MODEL DRIVEN DEVELOPMENT FOR MEDICAL DEVICES IN AADL/BLESS AND SPARK ADA: PCA PUMP PROTOTYPE

by

Jakub Jedryszek

B.S., Wroclaw University of Technology, Poland, 2012

B.A., Wroclaw University of Economics, Poland, 2012

A THESIS

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2014

Approved by:

Major Professor
John Hatcliff

Copyright

Abstract

Ada programming language is targeted at embedded and real-time systems.

SPARK Ada is designed for the development of safety and security critical systems. It contains properties, which allows to prove corectness of program and its entities.

AADL (Architecture Analysis & Design Language) is modeling language for representing hardware and software. It is used for real-time, safety critical and embedded systems.

BLESS (Behavior Language for Embedded Systems with Software) is AADL annex sub-language defining behavior of components. The goal of BLESS is automatically-checked correctness proofs of AADL models of embedded electronic systems with software.

Nowadays, we have trend to generate code from models. The ultimate goal of research, which this thesis if part of, is to create AADL/BLESS to SPARK Ada traslator. Ultimatelly there will be standardized AADL/BLESS models, which will be generating code base for developers extensions (like skeleton code for some Web Framework).

This thesis propose mapping from AADL/BLESS to SPARK Ada. As an example of Medical Device, PCA Pump (Patient Controlled Analgesia) is used. The foundation for this work is System Requirements for "Integrated Clinical Environment Patient-Controlled Analgesia Infusion Pump System Requirements" (DRAFT 0.10.1) [Lar14] and AADL Models with BLESS annexes created by Brian Larson. Additionally, there was a contribution made in clarifying the requirements document and extending AADL models.

Table of Contents

List of Figures

List of Tables

Acknowledgments

Say thank you for everybody involved directly and indirectly.

Dedication

For my family, mentors and all people who inspired me directly or indirectly in things I am doing.

I also dedicate this thesis to everyone who have supported me throughout the process.

Chapter 1

Introduction

The tale about software safety: why important, software everywhere, human life, etc. (info from 890 and Barnes' book).

Software Engineering for Real-Time and Safety-Critical systems is very different than creating Desktop applications. In both types of software we want to ensure correctnes and security. In case of e.g. e-mail client software assurance is not critical. When something happend, we just restart the app. However, in case of e.g. Airplane, software cannot just crash. If it crashes, then people die. Behind these reasons, we need different properties of programming language and its tools. For Web or Mobile apps our priority is Rapid Development. For Safety-Critical systems, the security and correctness is crucial.

Most important in Safety-Critical Systems: Hazard analysis (avoid, recover)! Hazard can cause: * Incident * Accident Accident - event, which cause loss (undesired) Incident - event, which not cause loss (but undesired), and could lead to accident Hazard + Environmental Conditions = Accident (loss) Event - state change

Overview of Safety-Critical System loop.

1.1 Motivation

There are many accidents where Medical Devices are involved. Very often, the reason is the lack of communication between different Medical Devices. [EXAMPLE ACCIDENT] The solution for such a problem is to create "Integrated Clinical Environment" (ICE). SAnToS Lab at Kansas State University is working on Medical Device Coordination Framework (MDCF), which is prototype implementation of ICE.

Devices working under MDCF will need to satisfy some requirements. To make Developer's life easier, the requirements will be not only in documentation, but also in code. The code will be generated from models. Model Driven Development in this case means we will have some base models for medical devices development and developer will extend and customize them. The same like you do File > 'New Java project' in Eclipse, we want to be able to do the same in e.g. GNAT Programming Studio: File > 'New Medical device project'. Model as specification/requirements.

PCA Pump is as an example of Medical Device, which ultimately will work under Medical Device Coordination Framework (MDCF), an open source framework under development by SAnToS Lab at Kansas State University and University of Pennsylvania. Summarizing, we want to be able to have MDCF, which coordinates Medical Devices. Additionally we want set of AADL/BLESS models, which can be automatically translated to SPARK Ada. These models will be base for Medical Devices Developers, who can extend and adjust them to implement specific devices. Why AADL? Because it describes hardware and software. It allows to validate that the software will work on some device. Why SPARK? Because it contains set of verification tools. Testing vs Verification (form 721 slides): Testing starts with a set of possible test cases, simulates the system on each input, and observes the behavior. In general, testing does not cover all possible executions. On the other hand, verification establishes correctness for all possible execution sequences. Techniques for Verification:

- Formal verification: prove mathematically that the program is correct – this can be difficult for large programs.

- Correctness by construction: follow a well- defined methodology for constructing programs.
- Model checking: enumerate all possible executions and states, and check each state for correctness.

SPARK is a subset of Ada language, which is easy to deal with it. In the future, when everything will be done (in case of proving perspective) in SPARK, it will (probably) be extended. Maybe finally, there will be no SPARK, but only Ada. Thus for now, SPARK is temporary subset of Ada for reasoning and correctness proving.

1.2 Goals

- learn about PCA Pump Infusion pumps properties
- SPARK Ada cross-compilation for ARM-device (BeagleBoard-xM)
- implement PCA Pump based on Brian's Requirement Document (using Ravenscar profile)
- propose AADL/BLESS to SPARK Ada mapping
- mock PCA Pump AADL/BLESS models in SPARK Ada (based on proposed mapping and implementation)
- implement not generated part (based on implementation)
- create AADL/BLESS to SPARK Ada translator
- Use SPARK toolset for software verification:
  - SPARK Examiner
  - SPARK Simplifier

- Proof Obligation Summarizer (POGS)
- GNATprove
- Sireum Kiasan

1.3 Contribution

Put all pieces together (SPARK, AADL, BLESS, ICE, PCA Pump) and analyze current state of target technologies. Review PCA Pump Requirements document Analyzed PCA Pump AADL models, then based on available resources proposed possible translation from AADL/BLESS to SPARK Ada. Created skeleton of generated SPARK code from AADL models. Implemented PCA Pump based on requirements document and skeleton code. Resolved some ambiguities and analyzed different implementation possibilities. Then the implementation is sort of proof that, this document and AADL models are base for future Infusion Pumps implementations. Presented SPARK 2005 Verification tools: its capabilities and issues. Created AADL/BLESS to SPARK Ada translator?

1.4 Organization

The thesis is organized in 8 chapters:

- Chapter 1 is the problem description and summary of contribution which was made.
- Chapter 2 is Background that gives details about Model Driven Development, SPARK Ada, AADL/BLESS, ICE and available tools for such environment.
- Chapter 3 describe patient-controlled analgesia (PCA) pump.
- Chapter 4 is about code generation from the model.
- Chapter 5 describes the implementation of PCA Pump Prototype. Faced issues and design decisions made.

- Chapter 6 describes verification of implemented PCA Pump Prototype.
- Chapter 7 summarizes all work which has been done in this thesis.
- Chapter 8 is the future work that can be done on this topic.

1.5 Terms and Acronyms

- **AADL** - Architecture Analysis & Design Language
- **BLESS** - Behavioral Language for Embedded Systems with Software
- **ICE** - Integrated Clinical Environment
- **MDCF** - Medical Device Coordination Framework
- **PCA** - Patient-Controlled Analgesia (pump)
- **AADL** - Architecture Analysis & Design Language
- **AADL** - Architecture Analysis & Design Language
- **AADL** - Architecture Analysis & Design Language
- **AADL** - Architecture Analysis & Design Language
- **AADL** - Architecture Analysis & Design Language

Chapter 2

Background

This chapter is brief introduction of all technologies and tools used in this thesis. It is SPARK Ada programming language and its tools (GNAT Programming Studio, Sireum Bakar, GNATprove), AADL modeling language, BLESS (AADL annex language). There is also overview of the context in which this work has been made: Integrated Clinical Environment standard (ICE) and PCA Pump (ICE compliant device). This is followed by main topic of the thesis: code generation from AADL and analysis of existing AADL translators (Ocarina, RAMSES).

2.1 Integrated Clinical Environment

Medical devices are safety-critical systems. Medical Devices Coordination Framework is an open, experimental ICE-compliant platform to bring together academic researchers, industry vendors, and government regulators. Medical Devices, which are ICE compliant can be connected to MDCF. It enables Medical Devices cooperation. [add some pictures etc.]

2.2 AADL

AADL stands for Architecture Analysis & Design Language. The aim of the AADL is to allow the description of Distributed Real-Time Embedded (DRE) systems by assembling separately developed blocks. Thus it focuses on the definition of clear block interfaces, and separates the implementations from those interfaces. AADL allows for the description of both software and hardware parts of a system [1].

AADL has its roots in DARPA [2] funded research. The first version (1.0) was approved in 2004 under technical leadership of Peter Feiler [3]. AADL is develop by SAE AADL committee [4]. AADL version 2.0 was published in January 2009. The most recent version (2.1) was published in September 2012 [5].

AADL is a language for Model-Based Engineering [FG13]. It can be represented in textual and graphical form. There are tools (like Osate 2.2.1), which transforms textual representation into graphical. There is also possiblity to represent AADL in XML (using 3rd party tools). An example AADL model called Thermometer is shown in graphical representation in figure 2.2 and in textual representation in listing 2.1.

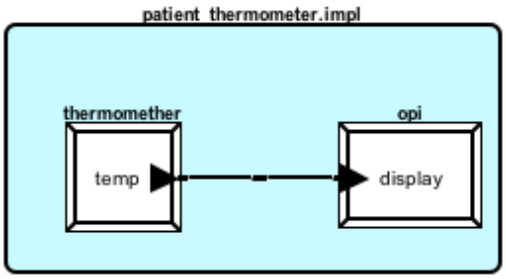

Figure 2.1: *AADL model of simple thermometer*

package Thermometer

[1] http://penelope.enst.fr/aadl
[2] http://www.darpa.mil
[3] http://wiki.sei.cmu.edu/aadl/index.php/The_Story_of_AADL/
[4] https://wiki.sei.cmu.edu/aadl/index.php/Main_Page
[5] https://wiki.sei.cmu.edu/aadl/index.php/Standardization

```
public
with Base_Types;
  system patient_thermometer
  end patient_thermometer;

  system implementation patient_thermometer.impl
  subcomponents
    thermomether : device thermometer_device.impl;
    opi : device operator_interface.impl;
  connections
    tdn : port thermomether.temp -> opi.display;
  end patient_thermometer.impl;

  device operator_interface
  features
    display : in data port Base_Types::Integer;
  end operator_interface;

  device implementation operator_interface.impl
  end operator_interface.impl;

  device thermometer_device
  features
    temp : out data port Base_Types::Integer;
  end thermometer_device;

  device implementation thermometer_device.impl
  end thermometer_device.impl;
end Thermometer;
```

Listing 2.1: *AADL model of simple thermometer*

Recently AADL becomes a new market standard. There are lots of tools for AADL models analysis, such as: STOOD [6], ADELE [7], Cheddar [8], AADLInspector [9] or Ocarina [10].

What is important, AADL is for architectural description. It should not be compared with UML suites, which allows to link with source code.

[6] http://www.ellidiss.com/products/stood
[7] https://wiki.sei.cmu.edu/aadl/index.php/Adele
[8] http://beru.univ-brest.fr/ singhoff/cheddar
[9] http://www.ellidiss.com/products/aadl-inspector
[10] http://www.openaadl.org

2.2.1 OSATE

Open Source AADL Tool Environment (OSATE) is a set of plug-ins on top of the open-source Eclipse platform. It provides a toolset for front-end processing of AADL models. OSATE is developed mainly by SEI (Software Engineering Institute - CMU) [11]. Latest available version of OSATE in the time when this work was published is OSATE2 [12].

2.3 BLESS

BLESS (Behavior Language for Embedded Systems with Software) is AADL annex sub-language defining behavior of components. The goal of BLESS is automatically-checked correctness proofs of AADL models of embedded electronic systems with software.

BLESS contains three AADL annex sublanguages:

- Assertion - it can be attached individually to AADL features (e.g. ports)
- subBLESS - can be attached only to subprograms; it has only value transformations and Assertions without time expressions
- BLESS - it can be attached to AADL thread, device or system components; it contains states, transitions, timeouts, actions, events and Assertions with time expressions...

How it fits into the picture. Why it was developed. Corectness prove in AADL + behavior [LCH13], from which we can generate SPARK Ada code.

2.4 SPARK Ada

First version of Ada programming language - Ada 83 - was designed to meet the US Department of Defence Requirements formalized in "Steelman" document [13]. Since that time,

[11] http://www.aadl.info/aadl/currentsite/tool/osate.html
[12] https://wiki.sei.cmu.edu/aadl/index.php/Osate_2
[13] http://www.adahome.com/History/Steelman/steelman.htm

Ada evolved. There were Ada 95, Ada 2005 and Ada 2012 (released in December 10, 2012) [14]. Ada is actively used in many Real-World projects[15], e.g. Aviation (Boeing [16]), Railway Transportation, Commercial Rockets, Satellites and even Banking. One of the main goals of Ada is to ensure software correctness and safety.

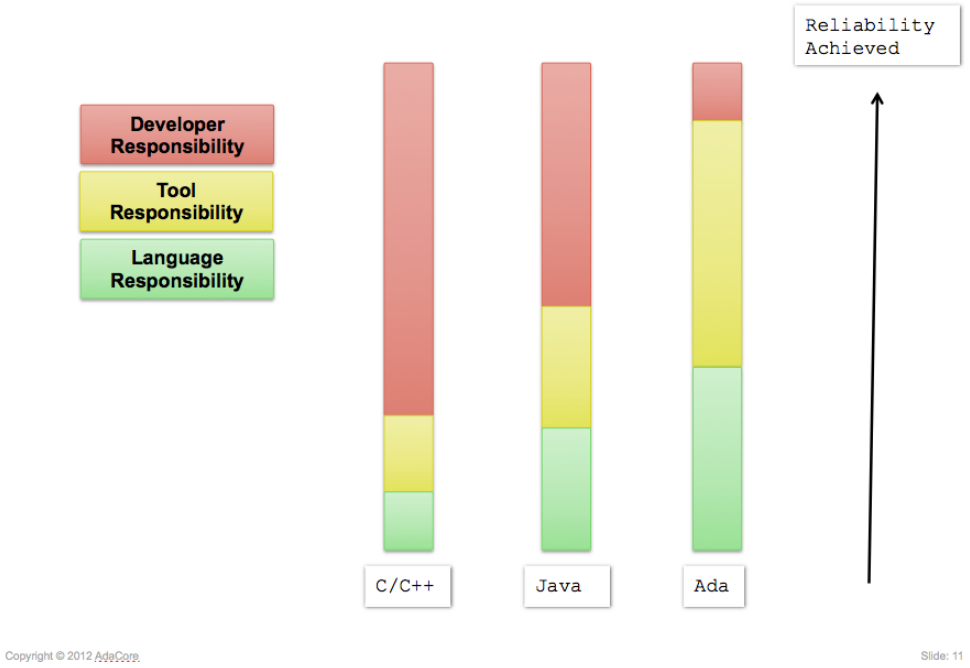

Figure 2.2: *Developer responsibility in Ada*[17].

SPARK is a programming language and static verification technology designed specifically for the development of high integrity software. It is a "safe" subset of Ada designed to be susceptible to formal methods, accompanied with a set of approaches and tools. Using SPARK, a developer takes a Z specification and performs a stepwise refinement from the specification to SPARK code. For each refinement step a tool is used to produce verification conditions (VC's), which are mathematical theorems. If the VC's can be proved then the

[14] http://www.ada2012.org
[15] http://www.seas.gwu.edu/ mfeldman/ada-project-summary.html
[16] http://archive.adaic.com/projects/atwork/boeing.html
[17] http://www.slideshare.net/AdaCore/ada-2012

refinement step will be known to be valid. However if the VC's cannot be proved then the refinement step may be erroneous [18].

First version was designed over 20 years ago. SPARK has established a track record of use in embedded and critical systems across a diverse range of industrial domains where safety and security are paramount [Bar13].

SPARK provides a significant degree of automation in proving exception freedom [IEC+06]. SPARK excludes some Ada constructs to make static analysis feasible [IEC+06]. Additionally SPARK contains tool-set for Software Verification:

- Examiner - analyze code and ensures that it conforms to the SPARK language; also verify program to some extent using Verification Conditions (VC)
- Simplifier - simplify Verification Conditions generated by Examiner
- Proof Checker - prove the Verification Conditions

First version of SPARK was based on Ada 83. The second version (SPARK 95) - on Ada 95. SPARK 2005 is based on Ada 2005. It is a subset of Ada 2005 with annotations. The annotation language support flow analysis and formal verification. Annotations are encoded in Ada comments (via the prefix --#). It makes every SPARK 2005 program, valid Ada 2005 program. Figure 2.2 shows example SPARK 2005 package specification.

```
package Odometer
--# own Trip, Total : Integer;
is
  procedure Zero_Trip;
  --# global out Trip;
  --# derives Trip from ;
  --# post Trip = 0;

  function Read_Trip return Integer;
  --# global in Trip;

  function Read_Total return Integer;
  --# global in Total;
```

[18] http://www.dwheeler.com/lovelace/s17s4.htm

```
   procedure Inc;
   --# global in out Trip, Total;
   --# derives Trip from Trip & Total from Total;
   --# post Trip = Trip~ + 1 and Total = Total~ + 1;

end Odometer;
```

Listing 2.2: *SPARK 2005 code: Odometer [Bar13]*

SPARK 2005 does not include constructs such as pointers, dynamic memory allocation or recursion [IEC+06].

SPARK 2014 [19] is based on Ada 2012 programming language targeted at safety- and security-critical applications [DEL+14]. Since Ada 2012 contains contracts, there is no need to use annotations like in SPARK 2005. Thus SPARK 2014 is subset of Ada 2012. It contains all features of Ada 2012 except:

- Access types (pointers)
- Exceptions
- Aliasing between variables
- Concurrency features of Ada (Tasking) - it's part of SPARK 2014 road-map to include support for tasking in the future, although likely not this year
- Side effects in expressions and functions

Sample mapping from SPARK 2005 to 2014 is shown on table 2.1. Complete mapping can be found in SPARK 2014 documentation [20] [AL14].

SPARK 2014 does not contains Examiner. Instead, proofs are made by gnatPROVE. The notion of executable contracts in Ada 2012, was inspired by SPARK. The previous Odometer example in SPARK 2014 is shown in figure 2.3.

[19] http://www.spark-2014.org
[20] http://docs.adacore.com/spark2014-docs/html/lrm/mapping-spec.html

Table 2.1: *Sample SPARK 2005 to 2014 mapping.*

SPARK 2005	SPARK 2014
--# global in out X, Y;	**with** Global => (In_Out => (X, Y));
--# derives X from Y & --# Y from X;	Depends => (X => Y, Y => X);
--# pre Y /= 0 and --# X > Integer'First;	**with** Pre => Y /= 0 **and** X > Integer'First;
--# post X = Y~ and Y = X~;	**with** Post => (X = Y'Old **and** Y = X'Old);

```
package Odometer
with SPARK_Mode
Abstract_State => (Trip, Total)
is
  procedure Zero_Trip
  with Global => (Output => (Trip)),
     Depends => (Trip => null),
     Post => (Trip = 0);

  function Read_Trip return Integer
  with Global => (Input => (Trip));

  function Read_Total return Integer
  with Global => (Input => (Total));

  procedure Inc
  with Global => (In_Out => (Trip, Total)),
    Depends => (Trip => Trip, Total => Total),
    Post => Trip = Trip'Old + 1 and Total = Total'Old + 1;

end Odometer;
```

Listing 2.3: *SPARK 2014 code: Odometer*

Fundamental SPARK contracts:

Table 2.2: Fundamental SPARK annotations

SPARK 2005	SPARK 2014	Description
−−# global	Global	list of used global variables within subprogram
−−# derives	Depends	describe dependencies between variables
−−# own	Abstract_State	declare variables defined in package body
−−# initializes	initializes	indicates variables, which are initialized
−−# inherit	not needed	allows to access entities of other packages
−−# pre	Pre	pre condition
−−# post	Post	post condition
−−# assert	Assert	assertion

It is possible to mix SPARK 2014 with Ada 2012. However, only the part which is SPARK 2014 compliant will be verified. Usually SPARK is used in the most critical parts of Software Systems [Cha00]. It means, that some part is written in e.g. Ada or C++ and the rest in SPARK. The reason of that is the SPARK limitation and lack of necessity to verify some modules.

The most popular IDE for SPARK Ada is GNAT Programming Studio [21].

There is also plugin for Eclipse: GNATbench [22] created by AdaCore. Tools for correctness proving.

2.4.1 GNAT compiler and Programming Studio

GNAT compiler is front end of gcc... IDE for SPARK Ada programs development. Includes proving tools. E.g. Sireum Bakar (developed by SAnToS lab) or GNATprove.

2.4.2 Ravenscar Tasking Subset

RavenSPARK is subset of the SPARK Ravenscar Profile (which is subset of Ada tasking). The Ravenscar Profile provides a subset of the tasking facilities of Ada95 and Ada 2005 suitable for the construction of high-integrity concurrent programs [Tea12].

The Ravenscar Profile is a subset of the tasking model, restricted to meet the real-time community requirements for determinism, schedulability analysis and memory-boundedness, as well as being suitable for mapping to a small and efficient run-time system that supports task synchronization and communication, and which could be certifiable to the highest integrity levels. The concurrency model promoted by the Ravenscar Profile is consistent with the use of tools that allow the static properties of programs to be verified. Potential verification techniques include information flow analysis, schedulability analysis, execution-order analysis and model checking. These techniques allow analysis of a system to be performed throughout its development life cycle, thus avoiding the common problem of finding only during system integration and testing that the design fails to meet its non-functional requirements. [AB04]

Concurrent programs require the use of different specification and verification techniques

[21] http://libre.adacore.com/tools/gps
[22] https://www.adacore.com/gnatpro/toolsuite/gnatbench/

from sequential programs. For this reason, tasks, protected units and objects, and synchronization features are currently excluded from SPARK 2014 [23] [AL14].

To create a task, the task type has to be declared and task variable of this type. Ravenscar does not allow dynamic task creation. Thus, all tasks have to exists for the full lifetime of the program. [AW01] Tasks can be declared only in packages. Not in subprograms or in other tasks. [Bar13] The priority of each tasks has to be specified by **pragma** Priority. [what is priorities range?] Listing 2.4 shows sample package with two tasks.

```
package Some_Pkg
--# own task t1 : Task1;
--#     task t2 : Task2;
is
  task type Task1
  is
    pragma Priority(10);
  end Task1;

  task type Task2
  is
    pragma Priority(9);
  end Task2;

end Some_Pkg;
```

Listing 2.4: *Sample tasks*

Declared tasks have to be implemented in the package body (listing 2.5).

```
package body Some_Pkg
is
  t1 : Task1;
  t2 : Task2;

  task body Task1
  is
  begin
    loop
      -- implementation;
    end loop;
  end Task1;

  task body Task2
```

[23] http://docs.adacore.com/spark2014-docs/html/lrm/tasks-and-synchronization.html

```
  is
  begin
    loop
      -- implementation;
    end loop;
  end Task2;

end Some_Pkg;
```

Listing 2.5: *Sample tasks body*

There are two ways to access variable in different tasks:

- It has to be protected object
- It has to be atomic type

Protected object encapsulate variable, in such a way that it is accessible, only through protected subprograms. This mechanism use locking, to ensure atomicity. Protected type declaration is similar to task: specification and body has to be defined. Listing 2.6 shows sample tasks with protected type Integer_Store, which enable to share Integer variable between tasks. What is important, protected type has to be declared before tasks, which will use it. Otherwise, it will not be visible for them.

```
package Some_Pkg
--# own protected Shared_Var : Integer_Store (Priority => 11);
--#     task t1 : Task1;
--#     task t2 : Task2;
is
   protected type Integer_Store
   is
      pragma Priority (11);

      function Get return Integer;
      --# global in Integer_Store;

      procedure Put(X : in Integer);
      --# global out Integer_Store;
      --# derives Integer_Store from X;
   private
      TheStoredData : Integer := 0;
   end Integer_Store;
```

```
   task type Task1
     --# global out Shared_Var;
   is
      pragma Priority(10);
   end Task1;

   task type Task2
     --# global in Shared_Var;
   is
      pragma Priority(9);
   end Task2;

end Some_Pkg;
```

Listing 2.6: *Sample tasks with protected object*

Protected type body also has to be defined in package body (listing 2.7).

```
package body Some_Pkg
is
   Shared_Var : Integer_Store;
   t1 : Task1;
   t2 : Task2;

   protected body Integer_Store is
      function Get return Integer
      --# global in TheStoredData;
      is
      begin
         return TheStoredData;
      end Get;

      procedure Put(X : in Integer)
      --# global out TheStoredData;
      --# derives TheStoredData from X;
      is
      begin
         TheStoredData := X;
      end Put;
   end Integer_Store;

   task body Task1
   is
   begin
      loop
         Shared_Var.Put(5);
      end loop;
   end Task1;

   task body Task2
```

```
   is
      Local_Var : Integer;
   begin
      loop
         Local_Var := Shared_Var.Get;
      end loop;
   end Task2;

end Some_Pkg;
```

Listing 2.7: *Sample tasks with protected object body*

Task1 is writing to Shared_Var and Task2 is reading Shared_Var. The highest priority is assigned to protected object, to ensure atomicity during operations on it. The lowest priority is assigned to Task2, which is reading Shared_Var. Reading is usually less expensive operation than writing. Thus, to avoid starvation, Task1 has higher priority than Task2. Notice, that Shared_Var is declared in package body, but refined in package specification.

Protected variables may not be used in proof contexts. Thus, if we try to use protected variable in proofs (pre- or postcondition), then SPARK Examiner returns Semantic Error 940 – Variable **is** a **protected** own variable. **Protected** variables may **not** be used **in** proof contexts.. Formal reasoning about interactions and especially temporal properties require other techniques such as model checking and lie outside the scope of SPARK [Bar13]. To preserve opportunity to use pre- and postconditions, atomic types have to be used.

To declare atomic type, we have to use **pragma** Atomic. However, there is restriction, that **pragma** Atomic cannot be applied to predefined type such as Integer. Thus, we have to define our custom type (which can be just rename of Integer) and apply **pragma** Atomic on this type. Listing 2.8 presents previous example with atomic types instead of protected objects.

```
package Some_Pkg
--# own Shared_Var;
--#     task t1 : Task1;
--#     task t2 : Task2;
--# initializes Shared_Var;
is
```

```
   type Int32 is new Integer;

   task type Task1
     --# global out Shared_Var;
   is
      pragma Priority(10);
   end Task1;

   task type Task2
     --# global in Shared_Var;
   is
      pragma Priority(9);
   end Task2;

end Some_Pkg;

package body Some_Pkg
is
   Shared_Var : Int32 := 0;
   t1 : Task1;
   t2 : Task2;

   task body Task1
   is
   begin
      loop
         Shared_Var := 5;
      end loop;
   end Task1;

   task body Task2
   is
      Local_Var : Integer;
   begin
      loop
         Local_Var := Integer(Shared_Var);
      end loop;
   end Task2;

end Some_Pkg;
```

Listing 2.8: *Sample tasks with atomic type*

Be aware that **pragma** atomic does not guaranty atomicity. In most cases, atomic types should not be used for tasking. Instead, protected types should be used.

Another important thing in tasking is Time library: Ada.Real_Time. It allows to run task periodically, using **delay until** statement, which suspends task until specified time. To use

delay in the task, it has to be declared in **declare** annotation: --# declare delay; [Bar13].

Details about tasking in SPARK are well described in Chapter 8 of Barnes' book [Bar13]. The "Guide for the use of the Ada Ravenscar profile in high integrity systems" [AB04] and the official Ravenscar Profile documentation (which includes examples) [Tea12] might be useful as well. The limitations of Tasking in SPARK are reviewed in Audsley's and Welllings' paper [AW01].

2.4.3 AUnit

AUnit is Unit Test Framework for Ada language. It can be also applied for verify SPARK Ada programs. AUnit tutorials [Fal14] AUnit Cookbook [Ada14]

2.4.4 Sireum Bakar

Overview: symbolic execution, Pilar, Kiasan and Alir [Thi11]. Sireum Kiasan [BHR+11] is a tool, which use symbolic execution for finding possible paths in program. Plugin for GNAT Programming Studio (SPARK 2005 and 2014 under development). Plugin for Eclipse (only for SPARK 2005). No support for Ravenscar profile. Separated sequential parts can be verified (Odometer?). Sequential version of Max_Drug_Per_Hour_Watcher?

2.4.5 GNAT Prove

GNATprove [24] is a formal verification tool for SPARK 2014 programs. It interprets SPARK Ada annotations exactly like they are interpreted at run time during tests.

[24] http://www.open-do.org/projects/hi-lite/gnatprove/

2.5 AADL/BLESS to SPARK Ada code generation

The ultimate goal of long term research, this thesis is part of, is AADL (with BLESS) to SPARK Ada translation.

2.5.1 Ocarina

Ocarina [LZPH09, LZPH09] generates code from an AADL architecture model to an Ada application running on top of PolyORB framework. In this context, PolyORB acts as both the distribution middleware and execution runtime on all targets supported by PolyORB. It generate Ada 2005 and C code. Since mid-2009, Telecom ParisTech is no longer involved in Ocarina, and is developing another AADL tool-chain, based on Eclipse, codenamed RAMSES [Hug13].

examples on github

run: ocarina -x scenario.aadl

2.5.2 Ramses

RAMSES is a model transformation framework dedicated to the refinement of AADL models. It contains code generation plug-in.

Chapter 3

PCA Pump

Description of PCA Pump, its functions, problems and how ICE can solve them. Requirements document [Lar14]. Requirements document overview [LHC13].

In this thesis, only the operation module is implemented.

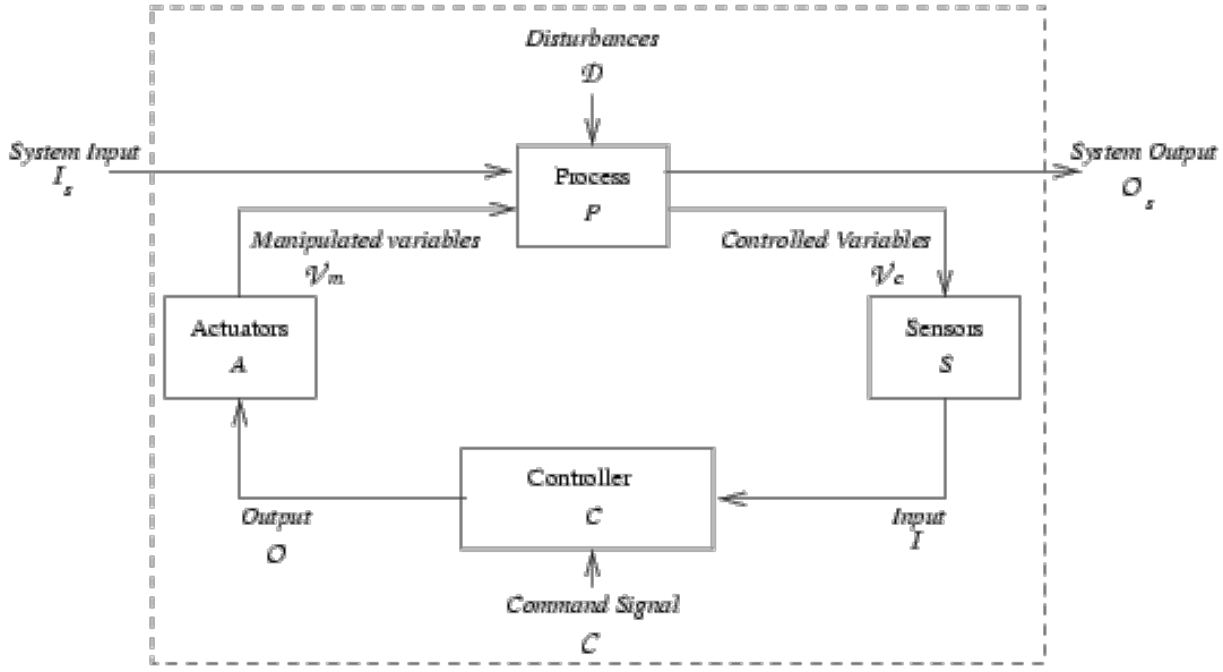

Figure 3.1: *Basic Process Control Loop*[1].

Pump internal implementation based on [Med10]. - basal dose deliver in increments - easier to track delivered amount (page 14)

3.1 PCA Pump Requirements Document

Selected use cases for implementation?

3.2 PCA Pump AADL/BLESS Models

Selected modules for implementation. Pictures etc.

3.3 BeagleBoard-XM

First step was create PCA Pump prototype on BeagleBoard-xM.

BeagleBoard-xM is Embedded device with AM37x 1GHz ARM processor (Cortex-A8 compatible). It has 512 MB RAM, 4 USB 2.0 ports, HDMI port, 28 General-purpose input/output (GPIO) ports and Linux Operating System (on microSD card). Moreover there is PWM support. All these properties makes this device good candidate for prototyping PCA Pump.

Pulse width modulation (PWM) is a technique for controlling analog circuits with a processor's digital outputs.

Expansion port 14(PWM) and 28(GND?) GPIO158 Java Program to Run the pump for 10 seconds

There is no existing SPARK Ada compiler running on ARM system. Hence, to compile SPARK Ada program for ARM device, we need to perform cross-compilation on other

[1]http://www.safeware-eng.com/system and software safety publications/Designing Specification Languages.htm

machine. There is GNAT compiler [Hor09] created by AdaCore, but there was no cross-compiler for ARM. However AdaCore was working on it. They had working version in 2013, but tested only on their target, Android-based device. BeagleBoard-xM is coming with Linux Angstrom Operating System. There is possibility to install Android on BeagleBoard-xM, but still not warranty everything will be working. Cooperation with AdaCore allowed to cross-compile SPARK Ada program for BeagleBoard-xM.

Include source of simple program? GNAT cross-compiler only for Linux Platform (cross-compilation has to be done on Linux).

compilation+linking command: arm−linux−gnueabi−gnatmake −d −Ppca_ravenscar.gpr.

3.4 Interface for Integrated Clinical Environment

PCA Pump will be connected to ICE. It will allow to monitor and control device by MDCF (ICE implementation). Describe communication with MDCF/ICE. PCA Pump ports for that etc.

Chapter 4

AADL/BLESS to SPARK Ada translation

First step was to create mock (based on doc, aadl models and implemented PCA Pump). Prototyping Embedded Systems using AADL lasts for a few years [CB09].

4.1 AADL/BLESS to SPARK Ada mapping

Mapping is driven by "Architecture analysis & Design Language (AADL) V2 Programming Language Annex Document" [SCD14]. This document was discussed during AADL User Days in Valencia (February 2013)[1] and in Jacksonville, FL (April 2013)[2]. Ocarina tool suite (based on older AADL annex documents [HZPK08]) and its examples[3] was also helpful in understanding of AADL to Ada translation. Only high level mapping is done. No implementation (thread interactions) like Ocarina does.

[1] http://www.aadl.info/aadl/downloads/committee/feb2013/presentations/13_02_04-AADL-Code%20Generation.pdf

[2] https://wiki.sei.cmu.edu/aadl/images/8/8a/Constraint_Annex_April22.v3.pdf

[3] https://github.com/yoogx/polyorb-hi-ada/tree/master/examples/aadlv2

4.1.1 Data types mapping

One of core AADL packages is Base_Types. It defined fundamental datatypes for AADL. Its definition is shown on listing 4.1.

```
package Base_Types
public

  with Data_Model;

  data Boolean
  properties
    Data_Model::Data_Representation => Boolean;
  end Boolean;

  data Integer
  properties
    Data_Model::Data_Representation => Integer;
  end Integer;

  -- Signed integer of various byte sizes

  data Integer_8 extends Integer
  properties
    Data_Model::Number_Representation => Signed;
    Source_Data_Size => 1 Bytes;
  end Integer_8;

  data Integer_16 extends Integer
  properties
    Data_Model::Number_Representation => Signed;
    Source_Data_Size => 2 Bytes;
  end Integer_16;

  data Integer_32 extends Integer
  properties
    Data_Model::Number_Representation => Signed;
    Source_Data_Size => 4 Bytes;
  end Integer_32;

  data Integer_64 extends Integer
  properties
    Data_Model::Number_Representation => Signed;
    Source_Data_Size => 8 Bytes;
  end Integer_64;

  -- Unsigned integer of various byte sizes

  data Unsigned_8 extends Integer
```

```
properties
  Data_Model::Number_Representation => Unsigned;
  Source_Data_Size => 1 Bytes;
end Unsigned_8;

data Unsigned_16 extends Integer
properties
  Data_Model::Number_Representation => Unsigned;
  Source_Data_Size => 2 Bytes;
end Unsigned_16;

data Unsigned_32 extends Integer
properties
  Data_Model::Number_Representation => Unsigned;
  Source_Data_Size => 4 Bytes;
end Unsigned_32;

data Unsigned_64 extends Integer
properties
  Data_Model::Number_Representation => Unsigned;
  Source_Data_Size => 8 Bytes;
end Unsigned_64;

data Natural extends Integer
properties
  Data_Model::Integer_Range => 0 .. Max_Target_Integer;
end Natural;

data Float
properties
  Data_Model::Data_Representation => Float;
end Float;

data Float_32 extends Float
properties
  Data_Model::IEEE754_Precision => Simple;
  Source_Data_Size => 4 Bytes;
end Float_32;

data Float_64 extends Float
properties
  Data_Model::IEEE754_Precision => Double;
  Source_Data_Size => 8 Bytes;
end Float_64;

data Character
properties
  Data_Model::Data_Representation => Character;
end Character;

data String
```

```
  properties
    Data_Model::Data_Representation => String;
  end String;

end Base_Types;
```

Listing 4.1: *AADL Base_ Types package*

In Ada 2012, and thus SPARK 2014, there is package Interfaces, which allows for easy mapping of AADL Base_Types package. Mapping proposed in Annex Document [SCD14] is presented on listing 4.2.

```
with Interfaces;

package Base_Types is
  type AADL_Boolean is new Standard.Boolean;
  type AADL_Integer is new Standard.Integer;
  type Integer_8 is new Interfaces.Integer_8;
  type Integer_16 is new Interfaces.Integer_16;
  type Integer_32 is new Interfaces.Integer_32;
  type Integer_64 is new Interfaces.Integer_64;
  type Unsigned_8 is new Interfaces.Unsigned_8;
  type Unsigned_16 is new Interfaces.Unsigned_16;
  type Unsigned_32 is new Interfaces.Unsigned_32;
  type Unsigned_64 is new Interfaces.Unsigned_64;
  type AADL_Natural is new Standard.Integer; -- XXX incomplete range?
  type AADL_Float is new Standard.Float;
  type Float_32 is new Interfaces.IEEE_Float_32;
  type Float_64 is new Interfaces.IEEE_Float_64;
    type AADL_Character is new Standard.Character;
end Base_Types;
```

Listing 4.2: *Mapping of Base_ Types for SPARK 2014*

Mapping for SPARK 2005: Integer, Natural, Boolean already defined in SPARK. Types Float, Character and String are not part of this thesis, because of verification tools limitation. Thus, in this thesis only Integer, Enumeration, Boolean and Record types are analyzed.

Each type is translated into simple type definition and protected type. Then it can be used in multitask programs with Ravescar Profile. For every protected type only setter (Put) and getter (Get) subprograms are defined. It can be extended by developer during

development phase.

Types: Integer, Boolean and Natural are already defined in SPARK Ada, thus only protected objects are generated for them.

Sample AADL Base_Types mapping to SPARK Ada is presented in table 4.1.

Table 4.1: Base AADL types to SPARK mapping.

AADL	SPARK Ada

data Integer
properties
  Data_Model::
  Data_Representation =>
  Integer;
end Integer;
	protected type Integer_Store
	  is
	  pragma Priority (10);
	
	  function Get **return** Integer;
	  −−# global in Integer_Store;
	
	  procedure Put(X : **in** Integer);
	  −−# global out Integer_Store;
	  −−# derives Integer_Store from X;
	  private
	  TheStoredData : Integer := 0;
	  end Integer_Store;

Continued on next page

Table 4.1 – continued from previous page

AADL	SPARK Ada

data Integer_16 extends Integer
properties
  Data_Model::
  Number_Representation
  => Signed;
  Source_Data_Size => 2
  Bytes;
end Integer_16;
	type Integer_16 **is new** Integer **range** −2**(2*8−1) ..
	  2**(2*8−1−1);
	
	protected type Integer_16_Store
	  is
	  pragma Priority (10);
	
	  function Get **return** Integer_16;
	  −−# global in Integer_16_Store;
	
	  procedure Put(X : **in** Integer_16);
	  −−# global out Integer_16_Store;
	  −−# derives Integer_16_Store from X;
	  private
	  TheStoredData : Integer_16 := 0;
	  end Integer_16_Store;
	
	  protected body Integer_16_Store **is**
	  function Get **return** Integer_16
	  −−# global in TheStoredData;
	  is
	  begin
	  return TheStoredData;
	  end Get;
	
	  procedure Put(X : **in** Integer_16)
	  −−# global out TheStoredData;
	  −−# derives TheStoredData from X;
	  is
	  begin
	  TheStoredData := X;
	  end Put;
	  end Integer_16_Store;

Continued on next page

Table 4.1 – continued from previous page

AADL	SPARK Ada

data Unsigned_16 extends
  Integer
properties
  Data_Model::
  Number_Representation
  => Unsigned;
  Source_Data_Size => 2
  Bytes;
end Unsigned_16;
	type Unsigned_16 **is new** Integer **range** 0 .. 2**(2*8−1);
	
	  protected type Unsigned_16_Store
	  is
	  pragma Priority (10);
	
	  function Get **return** Unsigned_16;
	  −−# global in Unsigned_16_Store;
	
	  procedure Put(X : **in** Unsigned_16);
	  −−# global out Unsigned_16_Store;
	  −−# derives Unsigned_16_Store from X;
	  private
	  TheStoredData : Unsigned_16 := 0;
	  end Unsigned_16_Store;
	
	  protected body Unsigned_16_Store **is**
	  function Get **return** Unsigned_16
	  −−# global in TheStoredData;
	  is
	  begin
	  return TheStoredData;
	  end Get;
	
	  procedure Put(X : **in** Unsigned_16)
	  −−# global out TheStoredData;
	  −−# derives TheStoredData from X;
	  is
	  begin
	  TheStoredData := X;
	  end Put;
	  end Unsigned_16_Store;

Type range is defined using AADL properties: Data_Model::Number_Representation and Source_Data_Size. In case of Integer representation range starts from negative value, for Unsigned: from 0. Maximum value for Integer is calculated using following formula presented on equation 4.1. The minimum value formula for Integer (4.2) and maxiumum value for Unsigned (4.3) use similar strategy.

$$Integer\_[Number\_Of\_Bytes * 8]\_Max = 2^{\text{Number\_Of\_Bytes}*8-1} - 1 \tag{4.1}$$

$$Integer\_[Number\_Of\_Bytes * 8]\_Min = -2^{\text{Number\_Of\_Bytes}*8-1} \tag{4.2}$$

$$Unsigned\_[Number\_Of\_Bytes * 8]\_Max = 2^{\text{Number\_Of\_Bytes}*8} - 1 \tag{4.3}$$

Mapping for enumeration types is presented on table 4.2. BLESS properties are ignored in translation.

Table 4.2: AADL/BLESS enumeration types to SPARK mapping.

AADL	SPARK Ada

data Enum_Type
  properties
  BLESS::Typed=>"enumeration (
  Enumerator1, Enumerator2, Enumerator3)";
  Data_Model::Data_Representation =>
  Enum;
  Data_Model::Enumerators => ("
  Enumerator1", "Enumerator2", "
  Enumerator3");
end Enum_Type;
	type Enum_Type **is** (Enumerator1,
	  Enumerator2, Enumerator3);
	
	protected type Enum_Type_Store
	  is
	  pragma Priority (10);
	
	  function Get **return** Enum_Type;
	  −−# global in Enum_Type_Store;
	
	  procedure Put(X : **in** Enum_Type);
	  −−# global out Enum_Type_Store;
	  −−# derives Enum_Type_Store from X;
	  private
	  TheStoredData : Enum_Type :=
	  Enumerator1;
	  end Enum_Type_Store;
	
	  protected body Enum_Type_Store **is**
	  function Get **return** Enum_Type
	  −−# global in TheStoredData;
	  is
	  begin
	  return TheStoredData;
	  end Get;
	
	  procedure Put(X : **in** Enum_Type)
	  −−# global out TheStoredData;
	  −−# derives TheStoredData from X;
	  is
	  begin
	  TheStoredData := X;
	  end Put;
	  end Enum_Type_Store;

Sometimes it is pragmatic to define a type, which has exactly the same range like some already existing type. Especially when it is used for some specific calculations. E.g. measuring the speed. Let's say, that Unsigned_16 was used. Then, during development of new

car model, it is not enough. In case when e.g. Speed_Type is not defined, there are two options. First: change definition (range) of Unsigned_16. That is bad choice, especially because its name specify the range. Another reason: it might be used not only for measuring the Speed, but maybe also for fuel level, which range is still fine. Second option is to change Unsigned_16 to e.g. Unsigned_32 everywhere in Speed Control Module (and maybe also in some external modules). When Speed_Type is defined and used everywhere for speed units, then only definition of Speed_Type has to be changed. To define type, using exising type in AADL Data_Model::Base_Type property is used. Translation to SPARK Ada is shown in 4.3.

Table 4.3: AADL types to SPARK mapping: Subtypes.

AADL	SPARK Ada
data Speed_Type **properties** BLESS::Typed=>"integer"; Data_Model::Base_Type => (classifier(Base_Types::Unsigned_16)); **end** Speed_Type;	**subtype** Speed_Type **is** Base_Types. Unsigned_16;

Using property Data_Model::Data_Representation array type in AADL can be defined. In addition to that, size fo array has to be specified by Data_Model::Dimension property. Sample mapping of array of 10 integers is shown in table 4.4.

Table 4.4: AADL arrays to SPARK mapping.

AADL	SPARK Ada

data Some_Array
  properties
  BLESS::Typed => "array [10] of
  Base_Types::Integer_32";
  Data_Model::Data_Representation =>
  Array;
  Data_Model::Base_Type => (classifier(
  Base_Types::Integer_32));
  Data_Model::Dimension => (10);
end Some_Array;
	subtype Some_Array_Index **is** Integer **range** 1
	  .. 10;
	type Some_Array **is array** (
	  Some_Array_Index) **of** Base_Types.
	  Integer_32;

AADL v2 allows to create struct data types, using Data_Model::Data_Representation => Struct. AADL Struct is mapped to SPARK Ada record type. The mapping is presented in table 4.5.

Table 4.5: AADL structs to SPARK Ada records mapping.

AADL	SPARK Ada

data Some_Record_Type
  properties
  BLESS::Typed => "record (
  Field1 : Base_Types::Integer_32;
  Field2 : Base_Types::Boolean;
  Field3 : Base_Types::Unsigned_32;
  );
  Data_Model::Data_Representation =>
  Struct;
  Data_Model::Element_Names => ("Field1",
  "Field2", "Field3");
  Data_Model::Base_Type =>
  (
  classifier(Base_Types::Integer_32),
  classifier(Base_Types::Boolean),
  classifier(Base_Types::Unsigned_32)
  );
end Some_Record_Type;
	type Some_Record_Type **is record**
	  Field1 : Integer_32;
	  Field2 : Boolean;
	  Field3 : Unsigned_32;
	  end record;

During AADL/BLESS to SPARK Ada types mapping, SPARK Examiner was helpful. Eg. it detected redundancy in enumerators. Both Alarm_Type and Warning_Type contained No_Alarm enumerators, which was a bug. Warning_Type should have No_Warning enumerator instead.

4.1.2 AADL ports mapping

Proposed ports mapping shown in table 4.6 is based on AADL runtime services from Annex 2 to "Programming Language Annex Document" [SCD14]. Additionaly, the mapping contains SPARK 2005 contracts.

Table 4.6: AADL to SPARK ports mapping.

AADL/BLESS	SPARK Ada
Port_Name : **in data port** Port_Type;	-- spec (.ads): **procedure** Receive_Port_Name; --# global out Port_Name; -- body (.adb): Port_Name : Port_Type; **procedure** Receive_Port_Name **is** **begin** -- TODO: implement receiving Port_Name value -- e.g.: -- Port_Name := Some_Pkg.Get_Port_Name; **end** Receive_Port_Name;
Port_Name : **out data port** Port_Type;	-- spec (.ads) **procedure** Get_Port_Name(Port_Name_Out : Port_Type); --# global in Port_Name; --# derives Port_Name_Out from Port_Name; -- body (.adb): Port_Name : Port_Type; **procedure** Get_Port_Name(Port_Name_Out : Port_Type) **is** **begin** Port_Name_Out := Port_Name; **end** Get_Port_Name;
Port_Name : **in event port**;	-- spec (.ads) **procedure** Put_Port_Name; -- body (.adb): **procedure** Put_Port_Name **is** **begin** -- TODO: implement event handler **end** Put_Port_Name;

Continued on next page

Table 4.6 – continued from previous page

AADL/BLESS	SPARK Ada
Port_Name : **out event port**;	-- spec (.ads) **procedure** Send_Port_Name; -- body (.adb): **procedure** Send_Port_Name **is** **begin** -- TODO: implement receiving Port_Name value -- e.g.: -- Some_Pkg.Put_Port_Name; **end** Send_Port_Name;
Port_Name : **in event data port** Port_Type;	-- spec (.ads) **procedure** Put_Port_Name(Port_Name_In : Port_Type); --# global out Port_Name; --# derives Port_Name from Port_Name_In; -- body (.adb): Port_Name : Port_Type; **procedure** Put_Port_Name (Port_Name_In : Port_Type) **is** **begin** Port_Name := Port_Name_In; **end** Put_Port_Name;

Continued on next page

Table 4.6 – continued from previous page

AADL/BLESS	SPARK Ada
Port_Name : **out event data port** Port_Type;	-- spec (.ads) **procedure** Send_Port_Name; --# global in Port_Name; -- body (.adb): Port_Name : Port_Type; **procedure** Send_Port_Name **is** **begin** -- TODO: implement receiving Port_Name value -- e.g.: -- Some_Pkg.Put_Port_Name(Port_Name); **end** Send_Port_Name;

There is a problem: "consumer.ads:1:13: Semantic Error 135 - The package Producer is undeclared or not visible, or there is a circularity in the list of inherited packages.".

4.1.3 Thread to task mapping

AADL Threads are mapped into SPARK Ada tasks according to table 4.7.

4.1.4 Subprograms mapping

4.1.5 Feature groups mapping

In SPARK Ada there are nested packages and child packages. Sample nested packages are shown in listing 4.3. Equivalent child packages are shown in listing 4.4. The name of a child package consists of the parent unit's name followed by the child package's identifier, separated by a period (dot) '.'. Calling convention is the same for child and nested packages

Table 4.7: *AADL threads to SPARK Ada tasks mapping.*

AADL/BLESS	SPARK Ada

package Some_Pkg
  thread Some_Thread
  features
  Some_Port : **out data port** Port_Type;
  end Some_Thread;
end Some_Pkg;
	package Some_Pkg
	is
	  task type Some_Thread
	  −−# global out Some_Port;
	  is
	  pragma Priority(10);
	  end Some_Thread;
	end Some_Pkg;

package Some_Pkg
  thread Some_Thread.imp
  end Some_Thread;
end Some_Pkg;
	package body Custom_Pkg
	is
	  st : Some_Thread;
	
	  task body Some_Thread
	  is
	  begin
	  −− implementation
	  end Some_Thread;
	end Custom_Pkg;

Table 4.8: *AADL subprograms to SPARK Ada subprograms(procedures/functions) mapping.*

AADL/BLESS	SPARK Ada

subprogram sp
features
  e : **in parameter** T;
  s : **out parameter** T;
end sp;
	procedure sp(e : **in** T; s : **out** T) **is**
	begin
	  −−# implementation
	end sp;

(e.g. P.N in listings 4.3 and 4.4. However, there is a difference between nested packages and child packages. In nested package declarations become visible as they are introduced, in textual order. For example, in listing 4.3 spec N cannot refer to M in any way. In case of child packages, with certain exceptions, all the functionality of the parent is available to a child and parent can access all its child packages. More precisely: all public and private

declarations of the parent package are visible to all child packages. Private child package can be accessed only from parent's body.

```
package P is
   D: Integer;

   -- a nested package:
   package N is
      X: Integer;
   private
      Foo: Integer;
   end N;

   E: Integer;
private
   -- nested package in private section:
   package M is
      Y: Integer;
   private
      Bar: Integer;
   end M;

end P;
```

Listing 4.3: *Nested packages in SPARK Ada*

```
package P is
   D: Integer;
   E: Integer;
end P;

-- a child package:
package P.N is
    X: Integer;
  private
    Foo: Integer;
end P.N;

-- a child private package:
private package M is
  Y: Integer;
private
  Bar: Integer;
end M;
```

Listing 4.4: *Child packages in SPARK Ada*

There was an idea to create child package to encapsulate one feature group in it. However, SPARK Ada does not allow to access child packages private part from parent. That will require to expose feature group internal variable, which will have to be acceessible globaly. It is definitely not good solution. Thus, feature group is translated with prefix Feature_Group_Name_*.

4.1.6 AADL package to SPARK Ada package mapping

On listing 4.5, there is shown sample AADL package with system. It contains all types of ports and feature group.

```
package Some_Pkg
public
with Base_Types;

feature group Some_Features
features
  Some_Out_Port: out data port Base_Types::Integer;
  Some_In_Port: in data port Base_Types::Integer;
end Some_Features;

system Some_System
features
  Some_Feature_Group : feature group Some_Features;

  In_Data_Port : in data port Base_Types::Integer;
  Out_Data_Port : out data port Base_Types::Integer;
  In_Event_Port : in event port;
  Out_Event_Port : out event port;
  In_Event_Data_Port : in event data port Base_Types::Integer;
  Out_Event_Data_Port : out event data port Base_Types::Integer;
end Some_System;

end Some_Pkg;
```

Listing 4.5: *Sample AADL package with system*

Based on ports mapping, presented in section 4.1.2, translation to SPARK Ada package is shown in listing 4.6.

```
package Some_Pkg
```

```
--# own Some_Features_Some_Out_Port : Integer;
--#     Some_Features_Some_In_Port : Integer;
--#     In_Data_Port : Integer;
--#     Out_Data_Port : Integer;
--#     In_Event_Data_Port : Integer;
--#     Out_Event_Data_Port : Integer;
--# initializes Some_Features_Some_Out_Port,
--#             Some_Features_Some_In_Port,
--#             In_Data_Port,
--#             Out_Data_Port,
--#             In_Event_Data_Port,
--#             Out_Event_Data_Port;
is

   function Some_Features_Get_Some_Out_Port return Integer;
   --# global in Some_Features_Some_Out_Port;

   procedure Some_Features_Receive_Some_In_Port;
   --# global out Some_Features_Some_In_Port;

   procedure Receive_In_Data_Port;
   --# global out In_Data_Port;

   function Get_Out_Data_Port return Integer;
   --# global in Out_Data_Port;

   procedure Put_In_Event_Port;

   procedure Send_Out_Event_Port;

   procedure Put_In_Event_Data_Port(In_Event_Data_Port_In : Integer);
   --# global out In_Event_Data_Port;
   --# derives In_Event_Data_Port from In_Event_Data_Port_In;

   procedure Send_Out_Event_Data_Port;
   --# global in Out_Event_Data_Port;

end Some_Pkg;

package body Some_Pkg
is
   Some_Features_Some_Out_Port : Integer := 0;
   Some_Features_Some_In_Port : Integer := 0;
   In_Data_Port : Integer := 0;
   Out_Data_Port : Integer := 0;
   In_Event_Data_Port : Integer := 0;
   Out_Event_Data_Port : Integer := 0;

   function Some_Features_Get_Some_Out_Port return Integer
   is
   begin
```

```
      return Some_Features_Some_Out_Port;
   end Some_Features_Get_Some_Out_Port;

   procedure Some_Features_Receive_Some_In_Port
   is
   begin
      -- implementation
   end Some_Features_Receive_Some_In_Port;

   procedure Receive_In_Data_Port
   is
   begin
      -- implementation
   end Receive_In_Data_Port;

   function Get_Out_Data_Port return Integer
   is
   begin
      return Out_Data_Port;
   end Get_Out_Data_Port;

   procedure Put_In_Event_Port
   is
   begin
      -- implementation
   end Put_In_Event_Port;

   procedure Send_Out_Event_Port
   is
   begin
      -- implementation
   end Send_Out_Event_Port;

   procedure Put_In_Event_Data_Port(In_Event_Data_Port_In : Integer)
   is
   begin
      In_Event_Data_Port := In_Event_Data_Port_In;
   end Put_In_Event_Data_Port;

   procedure Send_Out_Event_Data_Port
   is
   begin
      -- implementation
   end Send_Out_Event_Data_Port;

end Some_Pkg;
```

Listing 4.6: *Translation of sample AADL package from listing 4.5*

4.1.7 BLESS mapping

Table 4.9: BLESS to SPARK contracts mapping.

AADL/BLESS	SPARK Ada
BLESS::Assertion=>"<<COND1()>>"	--# assert COND1();

thread Some_Thread
features
  Some_Port : out event port
  {BLESS:Assertion => "<<(Var1 < Var2 **and**
  COND2())>>";};
end Some_Thread;
	task body Some_Thread
	is
	begin
	  loop
	  --# assert (Var1 < Var2 and COND2());
	  end loop;
	end Some_Thread;

thread implementation Some_Thread.imp
annex BLESS
{**
  invariant <<(Some_Var < Other_Var)>>
**};
end Some_Thread.imp;
	task body Some_Thread
	is
	begin
	  loop
	  --# assert (Some_Var < Other_Var);
	  end loop;
	end Some_Thread;

thread implementation Some_Thread.imp
annex BLESS
{**
  assert
  <<State1 : :COND1() **or** COND2()>>
  <<Var : :=
  (State1()) -> 0,
  (State2()) -> -1,
  (State3()) -> 9
  >>
**};
end Some_Thread.imp;
	task body Some_Thread
	is
	begin
	  loop
	  --# assert COND1() or COND2()
	  --# -> State1();
	  --# assert (Var = 0) -> State1() and
	  --# (Var = -1) -> State2() and
	  --# (Var = 9) -> State3();
	  end loop;
	end Some_Thread;

Continued on next page

Table 4.9 – continued from previous page

AADL/BLESS	SPARK Ada

subprogram Some_Subprogram
features
  param : out parameter Base_Types::Integer;
annex subBless
{**
  pre <<(param > 0)>>
  post <<(param = 0)>>
**};
end Some_Subprogram;
	procedure Some_Subprogram(Param : **in out** Integer);
	  −−# pre Param > 0;
	  −−# post Param = 0;

<<Pre()>>Action()<<Post()>>	**procedure** Action; −−# pre Pre; −−# post Post;
<<Pre()>>Action()<<Post()>>	**procedure** Action; −−# pre Pre; −−# post Post;

Generated (translated) code will not be complete. It will still require Developer's effort to implement missing parts. E.g. when assertion is not defined, it is developer responsibility to implement it.

4.2 "DeusEx" translator

The ultimate goal is to perform, translation described in 4.1 automatically. "DeusEx" translator will enable to perform translation of entire model and parts of the model. Initially, following functions will be supported:

- types translation

- threads to tasks translation
- subprogram to procedure/function translation
- single package translation

Translator will be created in Scala programming language.

Chapter 5

PCA Pump Prototype Implementation

Currently SPARK 2014 does not support tasking [AL14]. For SPARK 2005, GNAT compiler provides Ravenscar Profile [Tea12]. It provides a subset of the tasking facilities of Ada95 and Ada 2005 suitable for the construction of high-integrity concurrent programs.

In real-world applications, the embedded critical components are written in SPARK while the non-critical components are written in Ada. Components written in Ada should be hidden for SPARK Examiner with `--# hide` annotation.

The biggest challenge during PCA Pump development was the SPARK limitations. There are many common libraries, which cannot be verified by SPARK tools. Thus it required to isolate some functionalities or implement them in different way. An example might be reading and writing numbers to standard input.

5.1 Concurrency in SPARK Ada

Based on AADL models, PCA Pump has to be multitasking device. Thus, concurrency features are needed. In SPARK 2005, concurrency is enable with Ravenscar profile [Tea12]. For now, concurrency is not allowed in SPARK 2014.

5.2 Implementation based on Requirements Document

The first step, was to check if implementation of PCA Pump specified in Requirements Document is possible. To do that, simple version of PCA Pump based on Requirements Document was created. Only two AADL threads are implemented: Rate_Controler and Max_Drug_Per_Hour_Watcher.

5.3 Code generation from AADL models

Skeleton code generated from AADL models. Then implemented.

Show generated code.

5.4 Implementation for generated code

Overview of issues solved: * Bolus options: FBasal + FPatient or FPatient => implemented: FBasal + FPatient (consistent in doc) 5 modes: * Stopped: F=0 * KVO: F=0.1 * Basal: F=Fbasal * Patient: F = Fbasal + Fbolus (for vtbi/Fbolus) * Clinician: F = Fbalsal + Fbolus (for specified time)

Most common Examiner[Tea11b] erroes/warnings: *** Warning :302: This expression may be *** Semantic Error :725: Protected function or variable XXX may only appear directly in an assignment or return statement.

Discuss implementation of basal infusion: 0.1 ml pulses timed according to the desired rate. (based on CADD-Prizm page 14). Easier bolus monitoring/calculations. Possibility to separate pulse from engine logic. Just array with time stamps(?) or array with size = (60 * 60 /min_possible_time_between_activations) and set 1 if activation occured. In every second, update array: array[i]=array[i+1]. Array is protected object, so bolus thread cannot access it in the same time, when update thread. Another option: constant speed of

engine and speed-up on boluses. Harder bolus monitoring/calculations?

Internal calculations are in micro liters 1 micro liters (μl) = 0.001 ml thus 1 ml = 1000μl.

Chapter 6

Verification

Verification - what is that verification vs validation

SPARK tools FDL is the modelling language of the SPARK proof tools.

6.1 SPARK Examiner

The main SPARK verification tool is Examiner. It supports several levels of analysis:

- checking of SPARK language syntactic and static semantic rules
- data flow analysis
- data and information flow analysis
- formal program verification via generation of verification conditions
- proof of absence of run-time errors
- dead path analysis

[1] http://docs.adacore.com/sparkdocsdocs/Examiner_UM.htm

There is also an option to make the Examiner perform syntax checks only. Using this option on a source file does not require access to any other units on which the file depends, so files can be syntax checked on an individual basis. This allows any syntax errors to be corrected before the file is included in a complex examination. This option must only be used as a pre-processor: the absence of syntax errors does NOT indicate that the source text is a legal SPARK program. [Tea11b] (THIS PART IS COPY AND PASTE FROM Examiner doc - is it ok?)

Put here some examples: method without contract, examine, add specification, pass Examiner.

During implementation, code was regularly checked using SPARK Examiner.

What is very important, Examiner can perform data and information analysis of Ravenscar programs in exactly the same manner as for sequential programs [Tea12]. Unfortunately it does not allow protected objects in proof annotations (pre- and post-conditions).

When some parts of the system are written in full Ada (with non-valid SPARK constructs), then Examiner returns error. Ada parts can be excluded from Examiner analysis using `--# hide` annotation. The, only warning `10 – The body of subprogram Main is hidden – hidden text is ignored by the Examiner.` is returned by Examiner.

Examiner use SPARK index file to locate files necessary for verification. [Bar13]

Examiner can be used with `spark` command and appropriate flags described in Examiner Manual [Tea11b].

To use Examiner in GNAT Programming Studio:

- Run SPARK Make (right click on project / SPARK / SPARK Make)
- Set SPARK index file (to spark.idx generated by SPARKMake) [add photo from 721 paper]
- (optionally) set configuration file (Standard.ads)

- Choose appropriate version of SPARK (95 or 2005)
- Choose mode: Sequential (for single tasking programs) or Ravenscar (for multitasking programs)

To generate verification conditions (VCs), the –vcg switch has to be used. It can be set in GNAT Programming Studio (Project / Edit project properties / Switches / Examiner / Generate VCs). In addition to verification conditions, Examiner can check dead path conjectures. It checks, whether all of the program is useful. To generate dead path conjectures, the –dpc switch has to be used. It can be also set in GNAT Programming Studio (Project / Edit project properties / Switches / Examiner / Generate DPCs).

6.1.1 Flow analysis

There are two types of flow analysis:

- Data flow analysis:
  - Checks input/output behavior of parameters and variables.
  - Checks initialization of variables.
  - Checks that changed and imported variables are used later (possibly as output variables).
- Information flow analysis - verifies interdependencies between variables.

In data flow analysis, Examiner checks if input parameters are not modified, but used at least once (in at least one branch of program). In the same factor, output parameters cannot be read (before initialization) and has to be initialized (in all branches of program). Input/output parameters has to be both read and write (changed). In similar way, Examiner verify the global variables (specified in annotations). Functions can use only input parameters and can only read global variables. Therefore functions do not have side effects.

Global variables defined in package body (thus private) has to be declared by `--# own` annotation in package specification. If variable is also initialized, `--# initializes` annotation has to be used. In Ada, to use package in another package, **with** clause has to be used. In SPARK Ada, additionally `--# inherits` annotation has to be specified.

In information flow analysis, dependencies between variables are analyzed. These dependencies are specified by `--# derives` annotation.

6.1.2 Verification conditions

To generate verification conditions, two kinds of annotations are relevant for Examiner:

- pre-conditions: `--# pre`
- post-conditions: `--# post`

Notion of pre- and post-conditions represents Hoare logic. More precisely, Hoare triple:

$$\{P\}C\{Q\} \tag{6.1}$$

P and Q are assertions. C is a command (action) performed between them. P is pre-condition and Q is post-condition.

Additionally, assertions (`--# assert`) and checks (`--# check`) can be specified in procedure body. Then additional verification conditions are generated.

Functions does not have side effects (as stated in 6.1.1), thus only pre-condition can be applied. However, there is annotation `--# return`, which specify function return value.

Verification conditions are generated depended on number of paths in subprogram. Analysis are perform backwards, in other words: we start from post-conditions and consider what must holds before. Flow analysis is well described in chapter 11 of Barnes' book [Bar13].

6.2 SPARK Simplifier

Simplifier can discharge (prove correctness) of verification conditions (VCs) generated by Examiner, but not proved by Examiner. [Tea11c]

6.3 ZombieScope

ZombieScope is a SPARK tool, that analyses SPARK code to find dead paths, i.e. paths through the code that can never be executed.

6.4 Victor

Victor is a tool to translate SPARK verification conditions (VCs), as generated by the Examiner, into SMT-LIB (file format used to communicate with SMT solvers). [Tea] SMT (Satisfiability Modulo Theories) solver is a tool... experimental feature Integrated with SPARKSimp (by -victor flag) and POGS.

6.5 Proof Checker

Only mention. It is hardcore.

6.6 SPARKSimp Utility

SPARKSimp is a simple "make" style tool for the SPARK analysis tools. Currently, it supports the Simplifier, ZombieScope and ViCToR. It applies the Simplifier (and ViCToR, if requested, please see the Victor_Wrapper user manual [Tea] for more information) to all .vcg files and ZombieScope to all .dpc files it finds in a directory tree. [Tea10]

6.7 Proof Obligation Summarizer (POGS)

The Proof ObliGation Summarizer tool (POGS) reads and understands the structure of the verification condition files. It reports the status of proofs and dead path analyses in a human-readable form. [Tea11a]

6.8 Example verification

An overview of verification contracts and annotations can be found in chapter 12 of Barnes' book [Bar13]. On Odometer or RateController of PCA Pump?

6.9 Verification of PCA Pump

Examine(F8) -> Simplifier –> POGS.

4 warnings: Warning 402 - Default assertion planted to cut loop.

solution: −−# assert I > 1 −> TheStoredData(I−1) = TheStoredData(I); −−# assert I > 1 −> Result >= TheStoredData(I−1); −−# assert true; // add BLESS assertions? then resolve warning 402? −−# assert true; // add BLESS assertions? then resolve warning 402?

Verification of main.adb

db:4:10: Warning 391 − **If** the identifier Text_IO represents a **package** which contains a **task or** an interrupt handler **then** the partition−level analysis performed by the Examiner will be incomplete. Such packages must be inherited as well as withed.

db:6:10: Warning 391 − **If** the identifier Float_Text_IO represents a **package** which contains a **task or** an interrupt handler **then** the partition−level analysis performed by the Examiner will be incomplete . Such packages must be inherited as well as withed.

db:102:5: Warning 10 − The **body of** subprogram Main **is** hidden − hidden text **is** ignored by the Examiner.

db:14:49: Flow Error 602 − The undefined initial value **of** Pca_Pump.Operate may be used **in** the derivation **of** Pca_Pump.State.

db:14:49: Flow Error 602 − The undefined initial value **of** Pca_Pump.Fluid_Pulses may be used **in** the derivation **of** Pca_Pump.State.

db:14:49: Flow Error 602 − The undefined initial value **of** Pca_Pump.Prescription may be used **in** the derivation **of** Pca_Pump.State.

db:14:49: Flow Error 602 − The undefined initial value **of** Pca_Pump.Clinician_Bolus_Paused may be used **in** the derivation **of** Pca_Pump.State.

db:14:49: Flow Error 602 − The undefined initial value **of** Pca_Pump.Clinician_Bolus_Duration may be used **in** the derivation **of** Pca_Pump.State.

db:14:49: Flow Error 37 − The updating **of** variable Pca_Pump.Operate by a **task or** interrupt handler has been omitted from the partition annotation.

db:14:49: Flow Error 37 − The updating **of** variable Pca_Pump.Fluid_Pulses by a **task or** interrupt handler has been omitted from the partition annotation.

db:14:49: Flow Error 37 − The updating **of** variable Pca_Pump.Clinician_Bolus_Paused by a **task or** interrupt handler has been omitted from the partition annotation.

db:14:49: Flow Error 36 − The referencing **of** variable Pca_Pump.Operate by a **task or** interrupt handler has been omitted from the partition annotation.

db:14:49: Flow Error 36 − The referencing **of** variable Pca_Pump.Fluid_Pulses by a **task or** interrupt handler has been omitted from the partition annotation.

db:14:49: Flow Error 36 − The referencing **of** variable Pca_Pump.Prescription by a **task or** interrupt handler has been omitted from the partition annotation.

db:14:49: Flow Error 36 − The referencing **of** variable Pca_Pump.Clinician_Bolus_Paused by a **task or** interrupt handler has been omitted from the partition annotation.

db:14:49: Flow Error 36 − The referencing **of** variable Pca_Pump.Clinician_Bolus_Duration by a **task or** interrupt handler has been omitted from the partition annotation.

es/External/VMS/shared/aadl−medical/pca−pump−beagleboard/pca_ravenscar/main.adb:1:1: Warning − VC generation requested but no bodies presented. No VCs generated.

6.10 AUnit tests

- test incrementing array - test moving array (Pulse does not change dosed amount) - test prescription setters and getters - test state machine * change to bolus mode * change to KVO rate * etc.

6.11 gnatPROVE?

There is a new tool set "gnatPROVE" for SPARK 2014. It was not used because PCA Pump was developed in SPARK 2005. I CAN TRANSLATE SOME SINGLE FUNCTIONS AND USE GNAT PROVE TO VERIFY?

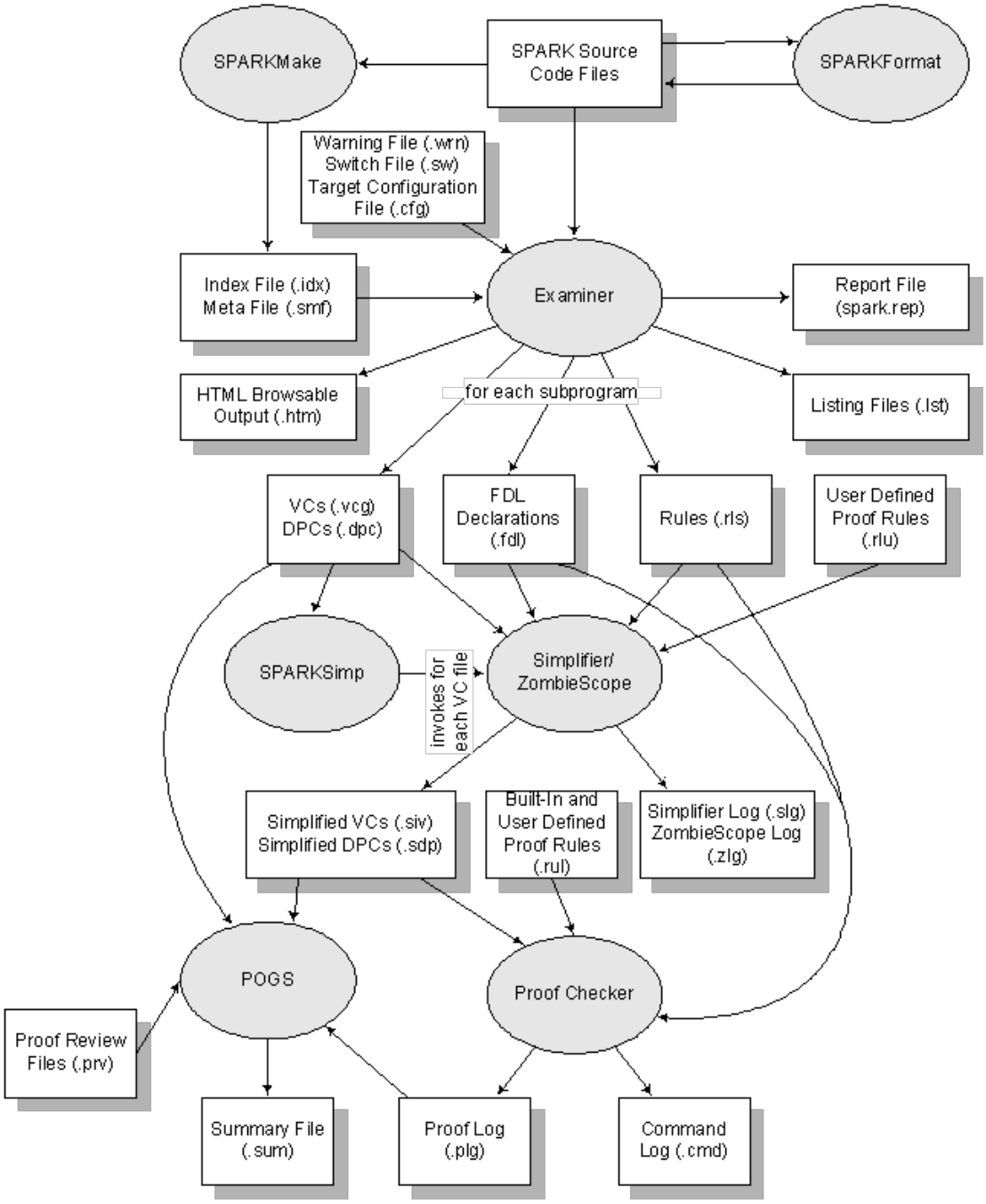

Figure 6.1: *Relationship of the Examiner and Proof Tools*[1].

Chapter 7

Summary

What I have done.

The work is done for SPARK 2005. SPARK 2014 (especially taking) and its tools (such as gnatPROVE) were not ready at the time, when this thesis was written.

Issues:

- not many online resources - no access to industry code - everything (AADL, SPARK2014, BLESS, tools) is under development - hard to create running application - need to rely on some resources, which are not nessesarly up to date

Chapter 8

Future work

What has to be done now.

translation of BLESS state machine

The semantics of BLESS contain notions of time that make translation to SPARK difficult.

translations for SPARK 2014

try to apply generics on types translation

try to apply child packages for feature groups

Translator: * it should ignore all not defined properties in data types translations

Bibliography

[AB04] Tullio Vardanega Alan Burns, Brian Dobbing. Guide for the use of the ada ravenscar profile in high integrity systems. *ACM SIGAda Ada Letters*, 24(2):1–74, Juin 2004.

[Ada14] AdaCore. Aunit cookbook. URL: http://docs.adacore.com/aunit-docs/aunit.html, Mars 2014.

[AL14] AdaCore and Altran UK Ltd. Spark 2014 reference manual. URL: http://docs.adacore.com/spark2014-docs/html/lrm, 2011-2014.

[AW01] Neil Audsley and Andy Wellings. Issues with using ravenscar and the ada distributed systems annex for high-integrity systems. In *IRTAW '00 Proceedings of the 10th international workshop on Real-time Ada workshop*, pages 33 – 39. ACM New York, NY, USA, 2001.

[Bar13] John Barnes. *SPARK - The Proven Approach to High Integrity Software*. Altran, 2013.

[BHR+11] Jason Belt, John Hatcliff, Robby, Patrice Chalin, David Hardin, and Xianghua Deng. Bakar kiasan: Flexibe contract checking for critical systems using symbolic execution. In *NASA Formal Methods*, pages 58–72. Springer Berlin Heidelberg, 2011.

[CB09] Mohamed Yassin Chkouri and Marius Bozga. Prototypying of distributed embedded systems using aadl. In *ACESMB 2009, Second International Workshop on*

Model Based Architecting and Construction of Embedded Systems, pages 65–79. Springer Berlin Heidelberg, 2009.

[Cha00] Roderick Chapman. Industrial experience with spark. *ACM SIGAda Ada Letters - special issue on presentations from SIGAda 2000*, XX(4):64–68, Décembre 2000.

[DEL+14] Claire Dross, Pavlos Efstathopoulos, David Lesens, David Mentre, and Yannick Moy. Rail, space security: Three case studies for spark 2014. In *ERTS 2014: Embedded Real Time Software and Systems*, 2014.

[Fal14] Ed Falis. Aunit tutorials. URL: http://libre.adacore.com/tools/aunit, Mars 2014.

[FG13] Peter H. Feiler and David P. Gluch. *Model-Based Engineering with AADL*. Addison-Wesley, 2013.

[Hor09] Bartłomiej Horn. Ada'05 compiler for arm based systems. thesis, Technical University of Lodz, Poland, 2009.

[Hug13] Jérôme Hugues. About ocarina. URL: http://www.openaadl.org/ocarina.html, 2013.

[HZPK08] Jérôme Hugues, Bechir Zalila, Laurent Pautet, and Fabrice Kordon. From the prototype to the final embedded system using the ocarina aadl tool suite. *ACM Transactions on Embedded Computing Systems*, 7(4):237–250, Juilliet 2008.

[IEC+06] Andrew Ireland, Bill J. Ellis, Andrew Cook, Roderick Chapman, and Janet Barnes. An integrated approach to high integrity software verification. *Journal of Automated Reasoning*, 36(4):379–410, Avril 2006.

[Lar14] Brian R. Larson. Integrated clinical environment patient-controlled analgesia infusion pump system requirements draft 0.10.1, Février 2014.

[LCH13] Brian R. Larson, Patrice Chalin, and John Hatcliff. Bless: Formal specification and verification of behaviors for embedded systems with software. In *NASA Formal Methods*, pages 276–290. Springer Berlin Heidelberg, 2013.

[LHC13] Brian R. Larson, John Hatcliff, and Patrice Chalin. Open source patient-controlled analgesic pump requirements documentation. In *Software Engineering in Health Care (SEHC), 2013 5th International Workshop*, pages 28–34. Institute of Electrical and Electronics Engineers (IEEE), 2013.

[LZPH09] Gilles Lasnier, Bechir Zalila, Laurent Pautet, and Jérôme Hugues. Ocarina : An environment for aadl models analysis and automatic code generation for high integrity applications. In *Reliable Software Technologies – Ada-Europe 2009*, pages 237–250. Springer Berlin Heidelberg, 2009.

[Med10] Smiths Medical. Cadd-prizm ambulatory infusion pump model 6100 and model 6101 - technical manual. URL: http://www.smiths-medical.com/upload/products/pdf/cadd_prizm_vip_system/in19824.pdf, Novembre 2010.

[SCD14] SAE AS-2C Architecture Description Language Subcommittee, Embedded Computing Systems Committee, and Aerospace Avionics Systems Division. Aerospace standard - architecture analysis & design language (aadl) v2 programming language annex document draft 0.9, Avril 2014.

[Tea] SPARK Team. Victor wrapper user manual.

[Tea10] SPARK Team. Sparksimp utility user manual. URL: http://docs.adacore.com/sparkdocs-docs/SPARKSimp_UM.htm, Novembre 2010.

[Tea11a] SPARK Team. Pogs user manual. URL: http://docs.adacore.com/sparkdocs-docs/Pogs_UM.htm, Septembre 2011.

[Tea11b] SPARK Team. Spark examiner user manual. URL: http://docs.adacore.com/sparkdocs-docs/Examiner_UM.htm, Décembre 2011.

[Tea11c] SPARK Team. Spark simplifier user manual. URL: http://docs.adacore.com/sparkdocs-docs/Simplifier_UM.htm, Juin 2011.

[Tea12] SPARK Team. The spark ravenscar profile. URL: http://docs.adacore.com/sparkdocs-docs/Examiner_Ravenscar.htm, 2012.

[Thi11] Hariharan Thiagarajan. Dependence analysis for inferring information flow properties in spark ada programs. thesis, Kansas State University, 2011.

Appendix A

PCA Pump Prototype - simple, working example

Content of this appendix.

Appendix B

PCA Pump Prototype - translated from AADL/BLESS