MODEL DRIVEN DEVELOPMENT AND VERIFICATION IN AADL/BLESS AND SPARK ADA FOR A PCA PUMP

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

The future of Medical Devices is their interoperability. Nowadays, medical devices works rather independently. It cause accidents, which could have been avoided if different devices communicated. In order to communicate, devices have to use the same interfaces and protocols. Dr. Julian Goldman developed idea of "Integrated Clinical Environment" (ICE). It is series of standards, which describes medical device interoperability. SAnToS lab created Medical Device Coordination Framework (MDCF), which is prototype implementation of ICE idea.

AADL (Architecture Analysis & Design Language) is modeling language for representing hardware and software. It is used for real-time, safety critical and embedded systems. AADL allows for the description of both software and hardware parts of a system. It is used to describe architecture, but AADL allows to add behavioral extensions through annex languages. 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.

Ada is one of the most popular (along with C/C++) programming language targeted at embedded and real-time systems. SPARK Ada is subset of Ada, designed for the development of safety and security critical systems. It contains subset, which allows to reason about and prove correctness of program and its entities.

Nowadays, there is a 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 translator. Ultimately set of standardized AADL/BLESS models for medical devices (ICE compliant) would be

created. From these models SPARK Ada code base would be generated. It would be further extended by developers.

This thesis propose mapping from AADL/BLESS to SPARK Ada. As an example of Medical Device, PCA (Patient Controlled Analgesia) pump is used. The foundation for this work is "Integrated Clinical Environment Patient-Controlled Analgesia Infusion Pump System Requirements" document [Lar14] and AADL Models with BLESS annexes created by Brian Larson. In addition to proposed mapping, PCA pump prototype was created. As a platform for prototyping, BeagleBoard-xM device was used. Some components of PCA pump prototype are verified by SPARK tools and Bakar Kiasan.

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

Software is present in all aspects of our life. From the simple program in alarm clock to iPad, through cars, refrigerators and computers. Furthermore, our lives are getting more and more depended on Software. Usually when we think about Software, we think about Applications for PC or Smart Phone. E.g. Calculator, Word processor or Stock Market application. In this case, rapid development and smooth operation is a key. However, there is also another, very important class of Software: Safety Critical Systems. It comprises software for Airplanes, Medical Devices, Satellites or Rockets.

Software Engineering for Real-Time and Safety-Critical Systems is very different than creating Business applications. In both types of software we want to ensure correctness and security. However, in each of them, to different extent. In case of mentioned Word processor, software assurance is not critical. When it crashes, it can be restarted. In worst case scenario, some part of work might be lost. Airplane software crash may put human life in danger or even cause the death. Thus for Safety-Critical systems, the security and correctness are crucial. Behind these reasons, different Software Design methodology and different properties of programming language and its tools are needed.

The most important part of Safety-Critical Systems Design is hazard analysis. How to avoid unintentional states and how to recover from them. Hazard can cause incident or

accident. Former is an event, which not cause a loss (but undesired), and could lead to accident. Latter cause the loss (and it is also undesired). Hazard analysis can be done manually by human or automatically by software tools. AADL, BLESS and SPARK Ada contains variety of them.

1.1 Motivation

There are many accidents where Medical Devices are involved. Very often, the reason is the lack of communication between them. Drug dosed by PCA Pump may affect patient's level of oxygen and carbon dioxide level. Thus adequate monitoring of patient's levels of oxygen and carbon dioxide is required. Moreover, integrated system, which will take adequate action in case of hazard is needed. The solution for such a problem is "Integrated Clinical Environment" (ICE). SAnToS Lab at Kansas State University, in cooperation with University of Pennsylvania are working on Medical Device Coordination Framework (MDCF) [HKL+12], which is prototype implementation of ICE. It is an open source framework for coordinating multiple medical devices to work together.

Devices working under MDCF have to satisfy some requirements. To make Developer's life easier, the requirements will be not only in documentation, but also in AADL/BLESS models. Model Driven Development in this case means that there from base models (in AADL/BLESS) for medical devices development, skeleton code (in SPARK Ada) will be generated. Then developer will extend and customize them according to his needs. In the same fashion like File > 'New Java project' in Eclipse, File > 'New Medical device project' will work in GNAT Programming Studio. AADL/BLESS Model will be specification and requirements. The ultimate goal is to create set of AADL/BLESS models for different medical devices, 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.

PCA pump prototype created in this thesis is as an example of Medical Device, which ultimately will work under MDCF.

1.2 Goals

The initial goals, which most of them is accomplished are as follows:

- identify PCA Pump and Infusion pumps properties and internals required for implementation
- SPARK Ada cross-compilation for ARM-device (BeagleBoard-xM)
- implement PCA Pump based on Brian Larson's Requirement Document [LHC13]
- develop AADL/BLESS to SPARK Ada mapping
- mock PCA Pump AADL/BLESS models in SPARK Ada (based on created mapping and implementation)
- implement not generated part (based on implementation) [NOT ACCOMPLISHED - REMOVE?]
- create AADL/BLESS to SPARK Ada translator [NOT ACCOMPLISHED - REMOVE?]
- Use SPARK tool set for software verification:
  - SPARK Examiner
  - SPARK Simplifier
  - Proof Obligation Summarizer (POGS)
  - Bakar Kiasan
  - GNATprove

1.3 Contribution

This thesis demonstrates how AADL/BLESS models can be mapped to SPARK Ada. Additionally it presents current possibilities and limitations of SPARK Ada language, Ravenscar profile and SPARK verification tools. The main contributions of this thesis are as follows:

- Review of PCA Pump Requirements document [LHC13]
- Cross-compilation and testing of SPARK Ada 2005 and 2014 code on BeagleBoard-xM platform
- Implementation of PCA Pump based on Requirements document [LHC13] and AADL/BLESS models, which validates them
- Creation of PCA pump prototype
- Analysis of different PCA pump implementation possibilities
- AADL/BLESS to SPARK Ada translation schemes
- Practical demonstration of SPARK 2005 verification tools: its capabilities and limitations

1.4 Organization

The thesis is organized in 8 [fix this: how to count all chapters?] chapters:

- Chapter 1 is the problem description and summary of contribution which has been made.
- Chapter 2 is Background that gives details about ICE, MDCF, Model Driven Development, AADL/BLESS, SPARK Ada and available tools for such environment.
- Chapter 3 describes Patient-Controlled Analgesia (PCA) pump.

- Chapter 4 presents mappings from AADL/BLESS to SPARK Ada.
- 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)
- **FDA** - Food and Drug Administration
- **GPS** - GNAT Programming Studio
- **GCC** - GNU Compiler Collection
- **GUI** - Graphical user interface
- **VC** - Verification Condition
- **DPC** - Dead Path Conjecture
- **POGS** - Proof Obligation Summarizer

- **VTBI** - Volume to be infused
- **KVO** - Keep Vein Open

Chapter 2

Background

This chapter is a brief introduction of all technologies and tools used in this thesis. There are: AADL modeling language, BLESS (AADL annex language), SPARK Ada programming language and its verification tools. There is also an 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

Idea of "Integrated Clinical Environment" (ICE) was initiated by Dr. Julian Goldman from Center for Integration of Medicine & Innovative Technology. The main idea is to create environment of medical devices network. It will allow clinician and software system to make decisions based not only on output from one device, but from all of them together. ICE purpose is to solve current issues with medical devices, which usually operate independently. It requires more human attention and control through checking output of every device manually and then making decision. ICE will make it easier, e.g. by introducing alarms, which can not only indicate problem but also interact with other devices and make decision

automatically. E.g. when PCA Pump infuse some drug to patient's vein and Pulse Oximeter detects low oxygen level, ICE can coordinate PCA pump shutdown.

Moreover, ICE comprises components that may be implemented by different vendors. Such components are medical devices and applications to supervise them. Figure 2.1 presents high overview of ICE system. Medical devices (PCA Pump, Respiratory Rate Monitor and Pulse Oximeter) are connected to the system. All of them are monitored and controlled. There is communication between devices and ICE, in order to exchange data between them and Electronic Medical Record (EMR) Database. Informations in EMR comprises drug library, patient's medical records, monitoring logs etc. ICE can make decisions (such as PCA Pump shutdown) based on that informations.

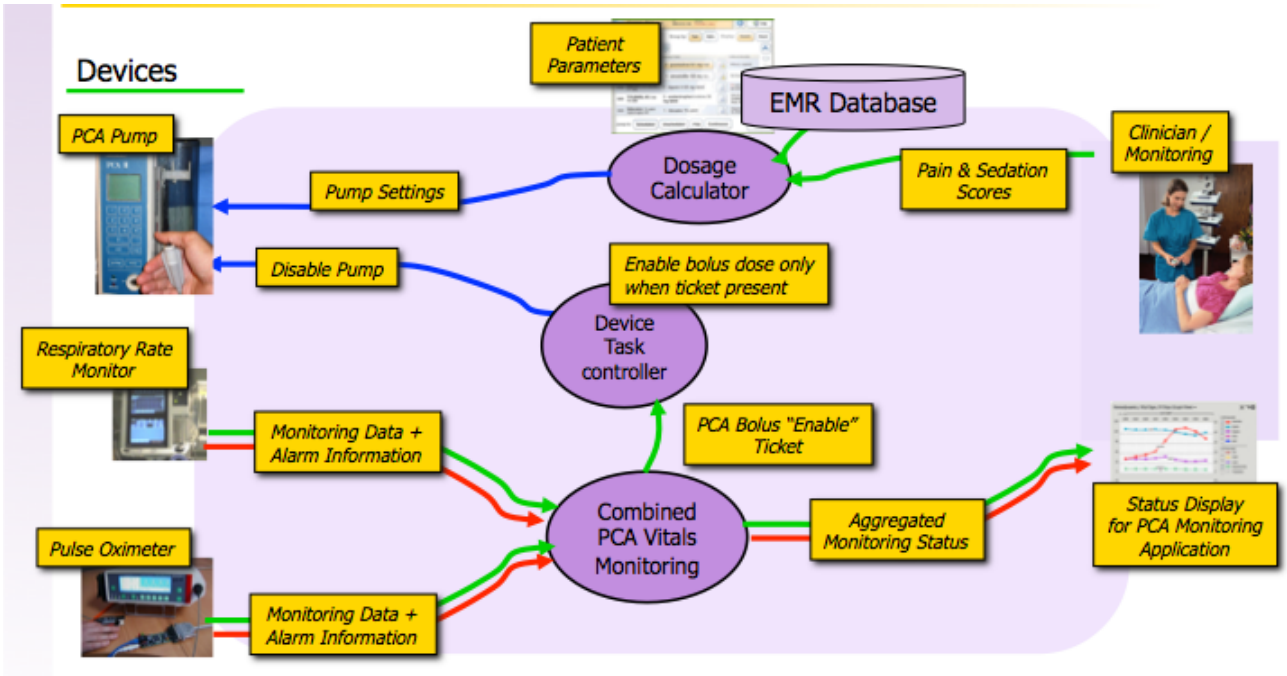

Figure 2.1: *ICE Closed Loop Control*

[ADD MORE INFORMATION?]

2.2 Medical Device Coordination Framework

Medical Device Coordination Framework (MDCF), jointly developed by SAnToS lab (Kansas State University) and University of Pennsylvania is prototype implementation of ICE. It is an open, experimental platform to bring together academic researchers, industry vendors, and government regulators. Project is response to request from Food and Drug Administration (FDA) to build a prototype of ICE. Medical Devices, which are ICE compliant can be connected to MDCF. The framework enables Medical Devices interoperability. MDCF is designed to illustrate by example the issues related to functional concepts, safety, security, verification and certification.

The goals of MDCF project comprises:

- Open source infrastructure
- Meet performance requirements of realistic clinical scenarios
- Provide middleware with reliability, real-time, security
- Provide an effective app programming model and development environment with integrated verification/validation support and construction of regulatory artifacts
- Support evaluation of device interfacing concepts
- Illustrate how to support real and mock devices
- Illustrate envisioned regulatory oversight and 3rd party certification

In this thesis, part of penultimate point will be illustrated. For now, MDCF use only mock devices, which are Java desktop applications. PCA Pump Prototype aim to be first real-device.

MDCF uses publish-subscribe architecture for communication between components: apps and devices. Figure 2.2 presents MDCF structure. Devices, like PCA pump, are clients.

MDCF Server is integration layer which comprises Core and applications working in top of it. [HLW12].

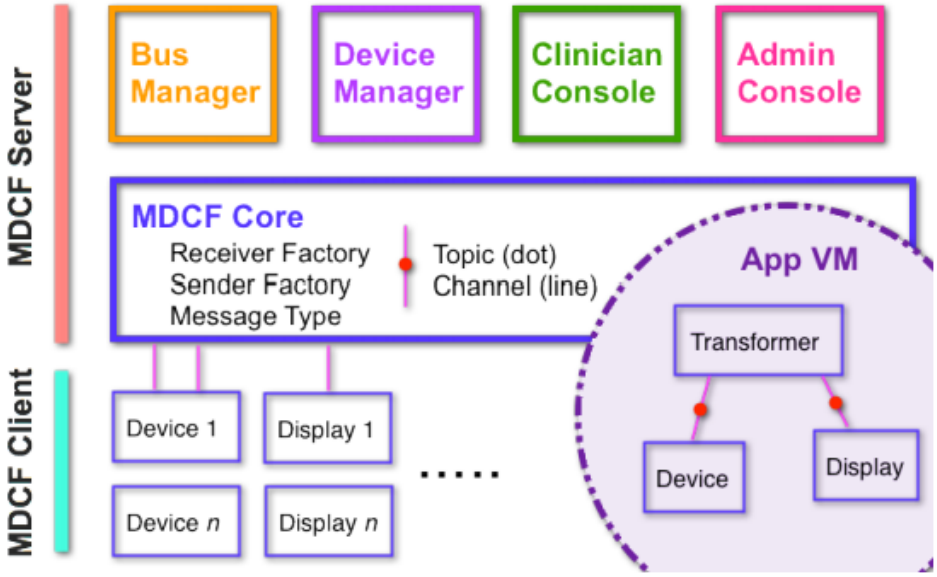

Figure 2.2: *MDCF architecture and example app virtual machine (lower right)*

[ADD MORE INFORMATION?]

2.3 AADL

AADL stands for Architecture Analysis & Design Language. It is used to model embedded and real-time systems. AADL allows for the description of both software and hardware parts of a system. It can be used not only for design phase of software development process, but also for analysis, verification or code generation.

AADL has its roots in DARPA[1] funded research. The first version (1.0) was approved in 2004 under technical leadership of Peter Feiler[2]. AADL is develop by SAE AADL committee[3]. AADL version 2.0 was published in January 2009. The most recent version (2.1)

[1] http://www.darpa.mil
[2] http://wiki.sei.cmu.edu/aadl/index.php/The_Story_of_AADL/
[3] https://wiki.sei.cmu.edu/aadl/index.php/Main_Page

was published in September 2012[4].

AADL is a language for Model-Based Engineering [FG13]. It can be represented in textual and graphical form. There are tools, like OSATE (see section 2.3.1), which transforms textual representation into graphical or XML.

AADL contains entities for modeling software and hardware components. It allows to create interactions and dependencies between them.

AADL Execution Platform Components and Devices:

- Processor / Virtual Processor - Provides thread scheduling and execution services
- Memory - provides storage for data and source code
- Bus / Virtual Bus - provides physical/logical connectivity between execution platform components
- Device - interface to external environment

Application Software Components of AADL (figure 2.3):

- System - hierarchical organization of components
- Process - protected address space
- Thread group - logical organization of threads
- Thread - a schedulable unit of concurrent execution
- Data - potentially sharable data
- Subprogram - callable unit of sequential code

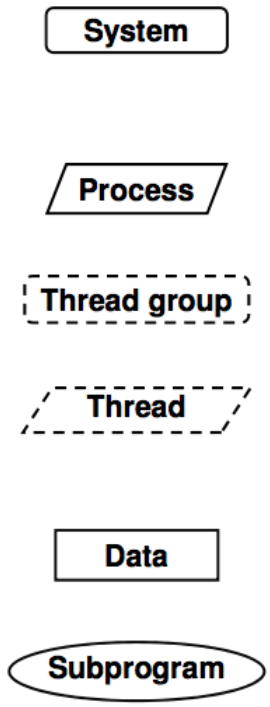

Figure 2.3: *AADL Application Software Components*

[4]https://wiki.sei.cmu.edu/aadl/index.php/Standardization

An example AADL model is shown in graphical representation, in the figure 2.4. Its textual representation is presented in listing 2.1.

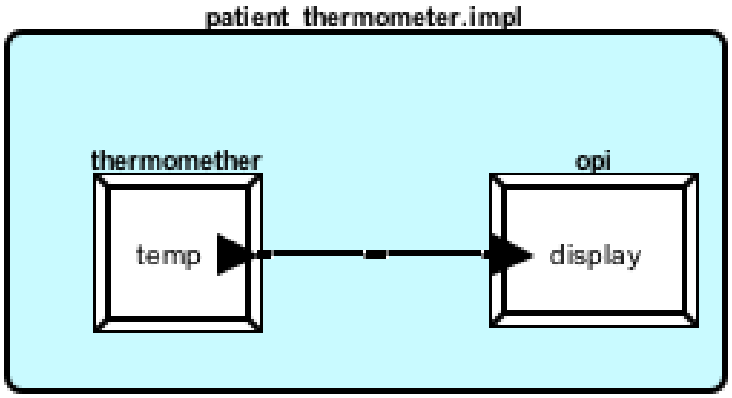

Figure 2.4: *AADL model of simple thermometer*

```
package Thermometer
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: OSATE (see section 2.3.1, STOOD[5], ADELE[6], Cheddar[7], AADLInspector[8] or Ocarina[9].

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

AADL can be extended with the following methods:

- user-defined properties: user can extend the set of applicable properties and add their own to specify their own requirements
- language annexes (the core language is enhanced by annex languages that enrich the architecture description. For now, the following annexes have been defined):
  - Behavior annex: add components behavior with state machines (e.g. BLESS)
  - Error-model annex: specifies fault and propagation concerns
  - ARINC653 annex: defines modelling patterns for modelling avionics system
  - Data-Model annex: describes the modelling of specific data constraint with AADL

More details about AADL can be found in Peter Feiler's book "Model-Based Engineering with AADL" [FG13].

2.3.1 OSATE

Open Source AADL Tool Environment (OSATE) is a set of plug-ins on top of the open-source Eclipse platform. It provides a tool set for front-end processing of AADL models. OSATE

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

is developed mainly by SEI (Software Engineering Institute - Carnegie Mellon University)[10]. Latest available version of OSATE in the time when this work was published is OSATE2[11].

OSATE relies on EMF, UML2 and Xtext[12]. It comprises e.g. AADL project wizard, AADL Navigator and AADL syntax analyzer. OSATE enables conversion of AADL in textual representation into graphical. There are also plug-ins for OSATE, like BLESS[13] or OCARINA[14].

2.4 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 sub-languages:

- 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

BLESS annex subclauses can be added to AADL models transparently to other uses of the system architecture. It includes a verification-condition (VC) generation framework and an accompanying proof tool that enables engineers to prove VCs via proof scripts build from system axioms and rules from a user-customizable rule library. [LCH13]

[10] http://www.aadl.info/aadl/currentsite/tool/osate.html
[11] https://wiki.sei.cmu.edu/aadl/index.php/Osate_2
[12] http://www.eclipse.org/Xtext/
[13] http://bless.santoslab.org/node/5
[14] http://libre.adacore.com/tools/ocarina/

The BLESS tool framework is implemented as a publicly available open source plug-in for OSATE (mentioned in section 2.3.1). It includes an editor for BLESS specifications and an environment operating the BLESS proof engine. [LCH13]

Some BLESS constructs can be translated into SPARK contracts, which is part of this thesis. Additionally, BLESS allows to model behavior of components.

[MORE DETAILS? EXAMPLES?]

2.5 SPARK Ada

First version of Ada programming language - Ada 83 - was designed to meet the US Department of Defense Requirements formalized in "Steelman" document[15]. Since that time, Ada evolved. There were Ada 95, Ada 2005 and Ada 2012 (released in December 10, 2012)[16]. Ada is actively used in many Real-World projects[17], e.g. Aviation (Boeing[18]), Railway Transportation, Commercial Rockets, Satellites and even Banking. One of the main goals of Ada is to ensure software correctness and safety. Due to this requirements, Ada minimize developer responsibility in comparison to other programming languages (see figure 2.5). It is achieved not only by language capabilities, but also by tools for verification.

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

[15] http://www.adahome.com/History/Steelman/steelman.htm
[16] http://www.ada2012.org
[17] http://www.seas.gwu.edu/ mfeldman/ada-project-summary.html
[18] http://archive.adaic.com/projects/atwork/boeing.html
[19] http://www.slideshare.net/AdaCore/ada-2012

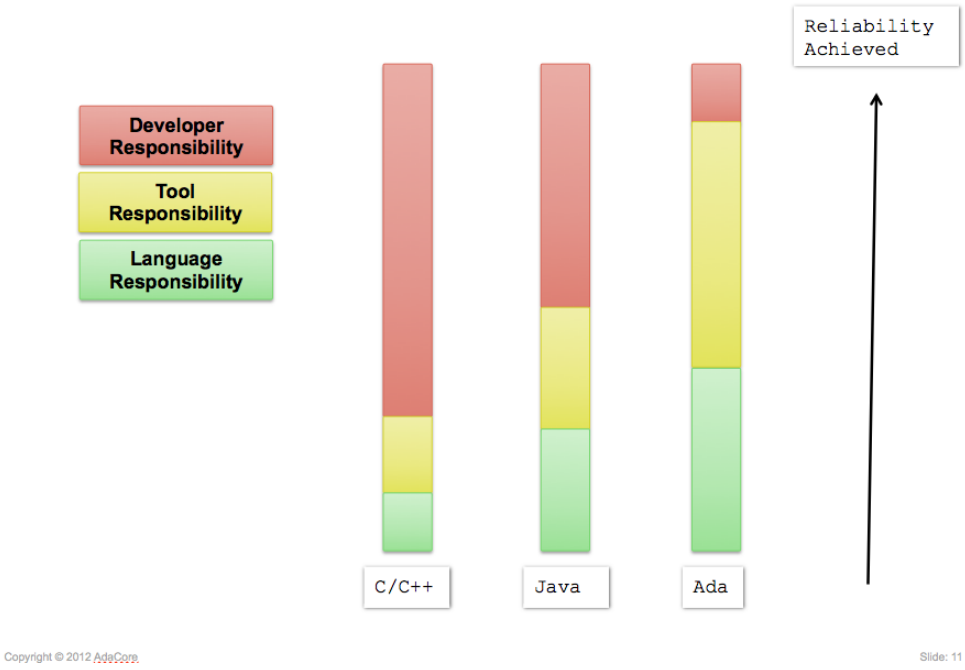

Figure 2.5: *Developer responsibility in Ada*[19].

produce verification conditions (VC's), which are mathematical theorems. If the VC's can be proved then the refinement step will be known to be valid. However if the VC's cannot be proved then the refinement step may be erroneous[20]. Sample Verification Condition contains checks for:

- array index out of range
- type range violation
- division by zero
- numerical overflow

[Add more examples where SPARK is used?]

SPARK provides a significant degree of automation in proving exception freedom [IEC+06]. Some Ada constructs are excluded from SPARK to make static analysis feasible [IEC+06].

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

Additionally SPARK contains tool-set for software verification (see section 2.6). In real-world applications, the embedded critical components are written in SPARK while the non-critical components are written in Ada.

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. SPARK annotations contains code contracts, which are analyzed by verification tools, but ignored by Ada compiler.

Listing 2.2 presents simple procedure with code contracts. It increments variable given as parameter by 1. The `derives` clause specify variable dependency. It future value depends on its current value. There is precondition saying that the value has to be lower than maximum value of `Integer` type. There is also post condition, which states that the value of variable (given as parameter) after the procedure execution has to be equal to its previous value incremented by 1 ('~' attached to variable means value of this variable, before procedure execution).

```
procedure Increment (X : in out Integer);
--# derives X from X;
--# pre X < Integer'Last;
--# post X = X~ + 1;
```

Listing 2.2: *Sample SPARK procedure with code contracts*

SPARK 2014[21] (based on Ada 2012) is under development. There is partial tool support (in GNAT Programming Studio), but some language features are still not supported. It is worth to mention, that Ada 2012 contains code contracts (which was inspired by previous versions of SPARK). Thus SPARK 2014 is just a subset of Ada 2012. [DEL+14] It contains all features of Ada 2012 except:

[21] http://www.spark-2014.org

- 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

Table 2.1 presents fundamental SPARK 2005 annotations and their equivalents in SPARK 2014 and Ada 2012.

Table 2.1: 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

Continued on next page

Table 2.1 – continued from previous page

SPARK 2005	SPARK 2014	Description
`--# post`	`Post`	post condition
`--# assert`	`Assert`	assertion

Sample mapping from SPARK 2005 to 2014 is shown in the table 2.2. Complete mapping can be found in SPARK 2014 documentation[22] [AL14].

Table 2.2: *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);`

The previous example (listing 2.2) translated to SPARK 2014 is shown in figure 2.3.

```
procedure Increment (X : in out Integer)
with Depends => (X => X),
  Pre => (X < Integer'Last),
  Post => (X = X'Old + 1);
```

[22] http://docs.adacore.com/spark2014-docs/html/lrm/mapping-spec.html

Listing 2.3: *Sample SPARK 2014 procedure and Code Contracts*

It is possible to mix SPARK 2014 with Ada 2012. However, only the part which is SPARK 2014 compliant will be verified. As mentioned before, 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 not safety-critical modules. SPARK 2014 does not contains Examiner like SPARK 2005. Instead, proofs are made by gnatPROVE (see section 6.5).

The most popular IDE for SPARK Ada is GNAT Programming Studio[23] (see section 2.5.2). There is also Ada plug-in for Eclipse - GNATbench[24] created by AdaCore.

2.5.1 GNAT compiler

GNAT compiler is Ada compiler created by AdaCore[25]. It is part of GNU Compiler Collection (GCC). The GNU Compiler Collection includes front ends for C, C++, Objective-C, Fortran, Java, Ada, and Go. It is one of the most popular compiler systems. It is included in all Linux distributions. GNU is open source, published on GNU General Public License. GCC is divided into front end and back end. It allows to create new front end for some language and reuse existing back end.

GNAT supports Ada 2012, Ada 2005, Ada 95 and Ada 83. The front-end and run-time are written in Ada. To make compilation easier, GNAT provides `gnatmake` tool. It takes as an argument project file (.gpr) or main program file (file, which contains main procedure) and builds entire program automatically. `gnatmake` invokes GCC to perform the actual compilation. It check all dependencies contained in `.ali` files. Each invocation of GCC produce object file (`.o`) and Ada Library Information file (`.ali`). Once compilation is done, `gnatmake` invokes `gnatbind`

[23] http://libre.adacore.com/tools/gps
[24] https://www.adacore.com/gnatpro/toolsuite/gnatbench/
[25] http://www.adacore.com

tool to check consistency and generate a main program. Then `gnatlink` performs linking using binding output and all object files.

GNAT compiler is available for all most popular platforms: Windows, Linux and MacOS. AdaCore, released also GNAT cross-compiler for ARM devices. However, for now, the compilation has to be done on 32-bit Linux platform.

2.5.2 GNAT Programming Studio (GPS)

GNAT Programming Studio (GPS) is Integrated development environment for SPARK Ada. It allows to easily manage and compile SPARK Ada projects using .gpr file. GPS includes set of verification tools. More precisely GUI for setting up their options, running them and analyze results. Additionally, it enables to create plug-ins using Python and PyGTK[26]. Sireum Bakar (developed by SAnToS lab) is GPS plug-in written in Python and PyGTK. The same with other plug-ins created by AdaCore like SPARK Examiner or GNATprove.

There are two versions of GPS: free (GPL) and commercial (Pro). There are version for all most popular platforms: Windows, Linux and MacOS.

2.5.3 Ravenscar Tasking Subset

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]. RavenSPARK is SPARK subset of the Ravenscar Profile. 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

[26] http://docs.adacore.com/gps-docs/users_guide/_build/html/extending.html

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]

Ravenscar profile is available in SPARK 2005, but not yet in SPARK 2014[27] [AL14]. Default profile (sequential) does not enable tasking. In other words, SPARK tools cannot analyze and reason about concurrent programs if Ravenscar profile flag is not provided.

To create a task, the task type has to be declared and task variable of this type. Ravenscar does not allows 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`. The range of available priority values is specified in the `System` package. The default range is 1 to 63. 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;
```

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

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

   task body Task2
   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 be not 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 following error: `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, **pragma** `Atomic` has to be used. However, there is restriction, that **pragma** `Atomic` cannot be applied to predefined type such as Integer. Thus, custom type has to be defined. It can be just rename of Integer. Then **pragma** `Atomic` can be applied 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*

It is important to mention, that **pragma** `Atomic` does not guaranty atomicity. In most cases, atomic types should not be used for tasking. Instead, protected types should be used. When an object is declared as atomic, it just means that it will be read from or written to memory atomically. The compiler will not generate atomic instructions or memory barriers when accessing to that object. **pragma** `Atomic` force compiler only to:

- check if architecture guarantees atomic memory loads and stores,
- disallow some compiler optimizations, like reordering or suppressing redundant accesses to the object

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 [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] is another good source. The limitations of Tasking in SPARK are reviewed in Audsley's and Welllings' paper [AW01].

2.6 SPARK Ada Verification

The goal of software verification is to assure software correctness and lack of errors. There are two types of verification:

- dynamic - performed during the execution of software, e.g. unit tests
- static - achieved by formal methods, mathematical calculations and logical evaluations

Dynamic verification starts with a set of possible test cases, simulates the system on each input, and observes the behavior. In general, it does not cover all possible executions. On the other hand, static verification establishes correctness for all possible execution sequences. Static and dynamic verification can be mixed, e.g. by generating test cases with static verification tools and then proving correctness with unit tests during runtime [DRH07].

Techniques for Static 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 consists of a verification tool-set:

- SPARKMake - generates index file (.idx) and meta file (.smf)

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

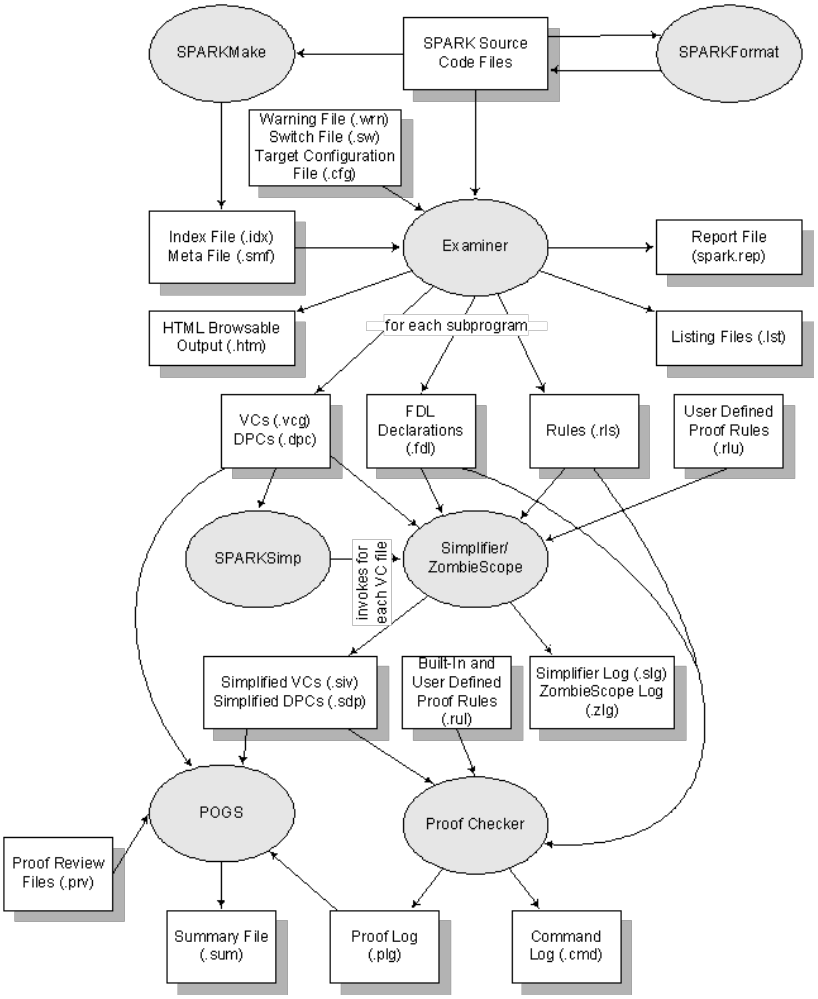

Figure 2.6: *Relationship of the Examiner and Proof Tools*[28].

- Examiner - check syntax, generates Verification Conditions (VCs) and Dead Path Conjectures (DPCs), and discharge (prove) them
- Simplifier - simplify and discharge VCs, which are not discharged by Examiner
- ZombieScope - find dead paths
- ViCToR - translate VCs and DPCs to format acceptable by SMT solver and prove correctness using specified SMT solver
- SPARKSimp - runs Simplifier or/and ZombieScope

- POGS - produces verification report
- Proof Checker - discharge VCs or DPCs not discharged by Examiner and Simplifier

Relationships between tools and verification flow is presented on figure 2.6. SPARK proof tools use FDL as the modeling language.

2.6.1 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

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? E.g.: method without contract, examine, add specification, pass Examiner.]

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) as mentioned in section 2.5.3.

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. Then, only a warning is returned by Examiner: `10 - The body of subprogram Main is hidden - hidden text is ignored by the Examiner.`

Examiner use SPARK index file (.idx) - generated by `SPARKMake` tool - 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 (figure 2.7)
- Set SPARK index file (to spark.idx generated by SPARKMake) (figure 2.8)
- (optionally) set configuration file (e.g. 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 (DPCs). 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).

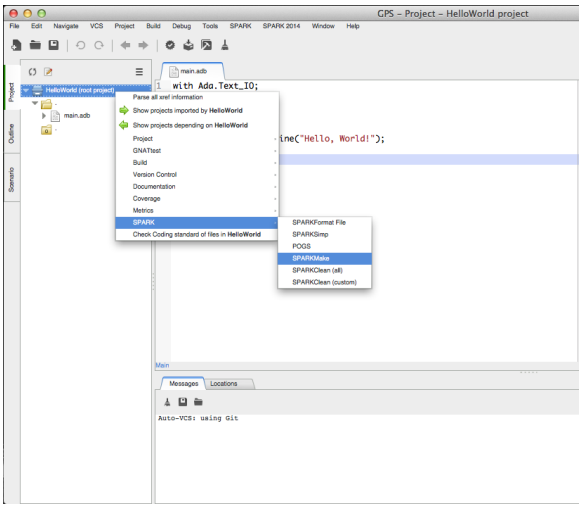

Figure 2.7: *Run SPARK Make*

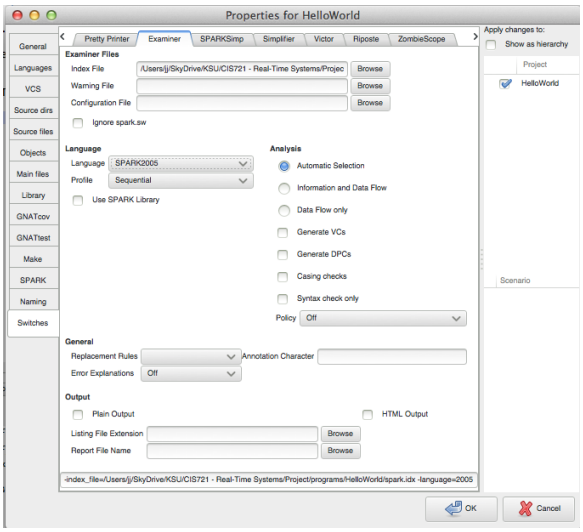

Figure 2.8: *Examiner Properties*

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.

Verification conditions

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

- preconditions: `--# pre`
- postconditions: `--# post`

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

$$\{P\}C\{Q\} \tag{2.1}$$

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 2.6.1), thus only precondition 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 is performed backwards, in other words: we start from post-conditions and consider what must holds before. Flow analysis is well described in chapter 11 of [Bar13].

If preconditions are not present, then the formula expresses that the post-condition holds always.

2.6.2 SPARK Simplifier

Simplifier, simplify verification conditions (VCs) generated by Examiner. It can also discharge (prove correctness) of those VCs, which are not proved by Examiner. [Tea11c] It takes as input `.vcg` files, `.fdl` files for its data declarations and - if available - proof-rule files (`.rls`, `.rlu`). Then it generates `.siv` files (simplified VCs) and `.slg` files (details about simplification, which has been made).

2.6.3 ZombieScope

ZombieScope is a SPARK tool, that analyze SPARK code to find dead paths, i.e. paths through the code that can never be executed. Program, which contains dead paths may not necessarily be incorrect, but a dead path is an indication of a potential code issue.

ZombieScope reads `.dpc` files generated by the Examiner. In order to generate dead path conjectures, `-dpc` flag has to be used or 'Generate DPCs' option has to be checked in Examiner

options, in GPS. It reads also `.fdl` files for its data declarations and the `.rls` file for proof-rules if present. ZombieScope generates two output files: `.sdp` file (dead path summary) and `.zlg` file (details about underlying contradiction search performed). ZombieScope is invoked by SPARKSimp by default and the summary file generated by POGS includes information about the dead path analysis.

2.6.4 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 for verification and proving the correctness of programs. ViCToR is integrated with SPARKSimp and POGS. To invoke ViCToR from SPARKSimp, flag `-victor` has to be used.

2.6.5 Proof Checker

Proof Checker is advanced verification tool, which require considerable experience in verification of SPARK programs. It is interactive program, which enables the user to direct the Checker to explore the use of various strategies and rules on the condition to be proved. Proof Checker can keep a log of the progress of a proof in `plg` file. It also keep command record in `cmd` file. More details about Proof Checker can be found in chapter 12 of [Bar13].

2.6.6 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) to all `.vcg` files and ZombieScope to all `.dpc` files, which it finds in a directory tree. [Tea10]

2.6.7 Proof Obligation Summarizer (POGS)

The Proof ObliGation Summarizer tool (POGS) reads and understands the structure of the verification conditions (`.vcg` files), their simplified version (`.siv` files) and dead path conjectures (`.dpc` files). It reports the status of proofs and dead path analyses in a human-readable, text form. [Tea11a]

2.6.8 AUnit

AUnit is Unit Test Framework for Ada language. It can be also applied for verify SPARK Ada programs. It was created based on Java JUnit (created by Kent Beck, Erich Gamma) and C++ CppUnit (created by M. Feathers, J. Lacoste, E. Sommerlade, B. Lepilleur, B. Bakker, S. Robbins) unit test frameworks [Ada14]. Similar like mentioned frameworks it enables simple test cases testing, fixtures, suites and provides reporting [Fal14]. As mentioned at the beginning of section 2.6 it is used mainly for dynamic verification.

GNAT Programming Studio can generate test cases skeleton for all subprograms. It can be generated using Tools -> GNATtest -> Generate unit test setup. This generator creates new project with AUnit tests. Project for which tests are generated is referenced in new generated project. In order to run tests, the test project has to be opened in GNAT Programming Studio. The project is created in `[project_dir]/gnattest/harness/test_[proj_name].gpr`. It generates empty (not implemented) test for each subprogram in project. To add/edit/remove tests or rename names, three files has to be edited:

- `[some_package]-test_data-tests.ads`
- `[some_package]-test_data-tests.adb`
- `[some_package]-test_data-tests-suite.adb`

Test has to be declared in `[some_package]-test_data-tests.ads` and implemented in `[some_package]-test_data-tests.adb`. Then it has to be added to test suite in `[some_package]-test_data-tests-suite.adb` file.

Tests can be also created manually. Then the AUnit distribution has to be referenced in project file and all test cases (and suits) has to be implemented by hand.

2.6.9 Sireum Bakar

Sireum[29] is a long-term research conducted by SAnToS lab at Kansas State Univeristy. Its goal is to develop an over-arching software analysis platform that incorporates various static analysis techniques such as data-flow framework, model checking, symbolic execution, abstract interpretation, and deductive reasoning techniques (e.g., using weakest precondition calculation). It can be used to build various kinds of software static analyzers for different kinds of properties.

It uses the Pilar language [SC12] as intermediate representation. Any language which can be translated to Pilar can be analyzed by Sireum. For now, there is translator for SPARK and Java.

Bakar is a toolset for analyzing SPARK Ada programs (Bakar means "spark" in Indonesian). Sireum Bakar currently includes:

- Kiasan - functional behaviors verification tool
- Alir - information flow analysis tool

Sireum distribution is available for Windows (32-bit, 64-bit), Linux (32-bit, 64-bit) and MacOS (64-bit). It can be downloaded from `http://www.sireum.org/`.

[29] http://www.sireum.org/

Bakar Kiasan

Bakar Kiasan [BHR+11] is a fully automated tool for verifying functional behaviors of SPARK programs specified as software contract (Kiasan means "symbolic" in Indonesian). Kiasan use symbolic execution technique. It provides various helpful feedback including generation of counter example for contract refutation, test cases for an evidence of contract satisfaction, verification reports, visual graphs illustrating pre/post states of SPARK procedures/functions, etc. It is much easier for hazard analysis than e.g. analysis of `.vcg` files generated by SPARK Examiner.

There exists Kiasan Plug-in for GNAT Programming Studio (GPS). Version 1, for GPS 5, supports SPARK 2005. Version 2, for GPS 6, which supports 2014 is under development. Both plug-ins are created by author of this thesis in Python and PyGTK. There is also plug-in for Eclipse, but only for SPARK 2005 programs.

Bakar Kiasan does not support Ravenscar profile. Thus, it can be used only for sequential programs verification. Figure 2.9 depicts sample Kiasan analysis result. Kiasan window has two parts: list of units (packages and subprograms) and analysis cases with pre and post states. Every unit has associated statistics:

- T# - Test cases (expected behavior)
- E# - Exception cases (unexpected behavior)
- Instruction coverage - amount of code covered by Kiasan analysis
- Branch coverage - number of branches covered by analysis (0% in 100% instruction coverage means, that there is no branches in analyzed unit)
- Time in which analysis was performed

After double click on some unit, code which is executed during execution of this unit is highlighted. Additionally below the list of units, there is a combo box which contains all

test cases associated with selected (by double click) unit. Once, some case is selected, code coverage equivalent to this test case is highlighted. Additionally, below combo box, there are states of unit execution. On the left hand side, there is pre-state, and on the right hand side there post-state of analysis. Variables with red font color, in post-state, are those which are changed in result of unit execution. The new created variables (during unit execution) are blue, but there are not present in figure 2.9.

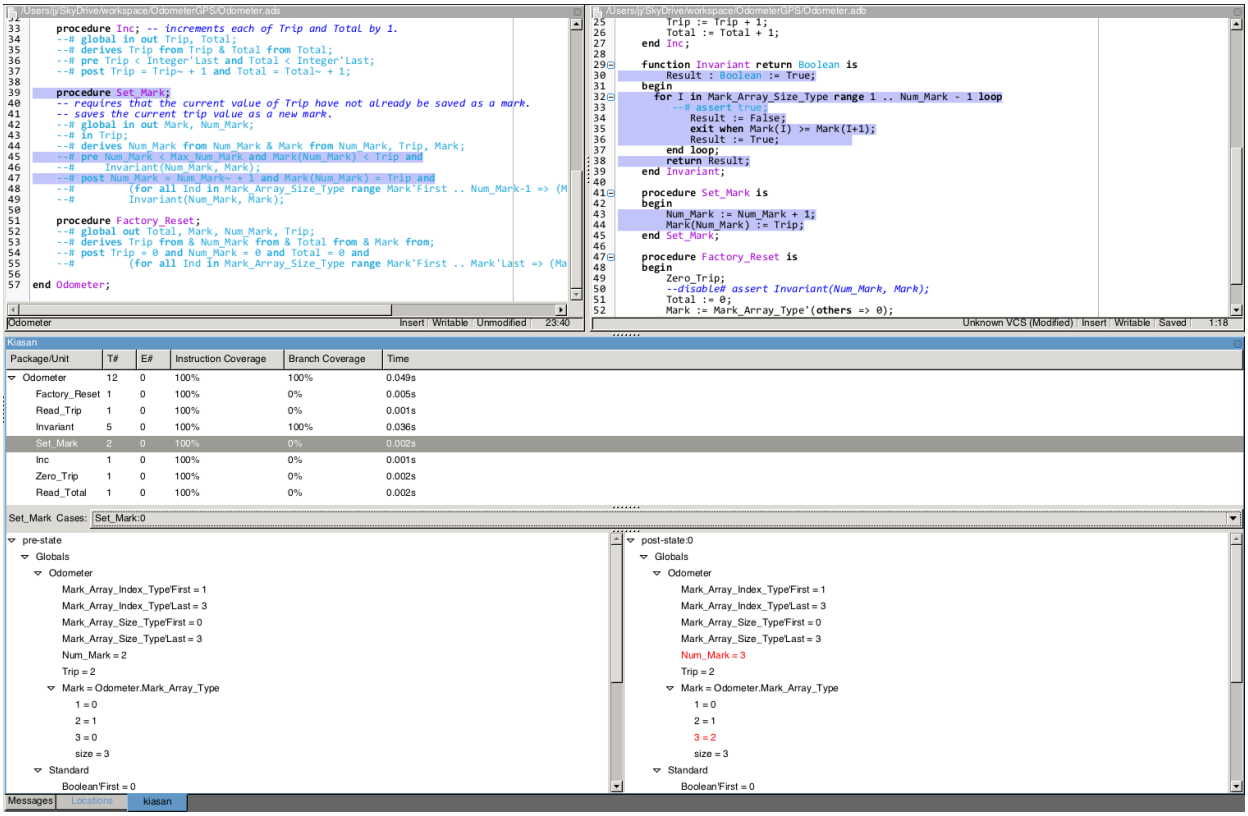

Figure 2.9: *Bakar Kiasan report*

Bakar Kiasan is useful especially, for solving verification issues. It can generate counter examples, which help to fix the code. [add screenshot with error case and discuss result?]

Bakar Alir

Alir is an information flow analysis tool for reasoning about SPARK's derive clauses/information flow (Alir means "flow" in Indonesian). Alir visualizes information flows to ease engineers in understanding information dependencies crucial for specifying and verifying SPARK's derive clauses. It provides various configurable intra-procedural and inter-procedural analyses. The inter-procedural analyses are control flow analysis, reaching definition analysis and data dependence analysis. The inter-procedural analysis in Alir includes building, System Dependence Graph (SDG), slicing and chopping on SDG. [Thi11]

2.6.10 GNAT Prove

GNATprove [30] is a formal verification tool for SPARK 2014 programs. It is based on the GNAT compiler. GNATprove interprets SPARK Ada annotations exactly like they are interpreted at run time during tests. It can prove that subprograms respect their contracts, expressed as preconditions and postconditions in the syntax of Ada 2012. The tool automatically discovers the subset of subprograms which can be formally analyzed. GNATprove is currently available for x86 linux, x86 windows and x86-64 linux.

[Add more details? Some example like in Kiasan section?]

2.7 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. Prototyping Embedded Systems using AADL lasts for a few years [CB09]. There are already existing tools, which performs code generation based on AADL:

- Ocarina

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

- Ramses

2.7.1 Ocarina

Ocarina [LZPH09] is a tool suite, which contains plug-ins for code generation, model checking and analysis. The code generation plug-in generates code from an AADL architecture model to an Ada or C 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. Ocarina is written in Ada.

There is plug-in for OSATE (see section 2.3.1), which enables code generation. Example AADL models, suitable for being an input of Ocarina are available on github repository: `https://github.com/yoogx/polyorb-hi-ada/tree/master/examples/aadlv2`.

Since mid-2009, Telecom ParisTech is no longer involved in Ocarina, and is developing another AADL tool-chain, based on Eclipse, codenamed RAMSES [Hug13].

[Include some examples and generated code? E.g. prod-cons example?]

2.7.2 RAMSES

RAMSES (Refinement of AADL Models for Synthesis of Embedded Systems) is a model transformation and code generation tool. It is written in Java. RAMSES produces C code, but does not generate Ada. It simplify AADL models, in order to generate C code. Simplified AADL models contain behavior annex subclauses. RAMSES can be used as OSATE plug-in or standalone application.

[I didn't find much about RAMSES online...]

Chapter 3

PCA Pump

Patient Controlled Analgesia (PCA) pump is a medical device, which allows the patient to self-administer small doses of narcotics (usually Morphine, Dilaudid, Demerol, or Fentanyl). PCA pumps are commonly used after surgery to provide a more effective method of pain control than periodic injections of narcotics. A continuous infusion (called a basal rate) permits the patient to receive a continuous infusion of pain medication. There is no need for a clinician to administer it. Patient can also request additional boluses, but only in specified intervals. It prevents from over infusion. In addition to basal and patient bolus, clinician can also request bolus called clinician bolus or square bolus.

Figure 3.1: *Patient Controlled Analgesia (PCA) pump*

Figure 3.1 shows LifeCare PCA pump. On the left hand side, there is drug reservoir. On the right - clinician panel, which allows to control the pump. Figure 3.2 shows PCA

Pump, made by company Alaris.

Figure 3.2: *Alaris Pump*

PCA Pump is safety-critical device which works in standard process control loop depicted in the figure 3.3. The controller obtains information about (observes) the process state from measured variables (feedback) and uses this information to initiate action by manipulating controlled variables to keep the process operating within predefined limits or set points (the goal) despite disturbances to the process. Such as different air pressure or device position (gravity impact). In general, the maintenance of any open-system hierarchy (either biological or man-made) will require a set of processes in which there is communication of information for regulation or control. [Lev12]

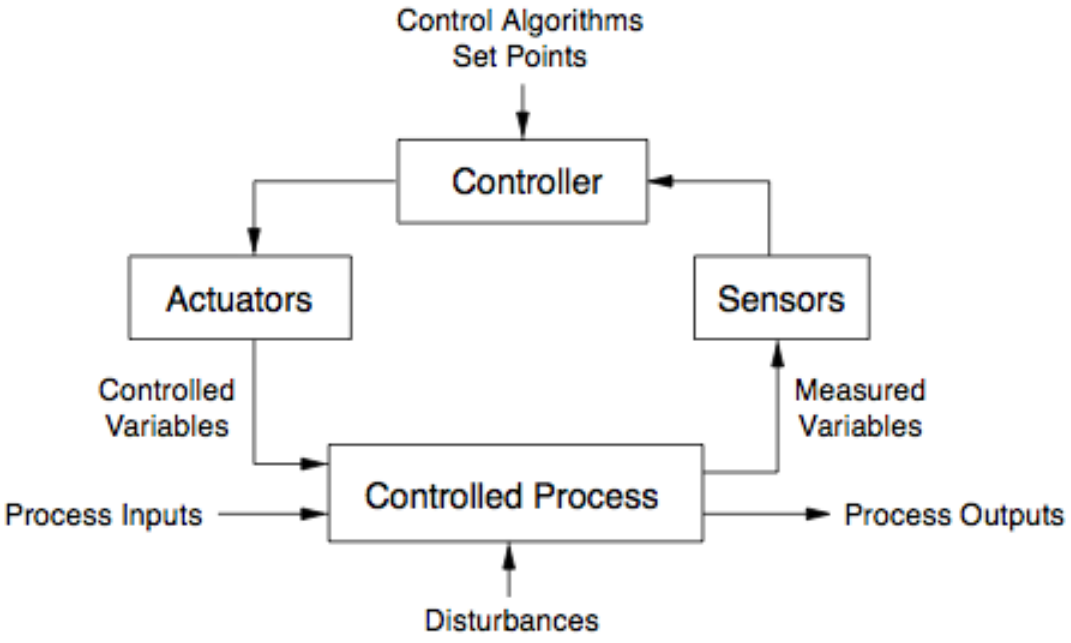

Figure 3.3: *Standard Process Control Loop.*

PCA Pump actuator is motor, which pump drug to the patient's vein. Controlled process is dosing the drug. Sensors measure amount of dosed drug. They might be used for double-check if ordered (by controller) amount of drug was appropriately delivered. Sometimes there might be some distrubances caused by mechanical issues and environmental conditions. Controller issues appropriate actions based on informations from sensors and clinician or patient's commands. High level overview of PCA Pump is depicted in the figure 3.4.

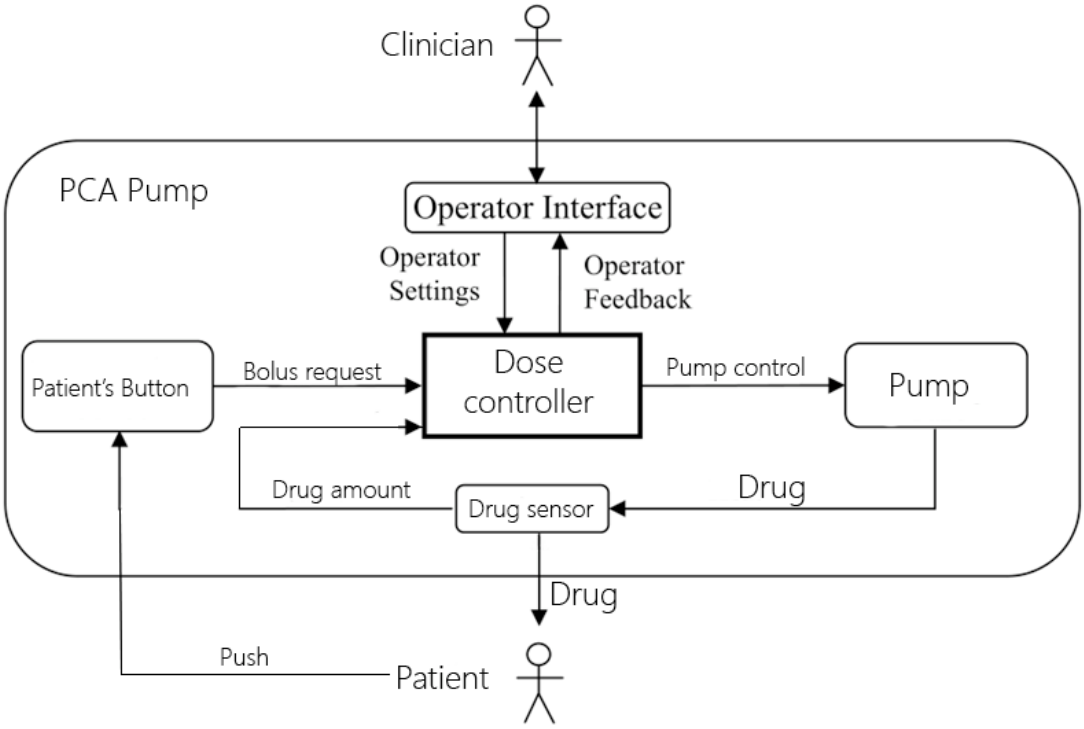

Figure 3.4: *PCA Pump system*

One of the hazards of using PCA pumps, is that there is inadequate monitoring of patient's levels of oxygen and carbon dioxide. Nursing staff on general medical units typically track respiration rate and other vital signs every four hours, which is not enough. There should be a way to monitor levels continuously. Additionally, it can be hard to tell if a person's breathing rate is dangerously low in certain circumstances. There are cases, where

lack of monitoring carbon dioxide level caused death.[1]

Another hazard is human mistake. For example, there is a case when nurse used a 5 mg/mL morphine cassette because a 1 mg/mL cassette was not available, but she programmed PCA Pump like for 1 mg/mL concentration. In addition to lack of monitoring of the pulse, patient died.[2]

As mentioned in chapter 2, the solution to that problem is medical devices interoperability. In addition, less human error-prone device is needed. It can be assured by using more than one system for their detection.

3.1 PCA Pump Requirements Document

Requirements of "Open Source PCA Pump" [LHC13] are captured in "Integrated Clinical Environment Patient-Controlled Analgesia Infusion Pump System Requirements" document [Lar14] created by Brian Larson. It is formalized set of capabilities, which Open PCA Pump should have, based on consultations with domain experts, FDA and Brian Larson's expertise gained while he was working in the medical device industry.

Conceptual model of Open PCA pump is depicted in the figure 3.5. As mentioned earlier, the pump is connected to ICE so it may be integrated with ICE apps and displays. The interface must provide prescription and patient information, current status to be displayed remotely on a supervisor user interface, and a means to stop infusing upon human command, or determination of an ICE app. Such an ICE app could monitor a patient's blood oxygenation and pulse rate, stopping the pump if depressed respiratory function is indicated. [Lar14]

Additionally, it cooperates with Drug Library, which contains information about drugs and its properties (like concentration). Data needed for pump operation, are captured on

[1] http://abcnews.go.com/Health/parents-warn-pca-pumps-daughters-death/story?id=16796805
[2] http://webmm.ahrq.gov/case.aspx?caseID=291

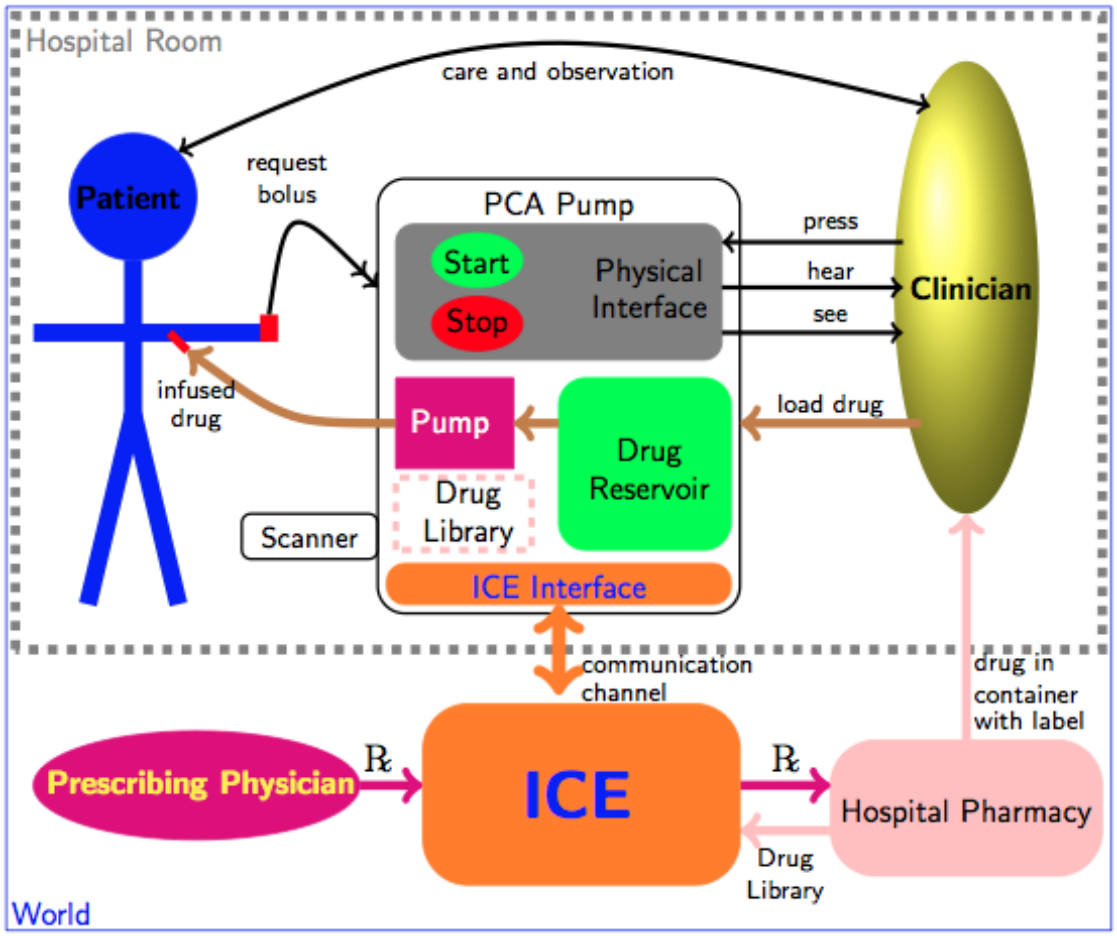

Figure 3.5: *Open PCA Pump concept*

electronic prescription, which contains:

- Patient's name
- Drug name
- Drug code
- Drug concentration
- Initial volume of drug in the vial
- Basal flow rate - the rate of continuous infusion

- Volume to be infused (VTBI) on patient's request
- Maximum amount of drug allowed per hour
- Minimum time between patient boluses
- Date, in which prescription has been filled
- Prescribing physician's name
- Pharmacist name

Pain medication is prescribed by a licensed physician, which is dispensed by the hospital's pharmacy. The drug is placed into a vial labeled with the name of the drug, its concentration, the prescription, and the intended patient. A clinician loads the drug into the pump, and attaches it to the patient. The pump infuses a prescribed basal flow rate which may be augmented by a patient-requested bolus or a clinician-requested bolus. This allows additional pain medication in response to patient need within safe limits. [Lar14]

Prescription captures all data needed for basal infusion and patient requested boluses (referred as bolus). In addition to that, Open PCA Pump allows Clinician Requested Bolus (refereed as square bolus). In order to do that, clinician has to enter the time (through PCA Pump panel) in which VTBI, specified in prescription, will be infused.

There can occur situations in which the maximum drug amount infused may exceed the allowed limit. E.g. when clinician issues too many square boluses. In such case, pump is switched to Keep Vein Open (KVO) mode, which has 1 ml/hr drug rate. Pump switches to KVO rate also when ICE interface request it. It may happen e.g. if patient's oxygen level is low. To recover from KVO state, pump has to be restarted by clinician in order to continue operation. In Summary, Open PCA Pump has following modes:

- Stopped

- Basal rate
- Patient's bolus (bolus)
- Clinician bolus (square bolus)
- Keep Vein Open (KVO)

There are also other scenarios, which are captured by Requirements Document [Lar14], like scanner to enable automatic entry of patient's and prescription data, occlusion detection, hardware errors alarms etc. Detailed overview of Open PCA Pump Requirements can be found in [LHC13].

[MORE DETAILS ABOUT PUMP?] [ADD STATE MACHINE IMAGE, LIKE IN UMINN REQ COD?]

3.2 PCA Pump AADL/BLESS Models

In addition to PCA Pump Requirements Document [Lar14], Brian Larson created AADL model with formal behavioral specifications written in his BLESS framework. AADL model, graphical representation is depicted on figure 3.6.

AADL model captures structure of device. BLESS - its behavior. Listing 3.1 shows `Rate_Controller` thread from `PCA_Operation` component with BLESS assertions in thread declaration and BLESS behavioral description in thread implementation. The thread declaration contains input and output ports. In addition to some of them, BLESS assertions are present. Assertions are defined in BLESS annex in thread implementation. In addition to assertions, states and transitions defined in thread implementation can potentially be translated into working SPARK Ada program. Presence of timing properties in states and transitions makes translation extremely difficult, thus there are omitted in this thesis and only assertions are considered. [TRUNCATE CODE LISTING?]

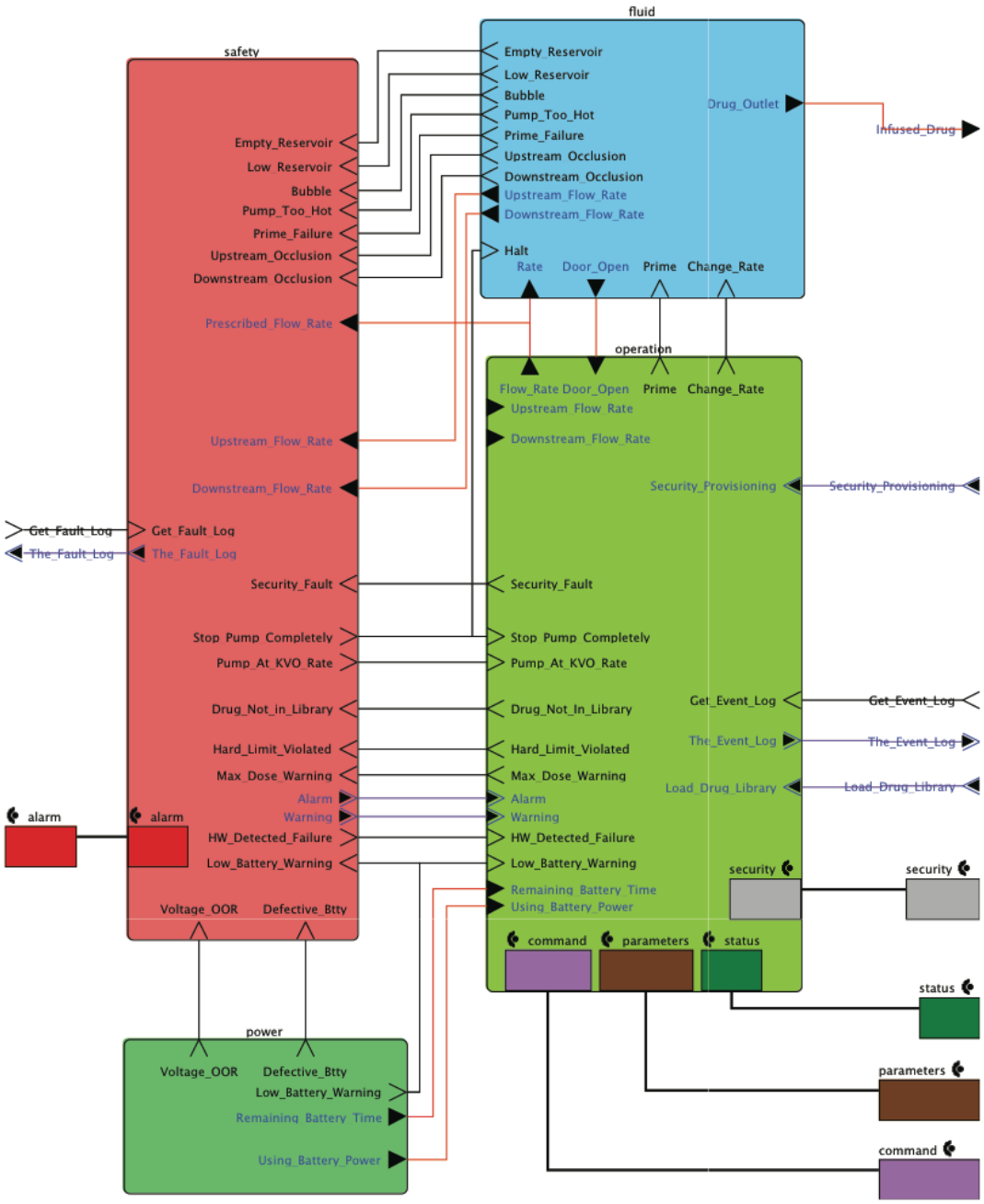

Figure 3.6: *Open PCA Pump AADL model*

```
thread Rate_Controller
features
    Infusion_Flow_Rate: out data port PCA_Types::Flow_Rate
      {BLESS::Assertion => "<<:=PUMP_RATE()>>";};
    System_Status: out event data port PCA_Types::Status_Type;
    Begin_Infusion: in event port
      {BLESS::Assertion => "<<Rx_APPROVED()>>";};
    Begin_Priming: in event port;
    End_Priming: in event port;
    Halt_Infusion: in event port;
    Square_Bolus_Rate: in data port PCA_Types::Flow_Rate
      {BLESS::Assertion => "<<:=SQUARE_BOLUS_RATE>>";};
    Patient_Bolus_Rate: in data port PCA_Types::Flow_Rate
      {BLESS::Assertion => "<<:=PATIENT_BOLUS_RATE>>";};
    Basal_Rate: in data port PCA_Types::Flow_Rate
      {BLESS::Assertion => "<<:=BASAL_RATE>>";};
    VTBI: in data port PCA_Types::Drug_Volume
      {BLESS::Assertion => "<<:=VTBI>>";};
    HW_Detected_Failure: in event port;
    Stop_Pump_Completely: in event port;
    Pump_At_KVO_Rate: in event port;
    Alarm : in event data port PCA_Types::Alarm_Type;
    Warning : in event data port PCA_Types::Warning_Type;
    Patient_Request_Not_Too_Soon: in event port
      {BLESS::Assertion => "<<:=PATIENT_REQUEST_NOT_TOO_SOON(now)>>";};
    Door_Open: in data port Base_Types::Boolean;
    Pause_Infusion: in event port;
    Resume_Infusion: in event port;
    CP_Clinician_Request_Bolus: in event port;
    CP_Bolus_Duration: in event data port ICE_Types::Minute;
    Near_Max_Drug_Per_Hour: in event port  --near maximum drug infused in any hour
      {BLESS::Assertion => "<<PATIENT_NEAR_MAX_DRUG_PER_HOUR()>>";};
    Over_Max_Drug_Per_Hour: in event port  --over maximum drug infused in any hour
      {BLESS::Assertion => "<<PATIENT_OVER_MAX_DRUG_PER_HOUR()>>";};
    ICE_Stop_Pump: in event port;
  properties
    Thread_Properties::Dispatch_Protocol => Aperiodic;
end Rate_Controller;

thread implementation Rate_Controller.imp
annex BLESS
{**
assert
--the infusion flow rate shall be:
--  =0, after stop pump completely (from safety architecture),
--      clinician pressed stop button
--  =KVO rate, after KVO rate command, or
--      exceeded max drug per hour
--      some alarms,
--  =max rate, during patient-requested infusion
--  =square bolus rate, during clinician-commanded infusion
--  =priming rate during pump priming
--  =basal rate, otherwise
<<HALT : :(la=SafetyPumpStop) or (la=StopButton) or (la=EndPriming)>>  --pump at 0 if stop button, or
   safety architecture says, or done priming
<<KVO_RATE : :(la=KVOcommand) or (la=KVOalarm) or (la=TooMuchJuice)>>  --pump at KVO rate when commanded,
   some alarms, or excedded hourly limit
<<PB_RATE : :la=PatientButton>>  --patient button pressed, and allowed
<<CCB_RATE : :(la=StartSquareBolus) or (la=ResumeSquareBolus)>>  --clinician-commanded bolus start or
   resumption after patient bolus
<<PRIME_RATE : :la=StartPriming>>  --priming pump
<<BASAL_RATE : :(la=StartButton) or (la=ResumeBasal) or (la=SquareBolusDone)>>  --regular infusion
<<PUMP_RATE : :=
  (HALT()) -> 0,                                    --no flow
```

```
  (KVO_RATE()) -> PCA_Properties::KVO_Rate,          --KVO rate
  (PB_RATE()) -> Patient_Bolus_Rate,  --maximum infusion upon patient request
  (CCB_RATE()) -> Square_Bolus_Rate,                 --square bolus rate=VTBI/duration, from data port
  (PRIME_RATE()) -> PCA_Properties::Prime_Rate,      --pump priming
  (BASAL_RATE()) -> Basal_Rate                       --basal rate, from data port
>>
invariant <<true>>
variables
  --time of last action
  tla :BLESS_Types::Time := 0;
  la :   --last action
    enumeration (
      SafetyStopPump,   --safety architecture found a problem
      StopButton,       --clinician pressed stop button
      KVOcommand,       --from control panel (clinician) or ICE (app) to pump Keep-vein-open rate
      KVOalarm,         --some alarms should pump at KVO rate
      TooMuchJuice,     --exceeded max drug per hour, pump at KVO until prescription and patient are re-
   authenticated
      PatientButton,    --patient requested drug
      ResumeSquareBolus,--infusion of VTBI finished, resume clinician-commanded bolus
      ResumeBasal,      --infusion of VTBI finished, resume basal-rate
      StartSquareBolus, --begin clinician-commanded bolus
      SquareBolusDone,  --infusion of VTBI finished
      StartPriming,     --begin pump/line priming, pressed "prime" button
      EndPriming,       --end priming, pressed "prime" button again, or time-out
      StartButton       --start pumping at basal rate
    );
  pb_duration :BLESS_Types::Time  --patient button duration = VTBI/Patient_Bolus_Rate
    <<PB_DURATION : :pb_duration=(VTBI/Patient_Bolus_Rate)>>;
states
  PowerOn : initial state;        --power-on
  WaitForRx : complete state;   --wait for valid prescription
  CheckPBR : state  --check Patient_Bolus_Rate is positive
    <<Rx_APPROVED()>>;
  RxApproved : complete state   --prescription verified
    <<Rx_APPROVED() and PB_DURATION()>>;
  Priming : complete state      --priming the pump, 1 ml in 6 sec
    <<(la=StartPriming) and (Infusion_Flow_Rate@now = PCA_Properties::Prime_Rate) and PB_DURATION()>>;
  WaitForStart : complete state   --wait for clinician to press 'start' button
    <<HALT() and (Infusion_Flow_Rate@now=0) and PB_DURATION()>>;
  PumpBasalRate : complete state  --pumping at basal rate
    <<((la=StartButton) or (la=ResumeBasal)) and (Infusion_Flow_Rate@now=Basal_Rate@now) and PB_DURATION()
   >>;
  PumpPatientButtonVTBI : complete state  --pumping patient-requested bolus
    <<(la=PatientButton) and PB_DURATION()
      and (Infusion_Flow_Rate@now=Patient_Bolus_Rate)>>;
  PumpCCBRate : complete state    --pumping at clinician-commanded bolus rate
    <<((la=StartSquareBolus) or (la=ResumeSquareBolus)) and (Infusion_Flow_Rate@now=Square_Bolus_Rate@now)
   and PB_DURATION()>>;
  PumpKVORate : complete state    --pumping at keep-vein-open rate
    <<((la=KVOcommand) or (la=KVOalarm) or (la=TooMuchJuice))  and PB_DURATION()
      and (Infusion_Flow_Rate@now=PCA_Properties::KVO_Rate)>>;
  PumpingSuspended : complete state  --clinician pressed 'stop' button
    <<((la=StopButton) or (la=SafetyStopPump)) and (Infusion_Flow_Rate@now=0)>>;
  Crash : final state;    --abnormal termination
  Done : final state    --normal termination
    <<Infusion_Flow_Rate@now=0>>;
transitions
--wait for valid prescription
  go : PowerOn-[ true ]->WaitForRx{};
--prescription validated
  rxo : WaitForRx-[ on dispatch Begin_Infusion ]-> CheckPBR{};
  pbr0 : CheckPBR-[ Patient_Bolus_Rate<=0 ]->Crash{}; --bad Patient_Bolus_Rate
  pbrok : CheckPBR-[ Patient_Bolus_Rate>0 ]->RxApproved
    {<<Rx_APPROVED() and (Patient_Bolus_Rate>0)>>  --likely will change from logic variable to predicate
```

```
   Rx_APPROVED()
      pb_duration := VTBI/Patient_Bolus_Rate  --calculate patient bolus duration
      --note division without knowing divsor is non-zero; should generate additional proof obligations for
   assignment using division
    <<Rx_APPROVED() and PB_DURATION()>>};
--clinician press 'prime' button
  rxpri : RxApproved-[ on dispatch Begin_Priming ]-> Priming
    {
    la :=StartPriming
        <<Begin_Priming@now and Rx_APPROVED() and (la = StartPriming) and PB_DURATION()>>
    ;
    Infusion_Flow_Rate!(PCA_Properties::Prime_Rate)  --infuse at prime rate
        <<(la = StartPriming) and Rx_APPROVED() and PB_DURATION() and
          (Infusion_Flow_Rate@now=PCA_Properties::Prime_Rate)>>
    };
--priming done, wait for start
  prd: Priming-[ on dispatch End_Priming or timeout (Begin_Priming) PCA_Properties::Prime_Time sec]->
   WaitForStart
    {
    la:=EndPriming
      <<HALT() and PB_DURATION()>>  --and Begin_Priming timed out
    ;
    Infusion_Flow_Rate!(0)   --stop priming flow
      <<HALT() and (Infusion_Flow_Rate@now=0) and PB_DURATION()>>
    };
--prime again
  pri: WaitForStart-[ on dispatch Begin_Priming ]-> Priming
    {
    la:=StartPriming
        <<Begin_Priming@now and PB_DURATION() and PRIME_RATE()>>
    ;
    Infusion_Flow_Rate!(PCA_Properties::Prime_Rate)  --infuse at prime rate
        <<PRIME_RATE() and PB_DURATION() and
          (Infusion_Flow_Rate@now=PCA_Properties::Prime_Rate)>>
   };
--clinician press 'start' button after priming
  sap: WaitForStart-[ on dispatch Begin_Infusion ]-> PumpBasalRate  --start after priming
    {
    la:=StartButton
      <<(la=StartButton) and Begin_Infusion@now  and PB_DURATION()>>
    ;
    Infusion_Flow_Rate!(Basal_Rate)   --infuse at basal rate
      <<(la=StartButton) and (Infusion_Flow_Rate@now=Basal_Rate@now) and PB_DURATION()>>
    };
--Patient_Request_Bolus during basal rate infusion
  pump_basal_rate:
  PumpBasalRate-[ on dispatch Patient_Request_Not_Too_Soon]-> PumpPatientButtonVTBI
    {
    la := PatientButton
      <<(la=PatientButton) and Patient_Request_Bolus@now and PB_DURATION()>>
    ;
    Infusion_Flow_Rate!(Patient_Bolus_Rate)   --infuse at patient button rate
      <<(la=PatientButton) and PB_DURATION()
        and (Infusion_Flow_Rate@now=Patient_Bolus_Rate)>>
    };  --end of pump_basal_rate
--VTBI delivered
  vtbi_delivered:
  PumpPatientButtonVTBI -[ on dispatch timeout (Infusion_Flow_Rate) pb_duration ms ]-> PumpBasalRate
    {
    la:=ResumeBasal
    ;
    <<(la=ResumeBasal) and PB_DURATION()>>  --and timeout of patient button duration
    Infusion_Flow_Rate!(Basal_Rate)   --infuse at basal rate
      <<(la=ResumeBasal) and (Infusion_Flow_Rate@now=Basal_Rate@now) and PB_DURATION()>>
    };  --end of vtbi_delivered
```

```
**};
end Rate_Controller.imp;
```

Listing 3.1: *Rate_Controller thread from PCA_Operation component with BLESS assertions*

3.3 BeagleBoard-XM

For Research and MDCF purposes, BeagleBoard-xM (an open-source hardware single-board computer produced by Texas Instruments), has been chosen as hardware platform for PCA pump prototyping.

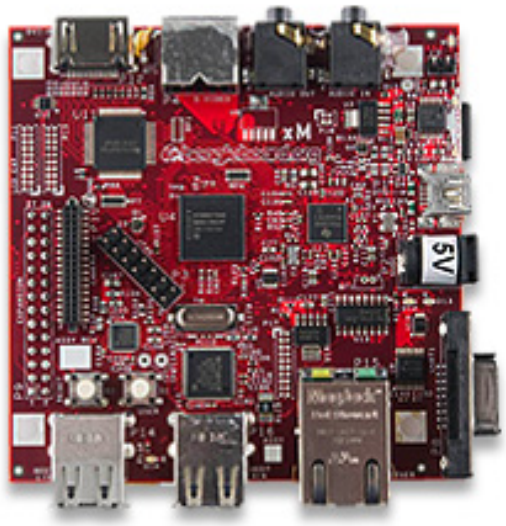

Figure 3.7: *BeagleBoard-xM*

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, which enables control of pump actuator.

Pulse-width modulation (PWM) is a technique for controlling analog circuits with a processor's digital outputs. The average value of voltage (and current) fed to the electrical load is controlled by turning the switch between supply and load on and off at a fast pace. The longer the switch is on compared to the off periods, the higher the power supplied to the load. Proportion of on and off periods is called the duty cycle and is expressed in percent. 100% means all the time on, 0% - all the time off. Figure 3.8 shows 10%, 30%, 50% and 90% duty cycles.

There is no existing SPARK Ada compiler running on ARM system. Hence, to compile

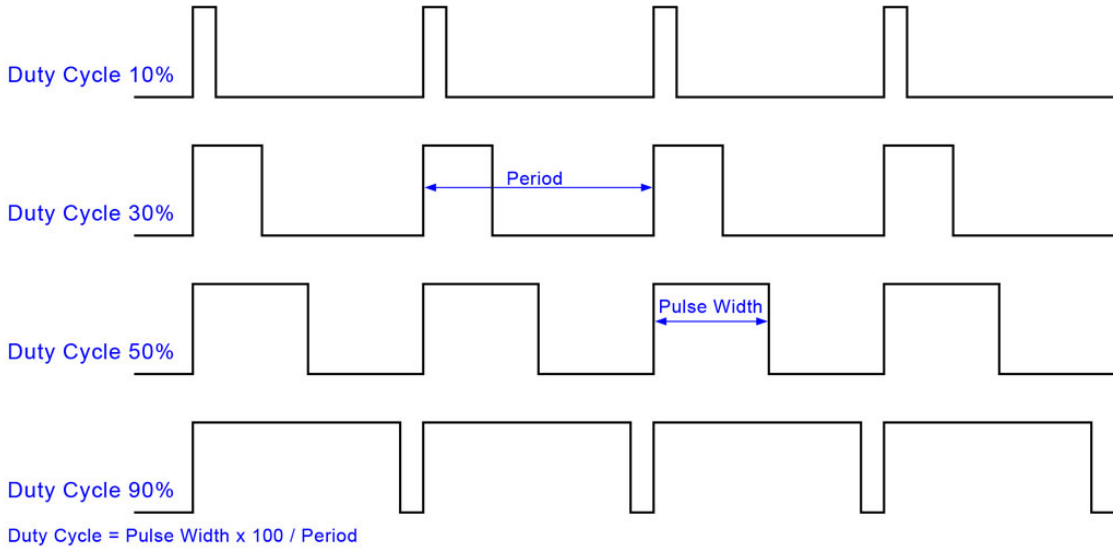

Figure 3.8: *An example of PWM duty cycles*

SPARK Ada program for ARM device, cross-compiler is needed. There is GNAT compiler [Hor09] created by AdaCore, but there was no cross-compiler for ARM. However, AdaCore was actively developing cross-compiler. They had working version in 2013, but tested only on their target Android-based device. In cooperation with AdaCore, cross-compiler for ARM was bundled and tested on BeagleBoard-xM. For now, GNAT cross-compiler works only on Linux 32-bit operating system.

In addition to USB ports, BeagleBoard-xM has also serial port and Ethernet port. It allows to copy programs compiled on Linux, using all three types of ports.

Chapter 4

AADL/BLESS to SPARK Ada translation

This chapter presents created AADL/BLESS to SPARK Ada translation schemes (4.1), proposed port communication (4.2) and discuss design of automatic translator, which can be created based on translation schemes (4.3).

4.1 AADL/BLESS to SPARK Ada mapping

Mapping of AADL models to SPARK Ada 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. Mapping of BLESS assertions was created in consultation with Brian Larson (BLESS creator).

[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 defines fundamental data types for AADL. Its definition is shown in 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*

Target language for this thesis is SPARK 2005. SPARK 2014 tools and especially multitasking capabilities were not ready, during the time when this thesis was written. Types: `Float`, `Character` and `String` are also not part of this thesis, because of verification tools limitation. Thus, only `Integer`, `Enumeration`, `Boolean` and **Record** types are taken into account in mappings.

Each type is translated into simple type definition and protected type. Then it can be used in multitasking 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. Protected objects can be also removed if they are not needed. Default value for priority, for each generated type is 10. It can be changed during development phase according to implementation details. Types: `Integer`, `Boolean` and `Natural` are already defined in SPARK Ada, thus only protected objects are generated for them. AADL `Base_Types` mapping to SPARK 2005 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;

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;

Continued on next page

Table 4.1 – continued from previous page

AADL	SPARK Ada

`data Type_With_Range`
`  properties`
`  Data_Model::`
`  Data_Representation =>`
`  Integer;`
`  Data_Model::Base_Type => (`
`  classifier (Base_Types::`
`  Unsigned_16));`
`  Data_Model::Integer_Range => 0`
`  .. 1000;`
`end Type_With_Range;`
	`type Type_With_Range is new Integer range 0 .. 1000;`
	
	`  protected type Type_With_Range_Store`
	`  is`
	`  pragma Priority (10);`
	
	`  function Get return Type_With_Range;`
	`  --# global in Type_With_Range_Store;`
	
	`  procedure Put(X : in Type_With_Range);`
	`  --# global out Type_With_Range_Store;`
	`  --# derives Type_With_Range_Store from X;`
	`  private`
	`  TheStoredData : Type_With_Range := 0;`
	`  end Unsigned_16_Store;`
	
	`  protected body Type_With_Range_Store is`
	`  function Get return Type_With_Range`
	`  --# global in TheStoredData;`
	`  is`
	`  begin`
	`  return TheStoredData;`
	`  end Get;`
	
	`  procedure Put(X : in Type_With_Range)`
	`  --# global out TheStoredData;`
	`  --# derives TheStoredData from X;`
	`  is`
	`  begin`
	`  TheStoredData := X;`
	`  end Put;`
	`  end Type_With_Range_Store;`

Type range is defined using AADL properties: `Data_Model::Number_Representation`, `Source_Data_Size` and `Data_Model::Integer_Range`. When `Data_Model::Integer_Range` property is not specified, then range is calculated. In case of `Integer` representation range starts from negative value, for `Unsigned` - from 0. Maximum value for `Integer` is calculated using the formula 4.1.

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

The minimum value formula for Integer (4.2) and maximum value for `Unsigned` (4.3) use similar strategy.

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

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

Mapping for enumeration types, presented on table 4.2, is pretty straightforward. BLESS properties are ignored in translation. In addition to simple types, protected types are generated.

Continued on next page

Table 4.2 – continued from previous page

AADL	SPARK Ada

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 := Enum_Type'First;`
	`  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 next car model, it becomes not enough. In case when e.g. `Speed_Type` is not defined, there are two possible resolutions. 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 existing type in AADL, derived type (defined with `extends` keyword) or `Data_Model::Base_Type` property can be used. Translation to SPARK Ada is shown in table 4.3. There are two ways to define type based on some other type in SPARK Ada:

- subtype - it is compatible with its parent, in other words: parent type variable can be assigned to it, if its value is in the subtype range
- derived type - it is incompatible with its parent (parent type variable cannot be assigned to it), but inherits its primitive operations

Table 4.3: AADL types to SPARK mapping: Subtypes.

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

AADL array type can be defined using property `Data_Model::Data_Representation`. In addition to that, size for 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 Ada 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;
	
	protected type Some_Array_Store
	is
	  pragma Priority (10);
	
	  function Get(Ind : **in** Integer) **return**
	  Base_Types.Integer_32;
	  --# global in Some_Array_Store;
	
	  procedure Put(Ind : **in** Integer; Val : **in**
	  Base_Types.Integer_32);
	  --# global in out Some_Array_Store;
	  --# derives Some_Array_Store from
	  Some_Array_Store, Ind, Val;
	private
	  TheStoredData : Some_Array := Some_Array'(
	  others => 0);
	end Some_Array_Store;
	
	protected body Some_Array_Store
	is
	  function Get(Ind : **in** Integer) **return**
	  Base_Types.Integer_32
	  --# global in TheStoredData;
	  is
	  begin
	  return TheStoredData(Ind);
	  end Get;
	
	  procedure Put(Ind : **in** Integer; Val : **in**
	  Base_Types.Integer_32)
	  --# global in out TheStoredData;
	  --# derives TheStoredData from
	  TheStoredData, Ind, Val;
	  is
	  begin
	  TheStoredData(Ind) := Val;
	  end Put;
	end Some_Array_Store;

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 struct to SPARK Ada record 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;

Data types translations are created based on Brian Larson's AADL/BLESS models of PCA Pump. They are syntacticly verified with SPARK Examiner. During development of types mapping, SPARK Examiner was helpful also for detecting inconsistencies in AADL models. Eg. it detected redundancy in enumerators. Both `Alarm_Type` and `Warning_Type` contained `No_Alarm` enumerator, which was a bug. All enumerators, for all types have to be unique. Thus `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]. Additionally, the mapping contains SPARK 2005 contracts. Data types used by ports has to be defined earlier, to be visible. Moreover, for port communication, protected types are used, to enable concurrency.

Table 4.6: AADL to SPARK ports mapping.

AADL/BLESS	SPARK Ada

Port_Name :
  in data port Port_Type;
	-- spec (.ads):
	--# own protected Port_Name : Port_Type_Store(Priority => 10)
	
	procedure Receive_Port_Name;
	--# global out Port_Name;
	
	-- body (.adb):
	Port_Name : Port_Type_Store;
	
	procedure Receive_Port_Name
	is
	begin
	  -- TODO: implement receiving Port_Name value
	  -- e.g.:
	  -- Port_Name.Put(Some_Pkg.Get_Port_Name);
	end Receive_Port_Name;

Port_Name :
  out data port Port_Type;
	-- spec (.ads)
	--# own protected Port_Name : Port_Type_Store(Priority => 10)
	
	procedure Get_Port_Name(Port_Name_Out : **out** Port_Type);
	--# global in Port_Name;
	--# derives Port_Name_Out from Port_Name;
	
	-- body (.adb):
	Port_Name : Port_Type_Store;
	
	procedure Get_Port_Name(Port_Name_Out : **out** Port_Type)
	is
	begin
	  Port_Name_Out := Port_Name.Get;
	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)
	--# own protected Port_Name : Port_Type_Store(Priority => 10);
	
	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_Store;
	
	procedure Put_Port_Name (Port_Name_In : Port_Type)
	is
	begin
	  Port_Name.Put(Port_Name_In);
	end Put_Port_Name;

Port_Name :
  out event data port Port_Type;
	-- spec (.ads)
	--# own protected Port_Name : Port_Type_Store(Priority => 10);
	
	procedure Send_Port_Name;
	  --# global in Port_Name;
	
	-- body (.adb):
	Port_Name : Port_Type_Store;
	
	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;

4.1.3 Thread to task mapping

AADL Threads are mapped into SPARK Ada tasks according to table 4.7. Communication between threads is described in section 4.2.1.

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;`
 	
`  thread implementation Some_Thread.imp`
`  end Some_Thread.imp;`
`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 body Some_Pkg`
	`is`
	`  st : Some_Thread;`
	
	`  task body Some_Thread`
	`  is`
	`  begin`
	`  loop`
	`  -- implementation`
	`  end loop;`
	`  end Some_Thread;`
	`end Some_Pkg;`

4.1.4 Subprograms mapping

Mappings of subprograms is also straightforward. However, it is different than mapping proposed in "AADL Code Generation Annex" [SCD14]. It does not use **renames** clause, but mapping directly to subprogram specification and body. For now, body is empty, because behavior (implementation) is not captured. Subprogram mapping should be revised and consulted with AADL committee members, in order to understand their design decisions.

Table 4.8: *AADL subprograms to SPARK Ada subprograms 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);
	
	procedure sp(e : **in** T; s : **out** T) **is**
	begin
	  --# implementation
	end sp;

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 (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. Thus, it will require to expose feature group internal variable as public. It is definitely not good solution. Thus, feature group is translated with prefix `Feature_Group_Name_*`. Feature group mapping is presented in section 4.1.6, in listings 4.5 and 4.6.

4.1.6 AADL package to SPARK Ada package mapping

On listing 4.5, there is sample AADL package with system. It contains all types of ports described in section 4.1.2 and one feature group with two ports as example of feature group mapping.

```
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*

For now, only single process SPARK Ada application is considered. Thus, ports are exposed only on system level. Communication between threads in process will be realized by protected objects and only SPARK annotations and data types will be needed as described in section 4.1.3. 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 AADL property set to SPARK Ada package mapping

There is no equivalent construct for AADL property set in SPARK Ada. Thus property set is mapped to SPARK Ada package. In this thesis, only properties of type **constant** `aadlinteger` are considered. There are issues with using non-constant types in SPARK Ada package (e.g. when using them in some type definition). Table 4.9 shows sample property set mapping to SPARK Ada package.

Table 4.9: *AADL property set to SPARK Ada package mapping.*

AADL	SPARK Ada

property set Some_Properties is
  Some_Property1 : constant **aadlinteger** => 10;
  Some_Property2 : constant **aadlinteger** => 27
  applies to (all);
  Some_Property3 : constant **aadlinteger** =>
  Some_Properties::Some_Property1 **applies to** (
  all);
end Some_Properties;
	package Some_Properties
	is
	  Some_Property1 : **constant** Integer := 10;
	  Some_Property2 : **constant** Integer := 27;
	  Some_Property3 : **constant** Integer :=
	  Some_Property1;
	end Some_Properties;

In AADL, all declarations must have an `applies to` clause. It is ignored in resulted SPARK Ada translation. However, it can be used in the future e.g. for automatic generation of **with** clauses.

4.1.8 BLESS mapping

In cooperation with Brian Larson, translations for BLESS assertions, invariant, pre- and postconditions were created. Table 4.10 presents their mapping to SPARK Ada. Generated (translated) code may not be complete. Then developer's effort to implement missing parts

will be required. E.g. when assertion is specified in AADL/BLESS model, but not defined, it has to be implemented.

Table 4.10: 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 4.10 – 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;

4.2 Port-based communication

Communication between AADL components is realized by ports. AADL ports can be declared in subprograms, threads, processes, systems and other entities. In this section communication between threads in single-process SPARK Ada application (4.2.1) and concept of communication between two systems (4.2.2) are presented.

4.2.1 Threads communication

Example of communication between threads, in single process is depicted in the figure 4.2.1. There are two threads (`some_thread` and `other_thread`) in one process. AADL model and its translation to SPARK Ada is presented in the table 4.11. Connection between threads has to be specified in process implementation. Based on mappings from section 4.1, protected object is defined, but subprograms are skipped, because communication takes place only internally. The result of translation consists of two tasks and private global protected object, which enable communication between them. Additionally, both tasks has global annotation (one with **out** mode, other with **in** mode), which announce use of protected object in their bodies.

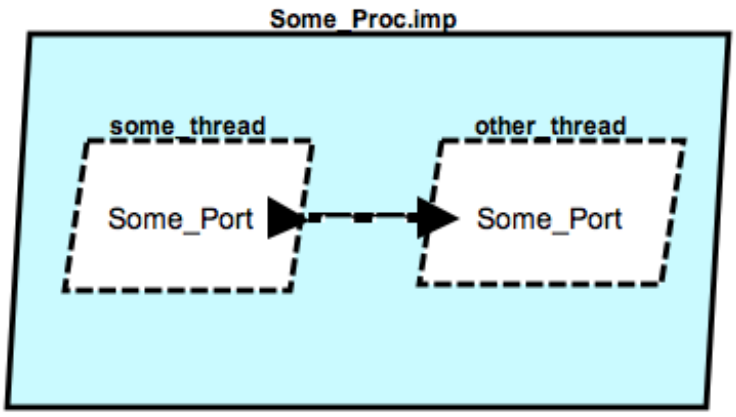

Figure 4.1: *Example of port communication between threads*

Threads can be also placed in different packages. The same example of two threads within one process, but in different packages is presented in table 4.12. In this case, subprograms present in mapping table, in section 4.2 are also present in resulted translation. Moreover, body of procedure `Receive_Some_Port` is implemented as a result of defined connection between threads in the process implementation, in AADL model.

Table 4.11: *AADL threads communication to SPARK Ada tasks communication translation*

AADL/BLESS	SPARK Ada

package Some_Pkg
public
with Base_Types;
 	
  process Some_Proc
  end Some_Proc;
 	
  process implementation Some_Proc.imp
  subcomponents
  some_thread: **thread** Some_Thread.imp;
  other_thread: **thread** Other_Thread.imp;
  connections
  connection: **port** some_thread.Some_Port ->
  other_thread.Some_Port;
  end Some_Proc.imp;
 	
  thread Some_Thread
  features
  Some_Port : **out data port** Base_Types::
  Integer;
  end Some_Thread;
 	
  thread implementation Some_Thread.imp
  end Some_Thread.imp;
 	
  thread Other_Thread
  features
  Some_Port : **in data port** Base_Types::Integer
  ;
  end Other_Thread;
 	
  thread implementation Other_Thread.imp
  end Other_Thread.imp;
 	
end Some_Pkg;
	with Base_Types;
	--# inherit Base_Types;
	package Some_Pkg
	--# own task st : Some_Thread;
	--# task ot : Other_Thread;
	--# protected Some_Port : Base_Types.
	  Integer_Store (Priority => 10);
	is
	
	private
	  task type Some_Thread
	  --# global out Some_Port;
	  is
	  pragma Priority (10);
	  end Some_Thread;
	
	  task type Other_Thread
	  --# global in Some_Port;
	  is
	  pragma Priority (10);
	  end Other_Thread;
	end Some_Pkg;
	
	package body Some_Pkg
	is
	  st : Some_Thread;
	  ot : Other_Thread;
	  Some_Port : Base_Types.Integer_Store;
	
	  task body Some_Thread
	  is
	  begin
	  loop
	  -- implementation
	  end loop;
	  end Some_Thread;
	
	  task body Other_Thread
	  is
	  begin
	  loop
	  -- implementation
	  end loop;
	  end Other_Thread;
	end Some_Pkg;

Table 4.12: AADL threads communication to SPARK Ada tasks communication translation (multiple packages)

AADL/BLESS

```
package Pkg1
public
with Base_Types, Pkg2;

  process Some_Proc
  end Some_Proc;

  process implementation
     Some_Proc.imp
  subcomponents
    some_thread: thread
     Some_Thread.imp;
    other_thread: thread Pkg2::
     Other_Thread.imp;
  connections
    connection: port some_thread.
     Some_Port -> other_thread.
     Some_Port;
  end Some_Proc.imp;

  thread Some_Thread
    features
      Some_Port : out data port
     Base_Types::Integer;
  end Some_Thread;

  thread implementation
     Some_Thread.imp
  end Some_Thread.imp;
```

SPARK Ada

```
with Base_Types;
--# inherit Base_Types;
package Pkg1
--# own task st : Some_Thread;
--#     protected Some_Port : Base_Types.Integer_Store (Priority =>
     10);
is
   procedure Get_Some_Port(Some_Port_Out : out Integer);
   --# global in Some_Port;
   --# derives Some_Port_Out from Some_Port;

private
   task type Some_Thread
   --# global out Some_Port;
   is
      pragma Priority (10);
   end Some_Thread;
end Pkg1;

package body Pkg1
is
   st : Some_Thread;
   Some_Port : Base_Types.Integer_Store;

   procedure Get_Some_Port(Some_Port_Out : out Integer)
   is
   begin
      Some_Port_Out := Some_Port.Get;
   end Get_Some_Port;

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

end Pkg1;
```

Continued on next page

Table 4.12 – continued from previous page

AADL/BLESS	SPARK Ada
package Pkg2 public with Base_Types; thread Other_Thread features Some_Port : in data port Base_Types::Integer; end Other_Thread; thread implementation Other_Thread.imp end Other_Thread.imp; end Pkg2;	with Base_Types; with Pkg1; --# inherit Base_Types, --# Pkg1; package Pkg2 --# own task ot : Other_Thread; --# protected Some_Port : Base_Types.Integer_Store (Priority => 10); is procedure Receive_Some_Port; --# global out Some_Port; --# in Pkg1.Some_Port; private task type Other_Thread --# global in Some_Port; is pragma Priority (10); end Other_Thread; end Pkg2; package body Pkg2 is ot : Other_Thread; Some_Port : Base_Types.Integer_Store; procedure Receive_Some_Port is Temp : Integer; begin Pkg1.Get_Some_Port(Temp); Some_Port.Put(Temp); end Receive_Some_Port; task body Other_Thread is begin loop -- implementation end loop; end Other_Thread; end Pkg2;

In the given example, communication is one way: from `Pkg1` package to `Pkg2` package. Thus, `Pkg1` package does not need to know that `Pkg2` package exists. In other words: it does not need to "with" it. However, if two way communication is needed (between `Pkg1` to `Pkg2`), then `Pkg1` package has to "with" `Pkg2` package. It is not a case in first example, where communication between threads take place in the same package. Modified model of second

example, with communication from `Pkg2` to `Pkg1`, is depicted in the figure 4.2.1 and presented in listing 4.7.

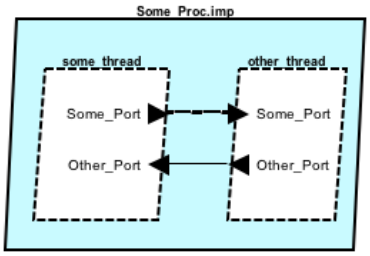

Figure 4.2: *Example of two way port communication between threads in different packages*

```
package Pkg1TwoWay
public
with Base_Types, Pkg2TwoWay;

  process Some_Proc
  end Some_Proc;

  process implementation Some_Proc.imp
  subcomponents
    some_thread: thread Some_Thread.imp;
    other_thread: thread Pkg2TwoWay::Other_Thread.imp;
  connections
    connection: port some_thread.Some_Port -> other_thread.Some_Port;
    connection2: port some_thread.Other_Port -> other_thread.Other_Port;
  end Some_Proc.imp;

  thread Some_Thread
    features
      Some_Port : out data port Base_Types::Integer;
      Other_Port : in data port Base_Types::Integer;
  end Some_Thread;

  thread implementation Some_Thread.imp
  end Some_Thread.imp;
end Pkg1TwoWay;

package Pkg2TwoWay
public
with Base_Types;
  thread Other_Thread
    features
      Some_Port : in data port Base_Types::Integer;
      Other_Port : out data port Base_Types::Integer;
  end Other_Thread;

  thread implementation Other_Thread.imp
  end Other_Thread.imp;
end Pkg2TwoWay;
```

Listing 4.7: *AADL model of two way port communication threads in different packages*

This model, translated to SPARK Ada is presented in listing 4.8. It will not compile. GNAT compiler returns `circular unit dependency` error. Additionally verification with SPARK Examiner returns error: `Semantic Error 135 - The package Pkg2TwoWay is undeclared or not visible, or there is a circularity in the list of inherited packages.` Now, the problem is that two-way communication is allowed in AADL, but not in SPARK, nor even in Ada. This require further investigation, which is omitted in this thesis.

```
with Base_Types;
with Pkg2TwoWay;
--# inherit Base_Types,
--#         Pkg2TwoWay;
package Pkg1TwoWay
--# own task st : Some_Thread;
--#     protected Some_Port : Base_Types.Integer_Store (Priority => 10);
--#     protected Other_Port : Base_Types.Integer_Store (Priority => 10);
is
   procedure Get_Some_Port(Some_Port_Out : out Integer);
   --# global in Some_Port;
   --# derives Some_Port_Out from Some_Port;

   procedure Receive_Other_Port;
   --# global out Other_Port;
   --#        in Pkg2TwoWay.Other_Port;

private
   task type Some_Thread
   --# global out Some_Port;
   is
      pragma Priority (10);
   end Some_Thread;
end Pkg1TwoWay;

package body Pkg1TwoWay
is
   st : Some_Thread;
   Some_Port : Base_Types.Integer_Store;
   Other_Port : Base_Types.Integer_Store;

   procedure Get_Some_Port(Some_Port_Out : out Integer)
   is
   begin
      Some_Port_Out := Some_Port.Get;
   end Get_Some_Port;

   procedure Receive_Other_Port
   is
      Temp : Integer;
   begin
      Pkg2TwoWay.Get_Other_Port(Temp);
      Other_Port.Put(Temp);
   end Receive_Other_Port;

   task body Some_Thread
   is
   begin
      loop
```

```
         -- implementation
         null;
      end loop;
   end Some_Thread;

end Pkg1TwoWay;

with Base_Types;
with Pkg1TwoWay;
--# inherit Base_Types,
--#         Pkg1TwoWay;
package Pkg2TwoWay
--# own task ot : Other_Thread;
--#     protected Some_Port : Base_Types.Integer_Store (Priority => 10);
--#     protected Other_Port : Base_Types.Integer_Store (Priority => 10);
is
   procedure Receive_Some_Port;
   --# global out Some_Port;
   --#        in Pkg1TwoWay.Some_Port;

   procedure Get_Other_Port(Other_Port_Out : out Integer);
   --# global in Other_Port;
   --# derives Other_Port_Out from Other_Port;

private
   task type Other_Thread
   --# global in Some_Port;
   is
      pragma Priority (10);
   end Other_Thread;
end Pkg2TwoWay;

package body Pkg2TwoWay
is
   ot : Other_Thread;
   Some_Port : Base_Types.Integer_Store;
   Other_Port : Base_Types.Integer_Store;

   procedure Receive_Some_Port
   is
      Temp : Integer;
   begin
      Pkg1TwoWay.Get_Some_Port(Temp);
      Some_Port.Put(Temp);
   end Receive_Some_Port;

   procedure Get_Other_Port(Other_Port_Out : out Integer)
   is
   begin
      Other_Port_Out := Other_Port.Get;
   end Get_Other_Port;

   task body Other_Thread
   is
   begin
      loop
         -- implementation
         null;
      end loop;
   end Other_Thread;

end Pkg2TwoWay;
```

Listing 4.8: *Two way port communication translated to SPARK Ada*

4.2.2 Systems communication

This section is a concept, how communication between different systems can look like. AADL system consists of process(es) and process consists of threads. Ports would be exposed by package if they are specified in system entity. Communication between two systems cab be described by another system. Figure 4.2.2 presents communication between two systems: panel and pump. AADL model of this system comprises 3 packages: `Main`, `Panel` and `Pump`. They are presented in listing 4.9. `Panel` package has one thread `Panel_Thread` with two **out** ports: `event port` and `event data port`. Both ports are exposed by process `panel_process` and then by system `panel`. `Pump` package has similar structure, but two **in** ports. Both are also exposed by process (`pump_process`) and system (`pump`). Connection between these two packages are defined in `Main` package.

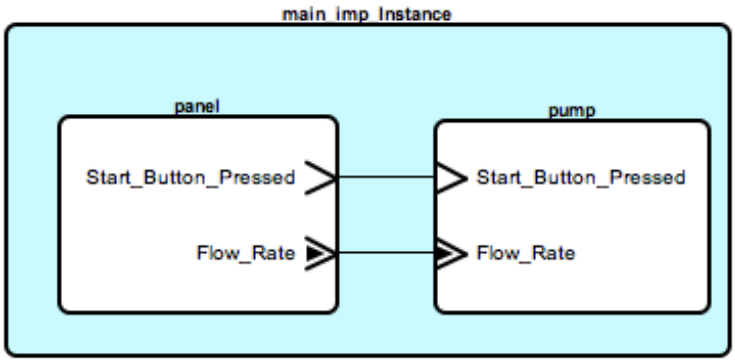

Figure 4.3: *Example of port communication between systems*

```
package Panel
public
with Base_Types;

  thread Panel_Thread
  features
    Start_Button_Pressed: out event port;
    Flow_Rate: out event data port Base_Types::Integer;
  end Panel_Thread;

  thread implementation Panel_Thread.imp
  end Panel_Thread.imp;

  process panel_process
```

```
    features
      Start_Button_Pressed: out event port;
      Flow_Rate: out event data port Base_Types::Integer;
  end panel_process;

  process implementation panel_process.imp
    subcomponents
      panel_thread: thread Panel_Thread.imp;
    connections
      sbp: port panel_thread.Start_Button_Pressed->Start_Button_Pressed;
      fr: port panel_thread.Flow_Rate->Flow_Rate;
  end panel_process.imp;

  system panel
    features
      Start_Button_Pressed: out event port;
      Flow_Rate: out event data port Base_Types::Integer;
  end panel;

  system implementation panel.imp
    subcomponents
      panel_process: process panel_process.imp;
    connections
      sbp: port panel_process.Start_Button_Pressed->Start_Button_Pressed;
      fr: port panel_process.Flow_Rate->Flow_Rate;
  end panel.imp;

end Panel;

package Pump
public
with Base_Types;

  thread Rate_Controller
    features
      Start_Button_Pressed: in event port;
      Flow_Rate: in event data port Base_Types::Integer;
  end Rate_Controller;

  thread implementation Rate_Controller.imp
  end Rate_Controller.imp;

  process pump_process
    features
      Start_Button_Pressed : in event port;
      Flow_Rate: in event data port Base_Types::Integer;
  end pump_process;

  process implementation pump_process.imp
    subcomponents
      Rate_Controller: thread Rate_Controller.imp;
    connections
      sbp: port Start_Button_Pressed->Rate_Controller.Start_Button_Pressed;
      fr: port Flow_Rate->Rate_Controller.Flow_Rate;
  end pump_process.imp;

  system pump
    features
      Start_Button_Pressed : in event port;
      Flow_Rate: in event data port Base_Types::Integer;
  end pump;

  system implementation pump.imp
    subcomponents
      pump_process : process pump_process.imp;
```

```
    connections
      sbp: port Start_Button_Pressed->pump_process.Start_Button_Pressed;
      fr: port Flow_Rate->pump_process.Flow_Rate;
  end pump.imp;
end Pump;

package Main
public
with Pump,
  Panel;

  system main
  end main;

  system implementation main.imp
    subcomponents
      panel: system Panel::panel.imp;
      pump: system Pump::pump.imp;
    connections
      sbp2sbp: port panel.Start_Button_Pressed->pump.Start_Button_Pressed;
      fr2fr: port panel.Flow_Rate->pump.Flow_Rate;
  end main.imp;
end Main;
```

Listing 4.9: *AADL model of port communication between systems*

Based on mappings from section 4.1, conforming SPARK Ada code is presented in listing 4.10. There are two packages: `Panel` and `Pump`. `Main` package is omitted. Both contain procedures representing ports interfaces, according to ports mapping from section 4.1.2. Additionally, both consists of empty thread declarations and bodies, which conforms to translations from section 4.1.3. However, in this case, both packages will work in different systems, thus different processes. To enable communication between different systems, deployment methodology and the middle-ware layer has to be created. It will allow not only for system to system communication, but also for communication with devices. It is omitted in this thesis.

```
with Pump;
with Base_Types;
--# inherit Pump,
--#         Base_Types;
package Panel
--# own task pt : Panel_Thread;
--#     protected Flow_Rate : Base_Types.Integer_Store (Priority => 10);
is
    procedure Send_Start_Button_Pressed;

    procedure Send_Flow_Rate;
    --# global in Flow_Rate;
    --#        out Pump.Flow_Rate;
```

```
private
    task type Panel_Thread
    --# global in out Flow_Rate;
    is
        pragma Priority (10);
    end Panel_Thread;

end Panel;

package body Panel
is
    pt : Panel_Thread;
    Flow_Rate : Base_Types.Integer_Store;

    procedure Send_Start_Button_Pressed
    is
    begin
        Pump.Put_Start_Button_Pressed;
    end Send_Start_Button_Pressed;

    procedure Send_Flow_Rate
    is
        Flow_Rate_Temp : Integer;
    begin
        Flow_Rate_Temp := Flow_Rate.Get;
        Pump.Put_Flow_Rate(Flow_Rate_Temp);
    end Send_Flow_Rate;

    task body Panel_Thread
    is
    begin
      loop
         -- implementation
      end loop;
    end Panel_Thread;
end Panel;

with Base_Types;
--# inherit Base_Types;
package Pump
--# own task rc : Rate_Controller;
--#     protected Flow_Rate : Base_Types.Integer_Store (Priority => 10);
is
    procedure Put_Start_Button_Pressed;

    procedure Put_Flow_Rate(Flow_Rate_In : Integer);
    --# global out Flow_Rate;
    --# derives Flow_Rate from Flow_Rate_In;

private
    task type Rate_Controller
    --# global in out Flow_Rate;
    is
        pragma Priority (10);
    end Rate_Controller;
end Pump;

package body Pump
is
    rc : Rate_Controller;
    Flow_Rate : Base_Types.Integer_Store;

    procedure Put_Start_Button_Pressed
    is
    begin
```

```
        -- TODO: implement event handler
    end Put_Start_Button_Pressed;

    procedure Put_Flow_Rate(Flow_Rate_In : Integer)
    is
    begin
        Flow_Rate.Put(Flow_Rate_In);
    end Put_Flow_Rate;

    task body Rate_Controller
    is
    begin
      loop
         -- implementation
        end loop;
    end Rate_Controller;
end Pump;
```

Listing 4.10: *Port communication translated to SPARK Ada*

4.3 Automatic translator

The ultimate goal is to create translator, which performs translations described in 4.1 and 4.2 automatically. Automatic translator should enable translation of entire model and parts of the model. Initially, translator should support only subset of AADL entities: the system, process, thread, subprogram and port communication. The following functions should be supported:

- data types translation (as described in section 4.1.1)
- threads to tasks translation (as described in 4.1.3)
- single ports translation (based on section 4.1.2)
- subprogram to procedure/function translation (based on section 4.1.4)
- single package translation with system, which contains ports and feature groups (as described in section 4.1.6)
- property set mapping to SPARK Ada package (like in section 4.1.7)

Second step, would be to introduce BLESS support. Which means, add supported BLESS constructs described in section 4.1.8:

- assertions for threads
- pre- and postconditions for subprograms

Recommended way to create translator is to parse AADL models, create Abstract Syntax Tree (AST) and emit code using Visitor pattern. Parser and AST can be generated using ANTLR[4] (Another Tool for Language Recognition) and its grammar development environment ANTLRWorks[5]. ANTLR 4 (with ANTLRWorks 2) enable automatic AST creation and handle left recursion, which makes parser development much easier and faster. Another tool, Xtext[6] can be also used (instead of ANTLR) for parser and AST generator. For emitting code, StringTemplate[7] (template engine for generating code) can be used.

Development should be performed incrementally. From adding translation for the simplest constructs, like data types or single ports, to port communication and BLESS. First step, would be AADL grammar development. It is recommended to initially, specify only part of required AADL subset and then extend it incrementally. In order to do that, AADL Syntax Card[8] might be helpful. During translator development unit testing and Test Driven Development is recommended. Translation schemes can be used as input and expected output of particular test cases. It will help to ensure correctness of translator while working on new features support.

[4] http://www.antlr.org/
[5] http://tunnelvisionlabs.com/products/demo/antlrworks
[6] http://www.eclipse.org/Xtext/index.html
[7] http://www.stringtemplate.org/
[8] https://wiki.sei.cmu.edu/aadl/images/d/d2/AADL_V2.1_Syntax_Card.pdf

Chapter 5

PCA Pump Prototype Implementation

First step was to create mock (based on doc, aadl models and implemented PCA Pump).

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 BeagleBoard-XM

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

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.`

Running programs on BB (ch.3 from 721 paper)

To run SPARK Ada programs on BeagleBoard-xM GNAT cross-compiler was used. It compiles and links SPARK Ada 2005 single and multitasking programs without any problems. In case of SPARK Ada 2014 only single tasking programs are allowed. Tasking features are not yet in SPARK 2014, which is under development. After successful compilation and linking, there was no issues with running programs on BeagleBoard-xM. Moreover, their behaviour on Intel processor (PC or MacBook) is the same like on ARM device.

5.3 Implementation based on Requirements Document

In this thesis, only the operation module is implemented. Req doc don't specified all implementation details, like how basal rate should be infused.

The first step, was to check if implementation of PCA Pump specified in Requirements Document is possible. Especially whether it will work on BeagleBoard-xM device. 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`.

Pump internal implementation based on [Med10]. - basal dose deliver in increments -

easier to track delivered amount (page 14)

The pump has three tasks in total:

- main task (requesting boluses, displaying volume infused etc.)
- rate controller - to control the speed of infusion rate
- max drug per hour watcher - to control overinfusion

The main PCA Pump Package can be verified using SPARK tools. The Pump engine (motor) is separated entity. It has an interface allowing request infusion of 0.1 ml of drug.

PCA Pump prototype was developed based on Requirements Documentation created by Brian Larson, John Hatcliff and Partice Chalin [Lar14].

5.4 Code generation from AADL models

PCA Pump prototype was mocked using translations schemas from chapter 4.

The original models were simplified and truncated for the purpose of this thesis. Finally only `PCA_Operation` module with 3 threads (`Max_Drug_Per_Hour_Watcher`, `Rate_Controller`, `Patient_Bolus_Checker`), types definitions (`Base_Types`, `PCA_Types`, `ICE_Types`, `Bless_Types`) and property set `PCA_Properties` were used as the source for code translation.

[code listings of aadl model]

Skeleton code 'generated' from simplified AADL models. Then implemented.

Show generated code.

5.5 Implementation for generated code

Mocked code was extended (e.g. to Prescription_Store Set/Get methods for record fields were added).

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

The strategy for Software Verification using SPARK tools is as follows. First, Examiner is run. It generates and discharge some Verification Conditions (VCs). Next, SPARKSimp. It runs Simplifier to simplify and discharge some (or all) VCs, which were not discharged by Examiner. Along with Simplifier, ZombieScope runs to find dead paths. SPARKSimp can also run Victor (and it is in this case) to discharge VCs not discharged by Examiner nor Simplifier. To get summary of results, POGS report is generated. In case, when not all Verification Conditions are discharged, analysis continues with Bakar Kiasan. After fixes made with Kiasan help, Examiner and SPARKSimp tools are run again to confirm correctness.

An overview of verification contracts and annotations can be found in chapter 12 of Barnes' book [Bar13].

6.1 Verification of implemented prototype

During PCA Pump prototype implementation, syntax was regularly check with SPARK Examiner.

First run

```
----- BEGIN PROOF SUMMARY -----
VCs discharged:        556
VCs false:               2
VCs undischarged:       37
Warnings: No
Errors:   YES
----- END PROOF SUMMARY -----
```

Listing 6.1: *POGS report for PCA Pump prototype*

Two false VCs applies to `Status_Store` protected type's subprograms: `Put` and `Get`. Error? The same implementation like for other enumeration types.

POGS report is not included here, because it has almost 3000 lines.

There is 37 undischarged VCs. As mentioned in chapter 2.6.9, Bakar Kiasan does not support Ravenscar profile. Thus, to be able to analyze monitoring dosed amount of drug, separate module was created. Verification process of this module is described in section 6.2.

6.2 Monitoring dosed amount

Verification of module responsible for tracking dosed amount of drug. Isolated to verify, because of Ravenscar limitations. Sequential.

Code:

```
package Pca_Pump
--# own Dosed;
--#     Dose_Volume;
--# initializes Dosed,
--#             Dose_Volume;
is
    subtype Integer_Array_Index is Integer range 1 .. 60*60;
    type Integer_Array is array (Integer_Array_Index) of Integer;

    procedure Increase_Dosed;
```

```
    --# global in out Dosed;
    --#        in Dose_Volume;
    --# derives Dosed from Dosed, Dose_Volume;

    function Read_Dosed return Integer;
    --# global in Dosed;

    procedure Move_Dosed;
    --# global in out Dosed;
    --# derives Dosed from Dosed;

end Pca_Pump;
```

Listing 6.2: *Dose monitor module specification*

```
package body Pca_Pump
is
    Dosed : Integer_Array := Integer_Array'(others => 0);
    Dose_Volume : Integer := 1;

    procedure Increase_Dosed
    is
    begin
        Dosed(Integer_Array_Index'Last) := Dosed(Integer_Array_Index'Last) + Dose_Volume;
    end Increase_Dosed;

    function Read_Dosed return Integer
    is
        Result : Integer := 0;
    begin
        for I in Integer_Array_Index loop
            --# assert I > 1 -> Result >= Dosed(I-1);
            Result := Result + Dosed(I);
        end loop;
        return Result;
    end Read_Dosed;

    procedure Move_Dosed
    is
    begin
```

```
        for I in Integer_Array_Index range 1 .. Integer_Array_Index'Last-1 loop
            --# assert I > 1 -> Dosed(I-1) = Dosed(I);
            Dosed(I) := Dosed(I+1);
        end loop;
        Dosed(Integer_Array_Index'Last) := 0;
    end Move_Dosed;

end Pca_Pump;
```

Listing 6.3: *Dose monitor module body*

Verification with Examiner, Simplifier, ZombieScope, Victor, POGS and then Bakar Kiasan. SPARKSimp run Simplifier and Victor with command `sparksimp -victor`.

Examiner: No errors or warnings

[REMOVE] SPARKSimp:

```
gnatspark sparksimp -P/Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/Pca_Verification/
    pca_verification.gpr
sparksimp -victor
SPARKSimp GPL 2012
Copyright (C) 2012 Altran Praxis Limited, Bath, U.K.
Simplifier  binary located at: /Users/jj/Sireum/apps/spark/2012/bin/spadesimp
ZombieScope binary located at: /Users/jj/Sireum/apps/spark/2012/bin/zombiescope
Victor binary located at: /Users/jj/Sireum/apps/spark/2012/bin/victor

Files to be simplified are:
/Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/Pca_Verification/pca_pump/increase_dosed.dpc,
    1474 bytes
/Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/Pca_Verification/pca_pump/increase_dosed.vcg,
    1457 bytes
/Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/Pca_Verification/pca_pump/move_dosed.dpc, 6072
     bytes
/Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/Pca_Verification/pca_pump/move_dosed.vcg, 9612
     bytes
/Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/Pca_Verification/pca_pump/read_dosed.dpc, 3089
     bytes
/Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/Pca_Verification/pca_pump/read_dosed.vcg, 5189
     bytes
```

```
 6 files require processing


Job-ID Status     Filename
====== ======     ========
     1 Started - ZOMBIESCOPE - /Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/
     Pca_Verification/pca_pump/increase_dosed.dpc
     1 Finished    0: 0: 0.10
     2 Started - SIMPLIFY - /Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/Pca_Verification/
     pca_pump/increase_dosed.vcg
     2 Finished    0: 0: 0.12
     3 Started - ZOMBIESCOPE - /Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/
     Pca_Verification/pca_pump/move_dosed.dpc
     3 Finished    0: 0: 0.17
     4 Started - SIMPLIFY - /Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/Pca_Verification/
     pca_pump/move_dosed.vcg
     4 Finished    0: 0: 0.18
     5 Started - ZOMBIESCOPE - /Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/
     Pca_Verification/pca_pump/read_dosed.dpc
     5 Finished    0: 0: 0.18
     6 Started - SIMPLIFY - /Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/Pca_Verification/
     pca_pump/read_dosed.vcg
     6 Finished    0: 0: 0.17
     7 Started - VICTOR - /Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/Pca_Verification/
     pca_pump/increase_dosed.vcg
     7 Finished    0: 0: 0.08
     8 Started - VICTOR - /Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/Pca_Verification/
     pca_pump/move_dosed.vcg
     8 Finished    0: 0: 0.02
     9 Started - VICTOR - /Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/Pca_Verification/
     pca_pump/read_dosed.vcg
     9 Finished    0: 0: 0.36
Total elapsed time:    0: 0: 1.39
[2014-07-01 14:45:00] process terminated successfully (elapsed time: 01.53s)
```

Listing 6.4: *SPARKSimp output*

[truncate?]POGS Report:

```
------------------------------------------------------------------------------
                        Semantic Analysis Summary
```

```
                                POGS GPL 2012
            Copyright (C) 2012 Altran Praxis Limited, Bath, U.K.
------------------------------------------------------------------------------

Summary of:

Verification Condition files (.vcg)
Simplified Verification Condition files (.siv)
Victor result files (.vct)
Riposte result files (.rsm)
Proof Logs (.plg)
Dead Path Conjecture files (.dpc)
Summary Dead Path files (.sdp)

"status" column keys:
    1st character:
        '-' - No VC
        'S' - No SIV
        'U' - Undischarged
        'E' - Proved by Examiner
        'I' - Proved by Simplifier by Inference
        'X' - Proved by Simplifier by Contradiction
        'P' - Proved by Simplifier using User Defined Proof Rules
        'V' - Proved by Victor
        'O' - Proved by Riposte
        'C' - Proved by Checker
        'R' - Proved by Review
        'F' - VC is False
    2nd character:
        '-' - No DPC
        'S' - No SDP
        'U' - Unchecked
        'D' - Dead path
        'L' - Live path

in the directory:
/Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/Pca_Verification

Summary produced: 01-JUL-2014 14:43:18.04
```

```
File /Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/Pca_Verification/pca_pump/increase_dosed.
    vcg
procedure Pca_Pump.Increase_Dosed

VCs generated 01-JUL-2014 14:42:26

VCs simplified 01-JUL-2014 14:43:04

File /Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/Pca_Verification/pca_pump/increase_dosed.
    dpc
DPCs generated 01-JUL-2014 14:42:26

DPC ZombieScoped 01-JUL-2014  14:43:0

VCs for procedure_increase_dosed :
 ------------------------------------------------------------------------------
| #   | From  | To                  | Proved By          | Dead Path | Status |
|------------------------------------------------------------------------------
| 1   | start | rtc check @ 9       | Undischarged       | Unchecked |   UU   |
| 2   | start |    assert @ finish  | Examiner           | Live      |   EL   |
 ------------------------------------------------------------------------------


File /Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/Pca_Verification/pca_pump/move_dosed.vcg
procedure Pca_Pump.Move_Dosed

VCs generated 01-JUL-2014 14:42:26

VCs simplified 01-JUL-2014 14:43:04

File /Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/Pca_Verification/pca_pump/move_dosed.dpc
DPCs generated 01-JUL-2014 14:42:26

DPC ZombieScoped 01-JUL-2014  14:43:0

VCs for procedure_move_dosed :
 ------------------------------------------------------------------------------
| #   | From  | To                  | Proved By          | Dead Path | Status |
```

```
|-------------------------------------------------------------------------
| 1   | start | rtc check @ 26      | Inference          | Unchecked |   IU   |
| 2   | start | rtc check @ 26      | Inference          | Unchecked |   IU   |
| 3   | start |    assert @ 27      | Inference          | Live      |   IL   |
| 4   | 27    |    assert @ 27      | Inference          | Live      |   IL   |
| 5   | 27    | rtc check @ 28      | Inference          | Unchecked |   IU   |
| 6   | start | rtc check @ 30      | Inference          | Unchecked |   IU   |
| 7   | 27    | rtc check @ 30      | Inference          | Unchecked |   IU   |
| 8   | start |    assert @ finish  | Examiner           | Dead      |   ED   |
| 9   | 27    |    assert @ finish  | Examiner           | Live      |   EL   |
 -------------------------------------------------------------------------


File /Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/Pca_Verification/pca_pump/read_dosed.vcg
function Pca_Pump.Read_Dosed

VCs generated 01-JUL-2014 14:42:26

VCs simplified 01-JUL-2014 14:43:05

File /Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/Pca_Verification/pca_pump/read_dosed.dpc
DPCs generated 01-JUL-2014 14:42:26

DPC ZombieScoped 01-JUL-2014  14:43:0

VCs for function_read_dosed :
 -------------------------------------------------------------------------
| #   | From  | To                  | Proved By          | Dead Path | Status |
|-------------------------------------------------------------------------
| 1   | start |    assert @ 17      | Inference          | Live      |   IL   |
| 2   | 17    |    assert @ 17      | Undischarged       | Live      |   UL   |
| 3   | 17    | rtc check @ 18      | Undischarged       | Unchecked |   UU   |
| 4   | 17    |    assert @ finish  | Inference          | Live      |   IL   |
 -------------------------------------------------------------------------


=========================================================================
Summary:
```

```
The following subprograms have undischarged VCs (excluding those proved false):

   1  /Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/Pca_Verification/pca_pump/increase_dosed
     .vcg
   2  /Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/Pca_Verification/pca_pump/read_dosed.vcg

Proof strategies used by subprograms
------------------------------------------------------------------------
Total subprograms with at least one VC proved by examiner:              2
Total subprograms with at least one VC proved by simplifier:            2
Total subprograms with at least one VC proved by contradiction:         0
Total subprograms with at least one VC proved with user proof rule:     0
Total subprograms with at least one VC proved by Victor:                0
Total subprograms with at least one VC proved by Riposte:               0
Total subprograms with at least one VC proved using checker:            0
Total subprograms with at least one VC discharged by review:            0

Maximum extent of strategies used for fully proved subprograms:
------------------------------------------------------------------------
Total subprograms with proof completed by examiner:                     0
Total subprograms with proof completed by simplifier:                   1
Total subprograms with proof completed with user defined rules:         0
Total subprograms with proof completed by Victor:                       0
Total subprograms with proof completed by Riposte:                      0
Total subprograms with proof completed by checker:                      0
Total subprograms with VCs discharged by review:                        0

Overall subprogram summary:
------------------------------------------------------------------------
Total subprograms fully proved:                                         1
Total subprograms with at least one undischarged VC:                    2  <<<
Total subprograms with at least one false VC:                           0
                                                                      -----
Total subprograms for which VCs have been generated:                    3



ZombieScope Summary:
------------------------------------------------------------------------
Total subprograms for which DPCs have been generated:                   3
```

```
Total number subprograms with dead paths found:                          1
Total number of dead paths found:                                        1



VC summary:
-------------------------------------------------------------------------
Note: (User) denotes where the Simplifier has proved VCs using one or
      more user-defined proof rules.

Total VCs by type:
------------------
                    Total   Examiner Simplifier    Undisc.
Assert/Post             8          3          4          1
Precondition            0          0          0          0
Check stmnt.            0          0          0          0
Runtime check           7          0          5          2
Refinem. VCs            0          0          0          0
Inherit. VCs            0          0          0          0
=========================================================
Totals:                15          3          9          3 <<<
%Totals:                          20%        60%        20%

==================== End of Semantic Analysis Summary ======================
```

Listing 6.5: *POGS report*

pca-pump-verification-step1.png

problem: Integer'First = Integer'Last = 1 :O

solution: added standard.ads:

```
e Standard is

pe Integer is range -2**31 .. 2**31-1;

andard;
```

pca-pump-verification-step2.png

Introduce type Drug_Volume Change `Integer_Array` to `Doses_Array` because it is not array of

integers anymore.

Result: no lower overflow in `Increase_Dosed`. Only upper overflow left.

pca-pump-verification-step3.png

Add contract to `Increase_Dosed` `--# pre Read_Dosed(Dosed) <= Drug_Volume'Last - Dose_Volume;` Examiner Error: `Semantic Error 1 - The identifier Read_Dosed is either undeclared or not visible at this point`.

Moved `Read_Dosed` to be before `Increase_Dosed`. Examiner Error: `pca_pump.ads:19:51: Semantic Error 35 - Binary operator is not declared for types Drug_Volume and Dose_Volume__type.`

Declared `Dose_Volume` type in `--# own`: `--# Dose_Volume : Drug_Volume;`

Rerun Examiner and SPARKSimp: [truncate?]

```
------------------------------------------------------------------------------
                          Semantic Analysis Summary
                                POGS GPL 2012
            Copyright (C) 2012 Altran Praxis Limited, Bath, U.K.
------------------------------------------------------------------------------

Summary of:

Verification Condition files (.vcg)
Simplified Verification Condition files (.siv)
Victor result files (.vct)
Riposte result files (.rsm)
Proof Logs (.plg)
Dead Path Conjecture files (.dpc)
Summary Dead Path files (.sdp)

"status" column keys:
    1st character:
        '-' - No VC
        'S' - No SIV
        'U' - Undischarged
        'E' - Proved by Examiner
        'I' - Proved by Simplifier by Inference
        'X' - Proved by Simplifier by Contradiction
```

```
        'P' - Proved by Simplifier using User Defined Proof Rules
        'V' - Proved by Victor
        'O' - Proved by Riposte
        'C' - Proved by Checker
        'R' - Proved by Review
        'F' - VC is False
    2nd character:
        '-' - No DPC
        'S' - No SDP
        'U' - Unchecked
        'D' - Dead path
        'L' - Live path

in the directory:
/Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/Pca_Verification

Summary produced: 02-JUL-2014 13:09:29.38

File /Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/Pca_Verification/pca_pump/increase_dosed.
     vcg
procedure Pca_Pump.Increase_Dosed

VCs generated 02-JUL-2014 13:08:10

VCs simplified 02-JUL-2014 13:08:20

File /Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/Pca_Verification/pca_pump/increase_dosed.
     dpc
DPCs generated 02-JUL-2014 13:08:10

DPC ZombieScoped 02-JUL-2014  13:08:2

VCs for procedure_increase_dosed :
 ------------------------------------------------------------------------------
| #   | From  | To                  | Proved By          | Dead Path | Status |
|------------------------------------------------------------------------------
| 1   | start | rtc check @ 9       | Undischarged       | Unchecked |   UU   |
| 2   | start |    assert @ finish  | Examiner           | Live      |   EL   |
 ------------------------------------------------------------------------------
```

File /Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/Pca_Verification/pca_pump/move_dosed.vcg

procedure Pca_Pump.Move_Dosed

VCs generated 02-JUL-2014 13:08:10

VCs simplified 02-JUL-2014 13:08:20

File /Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/Pca_Verification/pca_pump/move_dosed.dpc

DPCs generated 02-JUL-2014 13:08:10

DPC ZombieScoped 02-JUL-2014 13:08:2

VCs **for** procedure_move_dosed :

#	From	To	Proved By	Dead Path	Status
1	start	rtc check @ 26	Inference	Unchecked	IU
2	start	rtc check @ 26	Inference	Unchecked	IU
3	start	assert @ 27	Inference	Live	IL
4	27	assert @ 27	Inference	Live	IL
5	27	rtc check @ 28	Inference	Unchecked	IU
6	start	rtc check @ 30	Inference	Unchecked	IU
7	27	rtc check @ 30	Inference	Unchecked	IU
8	start	assert @ finish	Examiner	Dead	ED
9	27	assert @ finish	Examiner	Live	EL

File /Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/Pca_Verification/pca_pump/read_dosed.vcg

function Pca_Pump.Read_Dosed

VCs generated 02-JUL-2014 13:08:10

VCs simplified 02-JUL-2014 13:08:21

File /Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/Pca_Verification/pca_pump/read_dosed.dpc

DPCs generated 02-JUL-2014 13:08:10

```
DPC ZombieScoped 02-JUL-2014  13:08:2

VCs for function_read_dosed :

 ------------------------------------------------------------------------------
| #   | From  | To                 | Proved By          | Dead Path | Status |
|------------------------------------------------------------------------------
| 1   | start |    assert @ 17     | Inference          | Live      |   IL   |
| 2   | 17    |    assert @ 17     | Inference          | Live      |   IL   |
| 3   | 17    | rtc check @ 18     | Undischarged       | Unchecked |   UU   |
| 4   | 17    |    assert @ finish | Inference          | Live      |   IL   |
 ------------------------------------------------------------------------------



==============================================================================
Summary:

The following subprograms have undischarged VCs (excluding those proved false):

   1  /Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/Pca_Verification/pca_pump/increase_dosed
     .vcg
   1  /Volumes/External/VMS/shared/aadl-medical/pca-pump-beagleboard/Pca_Verification/pca_pump/read_dosed.vcg

Proof strategies used by subprograms
------------------------------------------------------------------------
Total subprograms with at least one VC proved by examiner:             2
Total subprograms with at least one VC proved by simplifier:           2
Total subprograms with at least one VC proved by contradiction:        0
Total subprograms with at least one VC proved with user proof rule:    0
Total subprograms with at least one VC proved by Victor:               0
Total subprograms with at least one VC proved by Riposte:              0
Total subprograms with at least one VC proved using checker:           0
Total subprograms with at least one VC discharged by review:           0

Maximum extent of strategies used for fully proved subprograms:
------------------------------------------------------------------------
Total subprograms with proof completed by examiner:                    0
Total subprograms with proof completed by simplifier:                  1
Total subprograms with proof completed with user defined rules:        0
```

```
Total subprograms with proof completed by Victor:                        0
Total subprograms with proof completed by Riposte:                       0
Total subprograms with proof completed by checker:                       0
Total subprograms with VCs discharged by review:                         0

Overall subprogram summary:
------------------------------------------------------------------------
Total subprograms fully proved:                                          1
Total subprograms with at least one undischarged VC:                     2  <<<
Total subprograms with at least one false VC:                            0
                                                                       -----
Total subprograms for which VCs have been generated:                     3



ZombieScope Summary:
------------------------------------------------------------------------
Total subprograms for which DPCs have been generated:                    3
Total number subprograms with dead paths found:                          1
Total number of dead paths found:                                        1



VC summary:
------------------------------------------------------------------------
Note: (User) denotes where the Simplifier has proved VCs using one or
      more user-defined proof rules.

Total VCs by type:
------------------
                    Total   Examiner Simplifier    Undisc.
Assert/Post             8          3          5          0
Precondition            0          0          0          0
Check stmnt.            0          0          0          0
Runtime check           7          0          5          2
Refinem. VCs            0          0          0          0
Inherit. VCs            0          0          0          0
============================================================
Totals:                15          3         10          2 <<<
%Totals:                          20%        67%        13%
```

```
==================== End of Semantic Analysis Summary ======================
```

Listing 6.6: *Second POGS report*

Now, we can see progress. Only 2 VCs (13%) are undischarged in comparison to 3 (20%) previously.

Then rerun Kiasan.

pca-pump-verification-step4

`Move_Dosed` and `Increase_Dosed` are fine: no Exception cases.

`Read_Dosed` ConstraintError: the value being assigned to Result is too small. After look at the pre and post state it seems weird. After investigation and talk with Kiasan Developer, it was determined that there is a bug in Kiasan v1(for SPARK 2005). More precisely: checking overflows. For the purpose of verification `Drug_Volume` type range was changed to $0 - (2^{15} - 1)$. Negative values in this case are unnecessary. It will give range up to around 1000000. Which is sufficient even if calculations are made in micro liters (as it is in case of PCA Pump implementation). 1000000 micro liters is 1000 ml, which is 1 liter. Which is extreme amount of drug in case of PCA Pump, according to Requirement Document [LHC13]. The bug with type ranges is fixed in Kiasan v2 (for SPARK 2014).

Another problem is size of Dosed array (3600 elements). First of all, Kiasan array bound and loop bound has to be increased (from default 10). Another thing is computational complexity. The state space grow exponentially and it takes a lot of time to analyze array of 3600 elements. Thus for verification purposes array size was change to 60 elements. Along with increasing array bounds and loop bounds for Kiasan also to 60.

After rerun Kiasan, there is valid test case for `Read_Dose`, but there are also 59 Exception cases: Range violation (UPPER), which means there is possible overflow. One way to fix it is to add `--# assume` annotation to loop in function body, but Kiasan v1 does not support it. Another way is to add pre-condition, which assure, that sum of elements is lower than

`Drug_Volume'Last`. SPARK does not provide simple library for summing array (like Contracts for Java provide). Thus, this function has to be implemented. However, its implementation is the same like `Read_Dosed`. It sum all elements of array. Sum function specification and body is presented on listing 6.7.

```
function Sum(Arr : Doses_Array) return Drug_Volume;


function Sum(Arr : Doses_Array) return Drug_Volume
is
    Result : Drug_Volume := 0;
begin
    for I in Doses_Array_Index loop
        --# assert true;
        Result := Result + Arr(I);
    end loop;
    return Result;
end Sum;
```

Listing 6.7: *Sum function for summing all elements of array*

After rerun Kiasan, there is only valid test case.

pca-pump-verification-step5

The last thing which can be improved by code contracts is checking if `Move_Dosed` procedure works as expected. In that purpose three postconditions were added (listing 6.8). First checks if the last element is equal to 0. Second and third checks two possible scenarios:

- before running procedure, the first element is equal to 0: amount of dosed drug in last hour will not change after Dosed procedure execution
- the first element is greater than 0: after Dosed procedure execution, the amount of drug dosed in last hour will decrease, because first element value will no longer be in last hour range

```
--# post Dosed(Doses_Array_Index'Last) = 0
--#      and (Dosed~(Doses_Array_Index'First)=0 -> Read_Dosed(Dosed~) = Read_Dosed(Dosed))
```

```
--#      and (Dosed~(Doses_Array_Index'First)>0 -> Read_Dosed(Dosed~) > Read_Dosed(Dosed));
```

Listing 6.8: *Postconditions added to Move_Dosed procedure*

After adding these postconditions Kiasan generates 2 test cases to check both mentioned scenarios. There is no error cases, which means that procedure works as expected.

Better way to validate such requirements is Unit testing. In section 6.4, there is overview of unit tests created to test behavior described above.

Running Examiner and SPARKSimp after all changes (truncated result):

```
VCs for procedure_increase_dosed :
 ------------------------------------------------------------------------------
| #   | From  | To                  | Proved By          | Dead Path | Status |
|------------------------------------------------------------------------------
| 1   | start | rtc check @ 20      | Undischarged       | Unchecked |   UU   |
| 2   | start |    assert @ finish  | Examiner           | Live      |   EL   |
 ------------------------------------------------------------------------------

VCs for procedure_move_dosed :
 ------------------------------------------------------------------------------
| #   | From  | To                  | Proved By          | Dead Path | Status |
|------------------------------------------------------------------------------
| 1   | start | rtc check @ 37      | Inference          | Unchecked |   IU   |
| 2   | start | rtc check @ 37      | Inference          | Unchecked |   IU   |
| 3   | start |    assert @ 38      | Inference          | Live      |   IL   |
| 4   | 38    |    assert @ 38      | Inference          | Live      |   IL   |
| 5   | 38    | rtc check @ 39      | Inference          | Unchecked |   IU   |
| 6   | start | rtc check @ 41      | Inference          | Unchecked |   IU   |
| 7   | 38    | rtc check @ 41      | Inference          | Unchecked |   IU   |
| 8   | start |    assert @ finish  | Inference          | Dead      |   ID   |
| 9   | 38    |    assert @ finish  | Undischarged       | Live      |   UL   |
 ------------------------------------------------------------------------------

VCs for function_read_dosed :
 ------------------------------------------------------------------------------
| #   | From  | To                  | Proved By          | Dead Path | Status |
|------------------------------------------------------------------------------
| 1   | start |    assert @ 28      | Inference          | Live      |   IL   |
```

```
| 2   | 28    |    assert @ 28      | Inference          | Live      |   IL   |
| 3   | 28    | rtc check @ 29      | Undischarged       | Unchecked |   UU   |
| 4   | 28    |    assert @ finish  | Inference          | Live      |   IL   |
 ------------------------------------------------------------------------------


VCs for function_sum :
 ------------------------------------------------------------------------------
| #   | From  | To                  | Proved By          | Dead Path | Status |
|------------------------------------------------------------------------------
| 1   | start |    assert @ 11      | Inference          | Live      |   IL   |
| 2   | 11    |    assert @ 11      | Inference          | Live      |   IL   |
| 3   | 11    | rtc check @ 12      | Undischarged       | Unchecked |   UU   |
| 4   | 11    |    assert @ finish  | Inference          | Live      |   IL   |
 ------------------------------------------------------------------------------


Total VCs by type:
------------------
                    Total   Examiner Simplifier    Undisc.
Assert/Post            11          1          9          1
Precondition            0          0          0          0
Check stmnt.            0          0          0          0
Runtime check           8          0          5          3
Refinem. VCs            0          0          0          0
Inherit. VCs            0          0          0          0
============================================================
Totals:                19          1         14          4 <<<
%Totals:                           5%        74%        21%
```

Listing 6.9: *Third POGS report*

Now, there is 4 undischarged VCs, but total number of generated VCs is 19. In previous runs there was only 15. Thus there is 4 new VCs and 2 of them are undischarged. The reason is introduction of `Sum` function of all subprograms which are using it. To confirm this, look at all undischarged VCs. Which is: 1st VC in `increase_dosed.siv` file (listing 6.10, 9th VC in `move_dosed.siv` file (listing 6.11, 3rd VC in `read_dosed.vcg` file (listing 6.12) and 3rd VC in `sum.vcg` file (listing 6.13). They conform to subprograms: `Increase_Dosed`, `Move_Dosed`, `Read_Dosed` and

Sum respectively.

```
procedure_increase_dosed_1.
H1:    read_dosed(dosed) <= 32767 - dose_volume .
H2:    for_all(i___1 : integer, 1 <= i___1 and i___1 <= 60 -> 0 <= element(
          dosed, [i___1]) and element(dosed, [i___1]) <= 32767) .
H3:    dose_volume >= 0 .
H4:    dose_volume <= 32767 .
H5:    integer__size >= 0 .
H6:    drug_volume__size >= 0 .
H7:    drug_volume__base__first <= drug_volume__base__last .
H8:    doses_array_index__size >= 0 .
H9:    drug_volume__base__first <= 0 .
H10:   drug_volume__base__last >= 32767 .
       ->
C1:    element(dosed, [60]) + dose_volume <= 32767 .
```

Listing 6.10: *Undischarged Verification Condition from increase_ dosed.siv file*

```
procedure_move_dosed_9.
H1:    element(dosed, [58]) = element(dosed, [59]) .
H2:    for_all(i___1 : integer, 1 <= i___1 and i___1 <= 60 -> 0 <= element(
          dosed, [i___1]) and element(dosed, [i___1]) <= 32767) .
H3:    element(dosed, [60]) >= 0 .
H4:    element(dosed, [60]) <= 32767 .
H5:    integer__size >= 0 .
H6:    drug_volume__size >= 0 .
H7:    drug_volume__base__first <= drug_volume__base__last .
H8:    doses_array_index__size >= 0 .
H9:    drug_volume__base__first <= 0 .
H10:   drug_volume__base__last >= 32767 .
       ->
C1:    element(dosed~, [1]) = 0 -> read_dosed(dosed~) = read_dosed(update(
          update(dosed, [59], element(dosed, [60])), [60], 0)) .
C2:    element(dosed~, [1]) > 0 -> read_dosed(dosed~) > read_dosed(update(
          update(dosed, [59], element(dosed, [60])), [60], 0)) .
```

Listing 6.11: *Undischarged Verification Condition from move_ dosed.siv file*

```
function_read_dosed_3.
H1:    loop__1__i > 1 -> result >= element(dosed, [loop__1__i - 1]) .
```

```
H2:    for_all(i___1 : integer, 1 <= i___1 and i___1 <= 60 -> 0 <= element(
          dosed, [i___1]) and element(dosed, [i___1]) <= 32767) .
H3:    sum(dosed) <= 32767 .
H4:    loop__1__i >= 1 .
H5:    loop__1__i <= 60 .
H6:    result >= 0 .
H7:    result <= 32767 .
H8:    integer__size >= 0 .
H9:    drug_volume__size >= 0 .
H10:   drug_volume__base__first <= drug_volume__base__last .
H11:   doses_array_index__size >= 0 .
H12:   drug_volume__base__first <= 0 .
H13:   drug_volume__base__last >= 32767 .
       ->
C1:    result + element(dosed, [loop__1__i]) <= 32767 .
```

Listing 6.12: *Undischarged Verification Condition from read_ dosed.siv file*

```
function_sum_3.
H1:    for_all(i___1 : integer, 1 <= i___1 and i___1 <= 60 -> 0 <= element(arr,
          [i___1]) and element(arr, [i___1]) <= 32767) .
H2:    loop__1__i >= 1 .
H3:    loop__1__i <= 60 .
H4:    result >= 0 .
H5:    result <= 32767 .
H6:    integer__size >= 0 .
H7:    drug_volume__size >= 0 .
H8:    drug_volume__base__first <= drug_volume__base__last .
H9:    doses_array_index__size >= 0 .
H10:   drug_volume__base__first <= 0 .
H11:   drug_volume__base__last >= 32767 .
       ->
C1:    result + element(arr, [loop__1__i]) <= 32767 .
```

Listing 6.13: *Undischarged Verification Condition from sum.siv file*

In `Move_Dosed` procedure, tools cannot prove implications in post conditions. Fortunately, it is already proved by Bakar Kiasan. The problem in `Increase_Dosed`, `Read_Dosed` and `Sum` is the same. Tools cannot verify, that adding `Result` and some element of `Dosed` array will not cause

overflow. Bakar Kiasan can prove correctness of `Increase_Dosed` and `Read_Dosed`. However only, with assumption that `Sum` is correct. `Sum` cannot be proved by Bakar Kiasan. Four exception cases indicating possible overflow are generated. Thus, the only way to prove correctness of this module is to assume, that helper function `Sum` is correct.

Complete code of module for dose monitoring can be found in D.

Unfortunately, introduced changes (pre- and postconditions) cannot be applied to PCA Pump prototype implementation, because - as mentioned in chapter 2.6 - protected objects cannot be used in proof annotations (pre- and postconditions).

This shows, how code implemented based on translation from AADL/BLESS can be verified using SPARK tools.

6.3 Verification of generated code

Raw, generated code cannot be verified, because Examiner return syntax errors. The reason is that some parts of code are not implemented. Especially, BLESS assertions, which are not even defined. In order to verify generated code, it has to be at least partially implemented.

6.3.1 Adding implementation to generated code

6.4 AUnit tests

Better way to prove expected behavior of `Move_Dosed` in Dose monitoring module is to create AUnit test. To check both behaviors of `Move_Dosed` procedure, two tests have been created:

- `Test_Move_Dosed_First_Element_Zero` - first element is 0, then after execution of the procedure dosed amount of drug should be not changed
- `Test_Move_Dosed_First_Element_Not_Zero` - first element is greater than 0, then after execution

of the procedure dosed amount of drug should be smaller than before

Both test cases are presented on listing 6.14.

```
procedure Test_Move_Dosed_First_Element_Zero (Gnattest_T : in out Test) is
  pragma Unreferenced (Gnattest_T);
  Pre_Sum : Pca_Pump.Drug_Volume := 0;
  Post_Sum : Pca_Pump.Drug_Volume := 0;
begin
  -- Arrange
  Pre_Sum := Pca_Pump.Read_Dosed;

  -- Act
  Pca_Pump.Move_Dosed;
  Post_Sum := Pca_Pump.Read_Dosed;

  -- Assert
  AUnit.Assertions.Assert
    (Post_Sum = Pre_Sum,
     "Total dose changed: " &  Pca_Pump.Drug_Volume'Image(Pre_Sum) & " /= " &  Pca_Pump.Drug_Volume'Image(
     Post_Sum));
end Test_Move_Dosed_First_Element_Zero;

procedure Test_Move_Dosed_First_Element_Not_Zero (Gnattest_T : in out Test) is
  pragma Unreferenced (Gnattest_T);
  Pre_Sum : Pca_Pump.Drug_Volume := 0;
  Post_Sum : Pca_Pump.Drug_Volume := 0;
begin
  -- Arrange
  Pca_Pump.Increase_Dosed;
  for I in Pca_Pump.Doses_Array_Index range 1 .. Pca_Pump.Doses_Array_Index'Last-1 loop
     Pca_Pump.Move_Dosed;
  end loop;
  Pre_Sum := Pca_Pump.Read_Dosed;

  -- Act
  Pca_Pump.Move_Dosed;
  Post_Sum := Pca_Pump.Read_Dosed;

  -- Assert
```

```
  AUnit.Assertions.Assert
    (Post_Sum < Pre_Sum,
     "Total dose changed: " &  Pca_Pump.Drug_Volume'Image(Pre_Sum) & " should be greater than " &  Pca_Pump.
     Drug_Volume'Image(Post_Sum));
end Test_Move_Dosed_First_Element_Not_Zero;
```

Listing 6.14: *AUnit tests for Move_Dosed procedure*

[add other test cases?]

6.5 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?

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 - AADL models don't contains subprograms, which would be useful in AADL2SPARK translator creation

Chapter 8

Future work

What has to be done now.

translation of BLESS state machine (issue: time notion): * states * transitions

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

translations for SPARK 2014 (for now, thread -> task translation can be done in Ada 2012 and then Sparking Ada)

try to apply generics on types translation

try to apply child packages for feature

extend property set translation (only aadlinteger and simple types are handled)

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

Decompose code (get rid of 1 big/huge package): * separate packages * child packages? * maybe every thread should be in child package?

PCA Pump: * Interface to ICE * drug concentration * monitoring of infused drug by sensors

Two way communication between SPARK packages: circular dependency issue (4.2.1).

Interport connections: table 10-3 in AADL book (p.193) and 1 to many connections

(multiple events calling).

Add task periods

Automatic translator for sequential programs (without protected objects).

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.

[DRH07] Xianghua Deng, Robby, and John Hatcliff. Kiasan/kunit: Automatic test case generation and analysis feedback for open object-oriented systems. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 3–12. IEEE Computer Society Washington, DC, 2007.

[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.

[HKL+12] John Hatcliff, Andrew King, Insup Lee, Alasdair MacDonald, Anura Fernando, Michael Robkin, Eugene Vasserman, Sandy Weininger, and Julian M. Goldman. Rationale and architecture principles for medical application platforms. In *Cyber-Physical Systems (ICCPS), 2012 IEEE/ACM Third International Conference on*, pages 3 – 12. IEEE, 2012.

[HLW12] John Hatcliff, Kejia Li, and Steve Warren. Component-based app design for platform-oriented devices in a medical device coordination framework. In *ACM SIGHIT International Health Informatics Symposium*, pages 343 – 352, 2012.

[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.

[Lev12] Nancy G. Leveson. *Engineering a Safer World*. The MIT Press, 2012.

[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.

[SC12] Loren Segal and Patrice Chalin. A comparison of intermediate verification languages: Boogie and sireum/pilar. In *Verified Software: Theories, Tools, Experiments*, pages 130–145. Springer, 2012.

[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

Simplified PCA Pump AADL/BLESS models

Content of this appendix.

Appendix B

PCA Pump Prototype - simple, working example

Content of this appendix.

Appendix C

PCA Pump Prototype - translated from AADL/BLESS

Appendix D

PCA Pump - dose monitor module

Final version of `PCA_Verification` and AUnit tests.