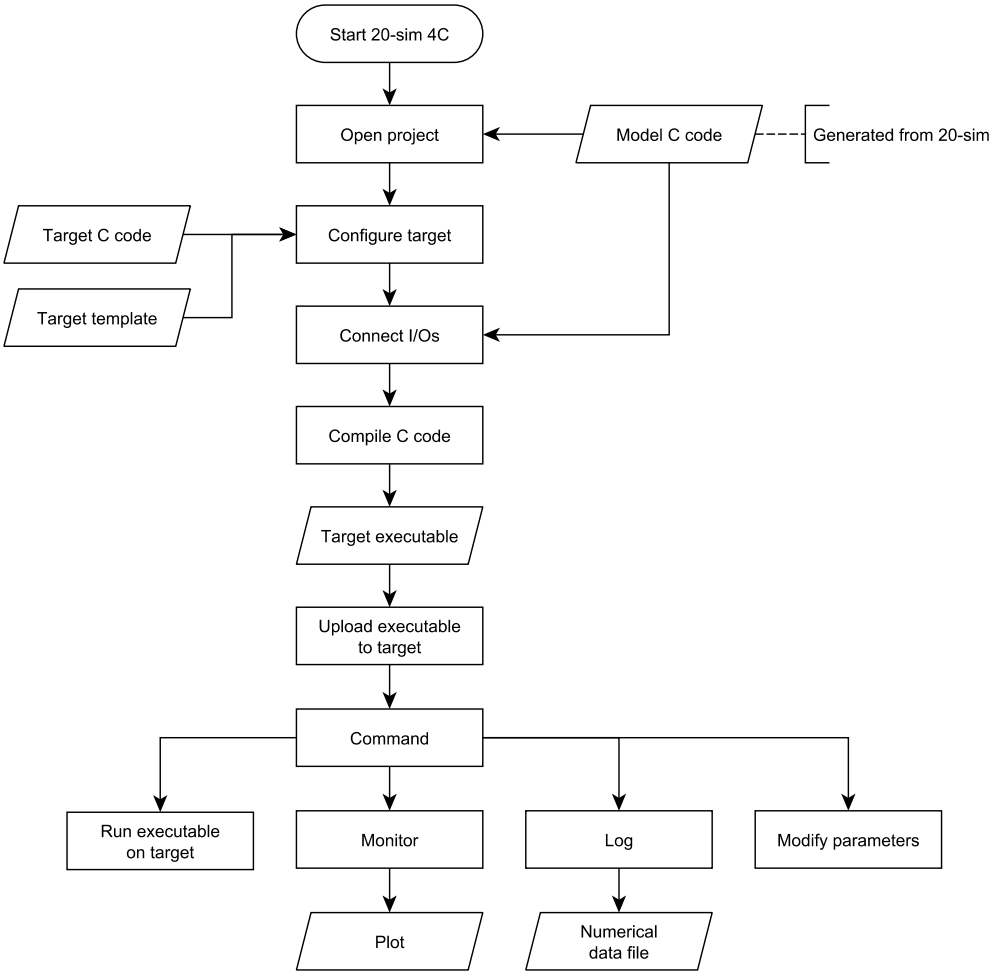

Figure 9: Flowchart of code generation from 20-sim 4C.

template is:

- Description of the target
- Options for automated target discovery
- Connection information
- List of available I/O connections
- Required compile commands
- Supported rapid prototyping features like: monitoring, logging and run-time parameter modifications

- Driver code needed for the target
- Additional target-specific source files
- Files to upload or flash

After the target selection, the user can connect model inputs and outputs to target I/O (e.g. AD and DA converters) or fieldbus I/O. The input code is then extended with the target-specific code, like real-time task initialization, the required I/O function calls, the needed driver code and 20-sim 4C hooks to allow real-time monitoring and logging of variables and on-the-fly parameter changes. Next step is the compilation into a target executable using the compiler settings from the target template. This typically involves calling the right cross compiler. After the compilation, the binary is ready to upload to the target, for example using an ethernet connection. After a succesful upload, the real-time model task will be started on the target. 20-sim 4C will then use the communication link with the target for monitoring, logging and parameter modification.

6.4 INTO-CPS extensions

The previous sections present the design principles of the 20-sim and 20-sim 4C code generation facilities. Both tools need extensions to fulfil the requirements from Section 2 and in particular the 20-sim and 20-sim 4C requirements from Section 2.1.2. The lists below summarize the planned and in progress changes to both tools to fulfil the INTO-CPS needs.

20-sim

- **FMI import:**
  1. Reading and preserving an FMU modelDescription.xml interface definition coming from the Modelio tool. [in progress]
  2. Run an FMU inside 20-sim. [planned]
- **FMI export:**
  1. FMU v1.0 and v2.0 co-simulation (standalone) export functionality. [in progress]
  2. FMU v1.0 and v2.0 co-simulation (toolwrapper) functionality. [in progress]

3. Possibility to embed a 20-sim 3D animation inside a FMU for displaying purposes. [in progress]

- **Integration methods:** Variable step-size method support. [planned]
- **Code generation:**
  1. Automated code generation (scripting). [done]
  2. Automated testing. [in progress]

20-sim 4C

- **FMI import:** FMU v2.0 (with included sources) as input source for 20-sim 4C [in progress].
- **Targets:** Work on supporting the targets selected by the industrial partners: PIC32 and B&R PLCs. See also section 2.2 [in progress].
- **Code generation:** Automated testing. [in progress]
- **HIL testing:** FMU interface to a real-time 20-sim 4C task running on a real-time target to allow Hardware-In-the-Loop testing. [planned]

7 Design Principles of the OpenModelica Code Generator

7.1 Introduction

The design principles of the code generation in the OpenModelica simulator are explained in this section. The OpenModelica simulator transforms Modelica code into different lower level languages that can be compiled into executable code. Currently OpenModelica can generate code in these languages: C, C++, JavaScript. Additionaly, the OpenModelica simulator can generate FMI 1.0 or 2.0 for Model Exchange and Co-Simulation.

7.2 OpenModelica Code Generation

The transfomation from Modelica language into executable code hapens in several phases (see also Figure 10):

- Flattening - removing of object orientation from the Modelica language and creation of an hybrid DAE (Differential Algebraic Equations)
- Basic Optimization - optimization of the hybrid DAE system, index reduction, matching, equation sorting, causalization
- Advanced Optimization - more optimization of the system of equations, alias elimination, tearing, common subexpression elimination, etc. (DAELow)
- Independent Simulation Code - the final system of equations is transformed into an independent simulation code structure (SimCode)
- Code Generation - the Independent Simulation Code stucture is given to several templates which can generate code in different languages, currently C, C++, JavaScript, and FMU
- Simulation - the code is compiled into a standalone executable from the generated code and executed

The OpenModelica template language called Susan [FPSP09] is used for writing the templates for code generation.

7.3 INTO-CPS extensions

In the INTO-CPS project OpenModelica will be extended to support the requirements in Section 2.1.3 .

Below we present the already available functionality in OpenModelica and the extensions planned:

- **FMI import:**
  1. Reading and preserving an FMU modelDescription.xml interface definition coming from the Modelio tool; [in progress]
  2. Running an FMU inside OpenModelica is already available for Model Exchange, extension for Co-Simulation is needed; [in progress]
- **FMI export:**
  1. FMU v1.0 and v2.0 for model-exchange and co-simulation (standalone) export functionality; [in progress, first draft available]
  2. FMU v1.0 and v2.0 for model-exchange and co-simulation (tool-wrapper) functionality; [in progress]

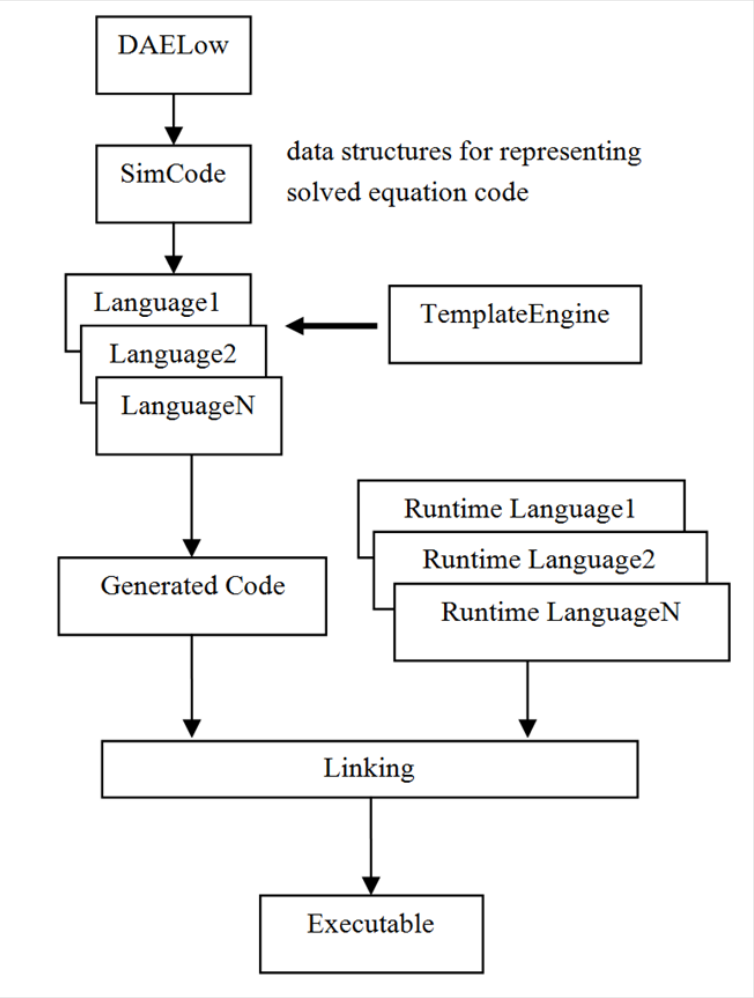

Figure 10: OpenModelica code generation using templates

3. source code FMUs which contain all the necessary sources that could be compiled on any target, primarily 20-sim 4C [in progress]

- **Integration methods:** all integration methods available in OpenModelica can be used inside the FMU for co-simulation
- **Code generation:**
  1. Automated code generation (via .mos scripting) is already available
  2. Automated testing (via .mos scripting) is already available

8 Related Work

This section presents related work with respect to the above described goals of the INTO-CPS CGs. For these CGs the main approach is twofold. First they have to transform a model into a target language. Afterwards, this target language has to be made executable on a platform. Both these aspects have been addressed by the researchers, and will be related below.

In general the focus is on related work which considers using automatic code generation as part of a model-driven development process, similar to what is suggested for the INTO-CPS project. So especially with respect to this related work will be highlighted. Additionally the focus is on going from a software model to generating runnable platform dependent code for embedded systems.

The remainder of this section is structured the following way. First section 8.1 discusses code generation as parts of MBD in general. Afterwards, section 8.2 describes the transformation from a model to Platform Independent Code (PIC), which is code that is not targeting any specific platform. This relates to all the INTO-CPS CGs, in that the model is transformed to semantically equivalent code with respect to the industrial needs. Then section 8.3 focuses on related work for generating Platform Dependent Code (PDC), which is code that can be executed on a specific hardware platform. Hence as previously stated it is required for the INTO-CPS CGs that the generated code is correct-by-construction for a specific platform. In the literature in general, as also stated above, it is distinguish between PIC and PDC. Afterwards, sections 8.4 and 8.5 present software for CPSs in particular and generating code from formal methods. Finally, section 8.6 presents related project, which have addressed code generation as parts of their research.

8.1 Code Generation in MBD

Using code generating as part of MBD can support the overall development as discussed in section 1. The part of using code generation has been addressed in general by research in MBD in order to ensure consistency between a model and the implementation (e.g. the code). This approach has also been taken for some of the related project presented in section 8.6. As an additional example in [AVCS+07] the focus is on using code generation as a means of Model-Driven Engineering (MDE) in order to ensure the original requirements in the implementation. As part of their approach they use

the Model-Driven Architecture (MDA) approach from Object Management Group (OMG). A part of MDA is to go from having a Platform Independent Model (PIM) to a Platform Specific Model (PSM). In this paper they provide an example with a model of a state-machine, which is automatically transformed to code. The authors show how code generation is used to ensure consistency between the model and the implementation in the Ada programming language [Ada83]. Code generation within MBD has also been addressed by commercial companies, such as MathWorks[8].

8.2 Model to PIC transformation

The platform independent code generation has been addressed by different researchers This is a very broad research area, since this corresponds to a transformation from a source language to a target language. Such PIC transformations are addressed by Times [AFM+03] and Simulink coder [Mat10]. As a consequence the focus is on specific transformations for the INTO-CPS modelling notations.

For VDM transformations have been made for Java [JLC15] for the Overture platform. In addition the VDMTools [CSK07] platform, which supports VDM-SL and VDM++ modelling too, is able to generate Java and C++ code, but it is not documented. These transformation, however, are made without targeting a specific platform. Even though for Java it is possible to run on different platform, because the Java Virtual Machine makes Java code platform independent, while the C++ code is not targeting any specific hardware platform.

8.3 Model to PDC transformation

An important part of the INTO-CPS CGs is to support correct-by-construction code targeting specific platforms in accordance with the industrial needs. Hence it should not be required to adapt the code manually for a given platform. Different papers have addressed this type of automatic generation of correct-by-code from different modelling languages. In general in the related work a key point is that PIC can be generated to different PDC implementations based on the actual hardware platform. In [CS12] they address four different deployment possibilities, related either using single-core

[8]Automatic Code Generation Within Model-Based Design: `http://se.mathworks.com/videos/automatic-code-generation-within-model-based-design-92624.html`.

or multi-core platforms. The main focus of related work is on CGs that are targeting single-core platforms, because the INTO-CPS case studies are targeting single-core platforms. However, it should still be noted that research for generating code for multi-core platforms has been addressed by [PBPR09, CVC10, CKL+11].

8.3.1 UML and State-machines

In [CCS12, FRGT10, MWP+10, VdLG+09, HTB08] they investigate generating code from Unified Modelling Language (UML) [UML99, FS03] models and state-machines. Their goal is, similar to INTO-CPS, to use a higher level of abstraction in order to focus on solving the high-level goal, while a CG is used in order to ensure the transformation between the model and code. Especially, in [CCS12], as part of the CHESS project[9], they research of generating correct-by-construction code for complex embedded systems, which is compliant with the goal of the INTO-CPS CGs described in this deliverable.

8.3.2 AADL

Another part of research with respect the generating code from models, has been carried out for using Architecture Analysis & Design Language (AADL) [AAD04] models for high-level modelling. An AADL model is similar to a SysML model, which is used for high-level modelling in INTO-CPS [Con14]. In [KPSL13] they look at how to generate from PIC to PDC by using AADL model to specify the properties of the platform. In their approach they have the main functionality given by the PIC. Afterwards, different versions of the PDC implementation can be generated from the same PIC. This is a consequence of that different hardware platforms give different implementation possibilities. Also research is carried out in [BDT08], in which the authors described generating code for real-time embedded systems from AADL models by using MDA tools. Additionally, in [LZPH09] the authors present a toolsuit *OCARINA* that supports code generation from AADL models.

[9]Project website at `http://www.chess-project.org`.

8.3.3 From PIC to PDC transformation

In [BDN12] they identify a gap between the MBD and MDA approach, which they attend to close with their attempt. The idea is to have a platform independent model (e.g. a functional model). Afterwards, a hardware platform is selected and its properties described. Finally, a mapping of the functional components to the specific hardware components is described. This information is then used as input for their code generator to generate PDC. Actually they also support generating support for middleware, which is especially interesting for the distributed aspects of VDM-RT. Similar work has been carried out in [SV07, Aut10, MMS07], in which they also address matching between functional and execution architectures.

8.4 Software in CPSs

In [ELM+12] they specifically address challenges of developing software for CPSs. In this paper the authors stress that a key difference between software and software for CPSs, is that time is a central aspect of the system behaviour. This paper presents a coordination language for software components, called PTIDES. Their approach is to use PTIDES in order to automatically generate the glue code for applications in Java or C. In order for PTIDES to be able to ensure time between the model and code, each software component has to provide it worst case execution time. With this information PTIDES can map the time from the model to the implementation in code.

8.5 Formal methods - B method

The B method [Abr91, Abr96] is a formal method which is similar to VDM. This formal method is also used for support software development from specifications, and can be used for the development of safety-critical systems. In [BDLJ14] initial work is described how a subset of the B method can be mapped to LLVM constructs. Afterwards the authors can use existing compilers for LLVM in order to target a specific target platform.

8.6 Related Projects

The survey of existing European research projects has identified five projects which address, in some capacity, the notion of automatically translating a formal model of a software system to executable code. Here we briefly review the approaches found therein.

In the DEPLOY project[10] a method is proposed for automating part of the process of refining Event-B specifications to Java implementations via an intermediate B-based notation, Object-oriented Concurrent B (OC-B). The purpose of OC-B is to incorporate enough features of object-oriented, concurrent programming languages to make automatic translation to Java or any other object-oriented language fully automatic. The refinement from a top-level Event-B system specification to OC-B is kept as a manual process.

The CERTAINTY project [11] develops a method for code generation for many-core architectures which also uses an intermediate system model. The core of the approach is a layered semantic model for interacting components, whose interaction is based on dynamic priorities. This is called the Behaviour, Interaction, Priority (BIP) model. One layer of the semantic model captures component behaviour as Petri nets augmented with C functions and data. The other layer captures both the interaction between components and the priorities of these interactions. The effect is a description of the scheduling policy governing the execution of the components. Code generation is focused at the transformation of a BIP model into executable C++ code as either single-threaded, multi-threaded and real-time implementations. Naturally, the multi-threaded and real-time targets depend on corresponding support from the underlying platform. Support for the transformation of existing implementations into BIP models also exists for implementations written in languages for which operational semantics exist. Currently the project lists transformation capabilities from the following languages in to BIP models: Lustre, discrete time Simulink, AADL, GeNoM C++, unrestricted C, TinyOS nesC, DOL system descriptions.

Code generation in the ADVANCE project[12] focuses on translating specifications written in Tasking Event-B to Java and Ada code, and to C code for both OpenMP and FMI. Tasking Event-B is an extension to Event-B which adds implementation-specific annotations to facilitate the code generation

[10] Project website at `www.deploy-project.eu`.
[11] Project website at `www.certainty-project.eu`.
[12] Project website at `www.advance-ict.eu`.

process.

In the CompCert project[13] they research formal verification for compilers for critical embedded software. Their main goal is to have a verified compiler for almost all ISO C90 / ANSI C language, and generating code for the PowerPC, ARM and x86 processors.

[13]Project website at `compcert.inria.fr`.

References

[AAD04] Architecture analysis & design language (aadl). Aerospace Standard AS5506, SAE Aerospace, November 2004.

[Abr91] J.R. Abrial. The B Method for Large Software Specification, Design and Coding (Abstract). In *VDM '91: Formal Software Development Methods*, pages 398–405. Springer-Verlag, October 1991. Volume 2.

[Abr96] J.-R. Abrial. *The B Book – Assigning Programs to Meanings*. Cambridge University Press, August 1996.

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[Ada83] Reference manual for the ada programming language. Technical report, United States Government (Department of Defence), American National Standards Institute, 1983.

[AFM+03] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times: a tool for schedulability analysis and code generation of real-time systems. In *In Proc. of FORMATS?03, number 2791 in LNCS*, pages 60–72. Springer-Verlag, 2003.

[Aut10] Autosar. Autosar specifications 4.0, 2010. http://www.autosar.org/.

[AVCS+07] Diego Alonso, Cristina Vicente-Chicote, Pedro Sánchez, Bárbara Álvarez, and Fernando Losilla. Automatic ada code generation using a model-driven engineering approach. In Nabil Abdennadher and Fabrice Kordon, editors, *Ada-Europe*, volume 4498 of *Lecture Notes in Computer Science*, pages 168–179. Springer, 2007.

[BB97] Breunese and J. F. Broenink. Modeling mechatronic systems using the sidops+ language. In *The Society for Computer Simulation International*, pages 301–306, 1997.

[BDLJ14] Richard Bonichon, David Déharbe, Thierry Lecomte, and Valério Medeiros Jr. LLVM-Based Code Generation for B. *Formal Methods: Foundations and Applications: 17th Brazilian Symposium, SBMF 2014*, pages 1–16, September 2014.

[BDN12] M. Bambagini and M. Di Natale. A code generation framework for distributed real-time embedded systems. In *Emerging Technologies Factory Automation (ETFA), 2012 IEEE 17th Conference on*, pages 1–10, Sept 2012.

[BDT08] M. Brun, J. Delatour, and Y. Trinquet. Code generation from aadl to a real-time operating system: An experimentation feedback on the use of model transformation. In *Engineering of Complex Computer Systems, 2008. ICECCS 2008. 13th IEEE International Conference on*, pages 257–262, March 2008.

[Blo14] Torsten Blochwitz. Functional mock-up interface for model exchange and co-simulation. https://www.fmi-standard.org/downloads, July 2014. Torsten Blochwitz Editor.

[Bro90] Jan F. Broenink. *Computer-aided physical-systems modeling and simulation: a bond-graph approach*. PhD thesis, Faculty of Electrical Engineering, University of Twente, Enschede, Netherlands, 1990.

[CCS12] Federico Ciccozzi, Antonio Cicchetti, and Mikael Sjödin. Round-trip support for extra-functional property management in model-driven engineering of embedded systems. *Information and Software Technology*, pages 1085–1100, June 2012.

[CKL+11] Minji Cha, Kyong Hoon Kim, Chung Jae Lee, Dojun Ha, and Byoung Soo Kim. Deriving high-performance real-time multi-core systems based on simulink applications. In *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*, pages 267–274, Dec 2011.

[Con13] Controllab Products B.V. http://www.20sim.com/, January 2013. 20-sim official website.

[Con14] INTO-CPS Consortium. Grant agreement for INtegrated TOol chain for model-based design of CPSs (INTO-CPS). Grant agreement number 644047, European Commission, December 2014.

[CS12] F. Ciccozzi and M. Sjödin. Enhancing the generation of correct-by-construction code from design models for complex embedded systems. In *Emerging Technologies Factory Automation (ETFA), 2012 IEEE 17th Conference on*, pages 1–4, Sept 2012.

[CSK07] CSK. VDMTools homepage. *http://www.vdmtools.jp/en/*, 2007.

[CVC10] R.L. Collins, B. Vellore, and L.P. Carloni. Recursion-driven parallel code generation for multi-core platforms. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 190–195, March 2010.

[ELM+12] J.C. Eidson, E.A. Lee, S. Matic, S.A. Seshia, and Jia Zou. Distributed real-time software for cyber physical systems. *Proceedings of the IEEE*, 100(1):45–59, 2012.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Publishing Company, 1990.

[FE98] Peter Fritzson and Vadim Engelson. Modelica - A Unified Object-Oriented Language for System Modelling and Simulation. In *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 67–90. Springer-Verlag, 1998.

[FL09] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, Second edition, 2009. ISBN 0-521-62348-0.

[FLM+05] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object–oriented Systems*. Springer, New York, 2005.

[FLS08] John Fitzgerald, Peter Gorm Larsen, and Shin Sahara. VDMTools: Advances in Support for Formal Modeling in VDM. *ACM Sigplan Notices*, 43(2):3–11, February 2008.

[FLTV07] J. S. Fitzgerald, P. G. Larsen, S. Tjell, and M. Verhoef. Validation Support for Real-Time Embedded Systems in VDM++. Technical Report CS-TR-1017, School of Computing Science, Newcastle University, April 2007. Revised version in Proc. 10th IEEE High Assurance Systems Engineering Symposium, November, 2007, Dallas, Texas, IEEE.

[FLV08] J. S. Fitzgerald, P. G. Larsen, and M. Verhoef. Vienna Development Method. *Wiley Encyclopedia of Computer Science and Engineering*, 2008. edited by Benjamin Wah, John Wiley & Sons, Inc.

[FPSP09] Peter Fritzson, Pavol Privitzer, Martin Sjlund, and Adrian Pop. Towards a text generation template language for Modelica. In

Francesco Casella, editor, *Proceedings of the 7th International Modelica Conference*, pages 193–207. Linkping University Electronic Press, September 2009.

[FRGT10] M. Fredj, A. Radermacher, S. Gerard, and F. Terrier. ec3m: Optimized model-based code generation for embedded distributed software systems. In *New Technologies of Distributed Systems (NOTERE), 2010 10th Annual International Conference on*, pages 279–284, May 2010.

[Fri04] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1.* Wiley-IEEE Press, January 2004.

[FS03] Martin Fowler and Kendall Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* Addison-Wesley, 2003.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Has14] Miran Hasanagić. Code Generation for Distributed Systems Modelled in VDM-RT. Master's thesis, Aarhus University, Department of Engineering, December 2014.

[HLB+15] Francois Hantry, Thierry Lecomte, Stelios Basagiannis, Christian König, and Jose Esparza. Case Studies 1, Public Version. Technical report, INTO-CPS Public Deliverable, D1.1a, December 2015.

[HLTJ15] Miran Hasanagic, Peter Gorm Larsen, and Peter W.V. Tran-Jørgensen. Generating Java RMI for the distributed aspects of VDM-RT models. In *Proceedings of the 13th Overture Workshop*, pages 75–89, National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-Ku, Tokyo, Japan, June 2015. Center for Global Research in Advanced Software Science and Engineering. GRACE-TR-2015-06.

[HTB08] Wolfgang Haberl, Michael Tautschnig, and Uwe Baumgarten. From COLA Models to Distributed Embedded Systems Code. *IAENG International Journal of Computer Science*, 35(3):427–437, September 2008.

[HV10] J. Hooman and M. Verhoef. Formal semantics of a VDM extension for distributed embedded systems. In D. Dams, U. Hannemann, and M. Steffen, editors, *Concurrency, Compositionality, and Correctness, Essays in Honor of Willem-Paul de Roever*, volume 5930 of *Lecture Notes in Computer Science*, pages 142–161. Springer-Verlag, 2010.

[JLC15] Peter W. V. Jørgensen, Morten Larsen, and Luis D. Couto. A Code Generation Platform for VDM. In Nick Battle and John Fitzgerald, editors, *Proceedings of the 12th Overture Workshop, Newcastle University, 21 June, 2014*. School of Computing Science, Newcastle University, UK, Technical Report CS-TR-1446, January 2015.

[JLL13] Peter W.V. Jørgensen, Kenneth Lausdahl, and Peter Gorm Larsen. An Architectural Evolution of the Overture Tool. In *The Overture 2013 workshop*, August 2013.

[Jon99] Cliff B. Jones. Scientific Decisions which Characterize VDM. In J.M. Wing, J.C.P. Woodcock, and J. Davies, editors, *FM'99 - Formal Methods*, pages 28–47. Springer-Verlag, 1999. Lecture Notes in Computer Science 1708.

[KPSL13] BaekGyu Kim, L.T.X. Phan, O. Sokolsky, and Insup Lee. Platform-dependent code generation for embedded real-time software. In *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2013 International Conference on*, pages 1–10, Sept 2013.

[LBF+10] Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. The Overture Initiative – Integrating Tools for VDM. *SIGSOFT Softw. Eng. Notes*, 35(1):1–6, January 2010.

[LFW09] Peter Gorm Larsen, John Fitzgerald, and Sune Wolff. Methods for the Development of Distributed Real-Time Embedded Systems using VDM. *Intl. Journal of Software and Informatics*, 3(2-3), October 2009.

[LH+96] P. G. Larsen, B. S. Hansen, et al. Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language, December 1996. International Standard ISO/IEC 13817-1.

[LJL15] Morten Larsen, Peter W.V. Jørgensen, and Peter Gorm Larsen. Improving Time Estimates in VDM-RT Models. pages 90–103, June 2015. GRACE-TR-2015-06.

[LL09] Peter Gorm Larsen and Kenneth Lausdahl. User manual for the overture combinatorial testing plug-in. Technical Report TR-2009-01, The Overture Initiative, www.overturetool.org, March 2009.

[LLW+15] Kenneth Lausdahl, Peter Gorm Larsen, Sune Wolf, Victor Bandur, Anders Terkelsen, Miran Hasanagić, Casper Thule Hansen, Ken Pierce, Oliver Kotte, Adrian Pop, Etienne Brosse, Jörg Brauer, and Oliver Möller. Design of the INTO-CPS Platform. Technical report, INTO-CPS Deliverable, D4.1d, December 2015.

[LPH+15] Peter Gorm Larsen, Ken Pierce, Francois Hantry, Joey W. Coleman, Sune Wolff, Kenneth Lausdahl, Marcel Groothuis, Adrian Pop, Miran Hasanagić, Jörg Brauer, Etienne Brosse, Carl Gamble, Simon Foster, and Jim Woodcock. Requirements Report year 1. Technical report, INTO-CPS Deliverable, D7.3, December 2015.

[LZPH09] Gilles Lasnier, Bechir Zalila, Laurent Pautet, and Jrome Hugues. Ocarina : An environment for aadl models analysis and automatic code generation for high integrity applications. In Fabrice Kordon and Yvon Kermarrec, editors, *Reliable Software Technologies Ada-Europe 2009*, volume 5570 of *Lecture Notes in Computer Science*, pages 237–250. Springer Berlin Heidelberg, 2009.

[Mat10] MathWorks. Simulink Coder: Generate C and C++ code from Simulink and stateflow models, 2010. http://www.mathworks.com/products/simulink-coder/).

[MMS07] J. Martinez, P. Merino, and A. Salmeron. Applying mde methodologies to design communication protocols for distributed systems. In *Complex, Intelligent and Software Intensive Systems, 2007. CISIS 2007. First International Conference on*, pages 185–190, April 2007.

[MWP+10] T.G. Moreira, M.A. Wehrmeister, C.E. Pereira, J. Petin, and E. Levrat. Automatic code generation for embedded systems: From uml specifications to vhdl code. In *Industrial Informat-*

ics (INDIN), 2010 8th IEEE International Conference on, pages 1085–1090, July 2010.

[OCT07] Overture-Core-Team. Overture Web site. http://www.overturetool.org, 2007.

[PBLG15] Adrian Pop, Victor Bandur, Kenneth Lausdahl, and Frank Groen. Integration of Simulators using FMI. Technical report, INTO-CPS Deliverable, D4.1b, December 2015.

[PBPR09] Jonathan Piat, Shuvra S. Bhattacharyya, Maxime Pelcat, and Mickaël Raulet. Multi-Core Code Generation From Interface Based Hierarchy. In *Conference on Design and Architectures for Signal and Image Processing (DASIP) 2009*, page online, Sophia Antipolis, France, September 2009.

[PL92] Nico Plat and Peter Gorm Larsen. An Overview of the ISO/VDM-SL Standard. *Sigplan Notices*, 27(8):76–82, August 1992.

[Sjö15] Martin Sjölund. *Tools and Methods for Analysis, Debugging, and Performance Improvement of Equation-Based Models*. Doctoral thesis No 1664, Linköping University, Department of Computer and Information Science, 2015.

[SN07] Rasmus Ask Sørensen and Jasper Moltke Nygaard. Evaluating Distributed Architectures using VDM++ Real-Time Modelling with a Proof of Concept Implementation. Master's thesis, Enginering College of Aarhus, December 2007.

[Sun00] Sun. *Java Remote Method Invocation Specification*. 2000.

[SV07] A. Sangiovanni-Vincentelli. Quo vadis, sld? reasoning about the trends and challenges of system level design. *Proceedings of the IEEE*, 95(3):467–506, March 2007.

[Tit06] Ben L. Titzer. Virgil: Objects on the Head of a Pin. *OOPSLA*, pages 191–207, October 2006.

[UML99] UML Revision Task Force. The Unified Modelling Language, version 1.3. Technical report, Object Management Group, June 1999.

[VdLG+09] J. Vidal, F. de Lamotte, G. Gogniat, P. Soulard, and J.-P. Diguet. A co-design approach for embedded system modeling

and code generation with uml and marte. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 226–231, April 2009.

[Ver05] Marcel Verhoef. On the Use of VDM++ for Specifying Real-Time Systems. *Proc. First Overture workshop*, November 2005.

[Ver09] Marcel Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. PhD thesis, Radboud University Nijmegen, 2009.

[VL07] Marcel Verhoef and Peter Gorm Larsen. Interpreting Distributed System Architectures Using VDM++ – A Case Study. In Brian Sauser and Gerrit Muller, editors, *5th Annual Conference on Systems Engineering Research*, March 2007. Available at http://www.stevens.edu/engineering/cser/.

[VLH06] Marcel Verhoef, Peter Gorm Larsen, and Jozef Hooman. Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods*, Lecture Notes in Computer Science 4085, pages 147–162. Springer-Verlag, 2006.

[WD96] Jim Woodcock and Jim Davies. *Using Z – Specification, Refinement, and Proof*. Prentice Hall International Series in Computer Science, 1996.

[Wol08] Sune Wolff. Universal Multiprotocol Home Automation Framework. Master's thesis, Aarhus University/Engineering College of Aarhus, December 2008.

A List of Acronyms

20-sim	Software package for modelling and simulation of dynamic systems
AADL	Architecture Analysis & Design Language
AD	Analog-to-Digital
AST	Abstract Syntax Tree
AU	Aarhus University
BIP	Behaviour, Interaction, Priority
CG	Code Generator
CGP	Code Generator Platform
CLE	ClearSy
CLP	Controllab Products B.V.
COE	Co-simulation Orchestration Engine
CPS	Cyber-Physical System
CPU	Central Processing Unit
CT	Continuous-Time
DA	Digital-to-Analog
DE	Discrete Event
DAE	Differential Algebraic Equations
DoA	Description of Action
DSE	Design Space Exploration
ESA	European Space Agency
FM	Formal Methods
FMI	Functional Mockup Interface
FMU	Functional Mockup Unit
HiL	Hardware-in-the-Loop
HVAC	Heating, Ventilation, and Air Conditioning
HW	Hardware
ILR	Intermediate Language Representation
IR	Intermediate Representation
ISO	International Organisation of Standards
MBD	Model Based Design
MDA	Model Driven Architecture
MiL	Model-in-the-Loop
OC-G	Object-oriented Concurrent B
ODE	Ordinary Differential Equations
OMG	Object Management Group
PDE	Partial Differential Equations
PDC	Platform Dependent Code
PLC	Programmable logic controller

PIC	Platform Independent Code
PIM	Platform Independent Model
PSM	Platform Specific Model
RMI	Remote Method Invocation
SIDOPS	Structure Interdisciplinary Description Of Physical Systems
SiL	Software-in-the Loop
SW	Software
SysML	Systems Modelling Language
TWT	TWT GmbH Science & Innovation
UML	Unified Modelling Language
UTRC	United Technology Research Center
VDM	Vienna Development Method
VICE	VDM++ In Constrained Environments
V&V	Validation & Verification
WP	Work Package