A MODEL-DRIVEN DEVELOPMENT AND VERIFICATION APPROACH FOR MEDICAL DEVICES

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

Abstract

Medical devices are safety-critical systems whose failure may put human life in danger. They are becoming more advanced and thus more complex. This leads to bigger and more complicated code-bases that are hard to maintain and verify. Model-driven development provides high-level and abstract description of the system in the form of models that omit details, which are not relevant during the design phase. This allows for certain types of verification and hazard analysis to be performed on the models. These models can then be translated into code. However, errors that do not exist in the models may be introduced during the implementation phase. Automated translation from verified models to code may prevent to some extent.

This thesis proposes approach for model-driven development and verification of medical devices. Models are created in AADL (Architecture Analysis & Design Language), a language for software and hardware architecture modeling. AADL models are translated to SPARK Ada, contract-based programming language, which is suitable for software verification. Generated code base is further extended by developers to implement internals of specific devices. Created programs can be verified using SPARK tools.

A PCA (Patient Controlled Analgesia) pump medical device is used to illustrate the primary artifacts and process steps. The foundation for this work is "Integrated Clinical Environment Patient-Controlled Analgesia Infusion Pump System Requirements" document and AADL Models created by Brian Larson. In addition to proposed model-driven development approach, a PCA pump prototype was created using the BeagleBoard-xM device as a platform. Some components of PCA pump prototype were verified by SPARK tools and Bakar Kiasan.

Table of Contents

List of Figures

List of Tables

Acknowledgments

"Showing gratitude is one of the simplest yet most powerful things humans can do for each other."

— Randy Pausch, *Last Lecture*

I would like to say thank you to all people, who helped me pursue Master of Science program in Computer Science at Kansas State University. Many thanks to Dr. Andrew Rys who encourage me to apply for Graduate School, and was always helpful with an advice. I wish to thank, my major professor, Dr. John Hatcliff who admitted me to SAnToS Laboratory research group, and enabled me to be involved in research. I met there many passionate people and great researchers. Furthermore, without Dr. Hatcliff's guidance, this thesis will not be accomplished. Thanks to Dr. Robby, who was always helping me in my research, giving valuable suggestions and ideas. Thank you to Jason Belt for sharing his knowledge and experience with me, which played significant role in my research career. Thanks to Brian Larson, whose work, was inspiration of my Master thesis. Many thanks to Dr. Eugene Vasserman for serving on my committee and for his valuable suggestions about this work. A special thanks for Venkatesh Prasad Ranganath. Conversations with him and his suggestions played significant role in accomplishing this thesis.

Dedication

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

1

Introduction

"Life is a journey, not a destination."

– Ralph Waldo Emerson

Software is present in all aspects of our lives, from the simple program in alarm clocks to iPads, through cars, refrigerators and computers. Furthermore, our lives are getting more and more dependent 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. This class comprised of software for airplanes, medical devices, satellites, and rockets. Safety-critical systems are usually real-time - their correctness depends not only on logic, but also upon the time constraints (hard and soft deadlines in which operations has to be accomplished).

Software Engineering for real-time 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 a different extent. In the case of the aforementioned word processor, software assurance is not critical. When it crashes, it can be restarted. In worst case scenario, some work might be lost. Airplane software errors may put human lives in danger or even cause death. Thus for safety-critical systems, the security and correctness

are crucial. Behind these reasons, different software design methodology, different properties of programming languages and verification tools are needed.

Part of safety-critical systems design and development is their verification. The goal of software verification is to assure that software satisfies requirements. Furthermore, during verification process some potential issues might be detected by discovering possible program states and execution paths.

1.1 Motivation

Nowadays, medical devices work rather independently. This leads to many accidents, which could have been avoided by their interoperability. For example, over-dose of a drug (e.g. morphine) delivered by the patient-controlled analgesia (PCA) pump after surgery can lead to low blood oxygenation or even lack of pulse [OG11]. That can lead to patient's death. The PCA pump does not monitor an oxygen level, but oxygen monitoring device does. If these two devices are organized in centralized system, which implements safety interlock mechanism to shutdown the pump when low blood oxygenation[1] is detected, accident can be avoided.

In order to communicate, devices have to use compatible interfaces and protocols. There is a concept of "Integrated Clinical Environment" (ICE). It is captured in the standard ASTM F2761, which describes a functional architecture for inter operable systems [HKL+12]. The "Laboratory for Specification, Analysis, and Transformation of Software" (SAnToS Laboratory) created "Medical Device Coordination Framework" (MDCF) [HKL+12], which is prototype implementation of ICE. The MDCF vision for ICE is to have requirements documents and conforming software and hardware models. This will enable different medical devices, created by different vendors, to be connected and work under supervision of a centralized system.

[1] Blood oxygenation is also referred as SpO_2

In last decades, model-driven development [SVC06] became standard for safety critical systems design. It provides higher level of abstraction, which enables to focus on business problems instead of technology. Models captures domain knowledge and systems analysis, disregarding implementation details, is possible. Additionally, software validation and verification can be executed at design-time. The model-driven development approach proposed in this thesis is a response for the need to create code from models. The PCA pump prototype created in this thesis is as an example of a medical device, which ultimately will work under MDCF.

1.2 Technologies

AADL (Architecture Analysis & Design Language) [FG13] is a modeling language for representing hardware and software. It is used for real-time, safety critical and embedded systems [FWH]. 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) [LCH13] is an AADL annex sub language defining behavior of components. The goal of BLESS is to automatically check the correctness of AADL models.

Ada is one of the most popular programming languages (along with C/C++) targeted at embedded and real-time systems. SPARK Ada [Bar13] is a subset of Ada, designed for the development of safety and security critical systems. This subset is designed to facilitate static analysis and program verification, which allows to reason about and prove correctness of programs and their entities. There are also SPARK tools for software verification, including tools provided by Altran UK and AdaCore (the developers of SPARK) as well as research groups such as SAnToS Laboratory at Kansas State University.

1.3 Contribution

This thesis demonstrates mapping of AADL/BLESS models to code in 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 "Open Patient-Controlled Analgesia Infusion Pump System Requirements" [LH14, LHC13].
- Identification and analysis of PCA pump and Infusion pumps properties and internals required for implementation.
- Cross-compilation and testing of SPARK Ada 2005 and 2014 programs on BeagleBoard-xM platform.
- Implementation of PCA pump based on [LH14] and AADL/BLESS models.
- AADL/BLESS to SPARK Ada translation schemes.
- Translation of simplified PCA Pump models (based on created translation schemes).
- Design requirements for AADL/BLESS to SPARK Ada translator.
- Practical demonstration of SPARK 2005 and SPARK 2014 verification tools: its capabilities and limitations:
  - SPARK Examiner
  - SPARKSimp
  - Proof Obligation Summarizer (POGS)
  - Bakar Kiasan
  - GNATprove

1.4 Organization

This thesis is organized as follows:

- Chapter 2 is background that gives details about ICE, MDCF, Model-Driven Development, AADL, BLESS, SPARK Ada and its verification tools.
- 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 and code translated from simplified version of AADL models.
- Chapter 7 summarizes all work which has been done in this thesis.
- Chapter 8 is the future work that can be done in this topic.

2

Background

"Experience is not what happens to you;
it's what you do with what happens to you."

– *Aldous Huxley*

This chapter is a brief introduction of all technologies and tools used in this thesis. They 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 done: Integrated Clinical Environment (ICE) standard 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

The concept of the "Integrated Clinical Environment" (ICE) was initiated and championed by Dr. Julian Goldman from Center for Integration of Medicine & Innovative Technology.[1] The main idea is to create a platform for integrating medical devices in a local area network. ICE will enable clinicians and software system to make decisions based not only on output

[1] http://www.cimit.org/

from one device, but from different devices working together in network. Moreover, ICE comprises components that may be implemented by different vendors. Such components are medical devices and applications to supervise them. The purpose of ICE is to solve current issues with medical devices, which usually operate independently and requires more human attention and control through checking output of every device manually and then making decisions. ICE propose a concept of Medical Application Platform [HKL+12] that assure medical devices interoperability and provides execution environment for clinical applications. Different devices can exchange data and centralized system can make decisions (based on this data) automatically. For example when PCA pump infuse some drug to patient's vein and Pulse Oximeter detects low oxygen level, ICE can coordinate PCA pump shutdown.

Figure 2.1 presents high-level overview of one particular application of an ICE system. Medical devices (PCA Pump, Respiratory Rate Monitor and Pulse Oximeter) are connected to the system, which monitors or controls them. There is communication between devices and ICE in order to exchange data. ICE can make decisions (such as PCA Pump shutdown) based on them.

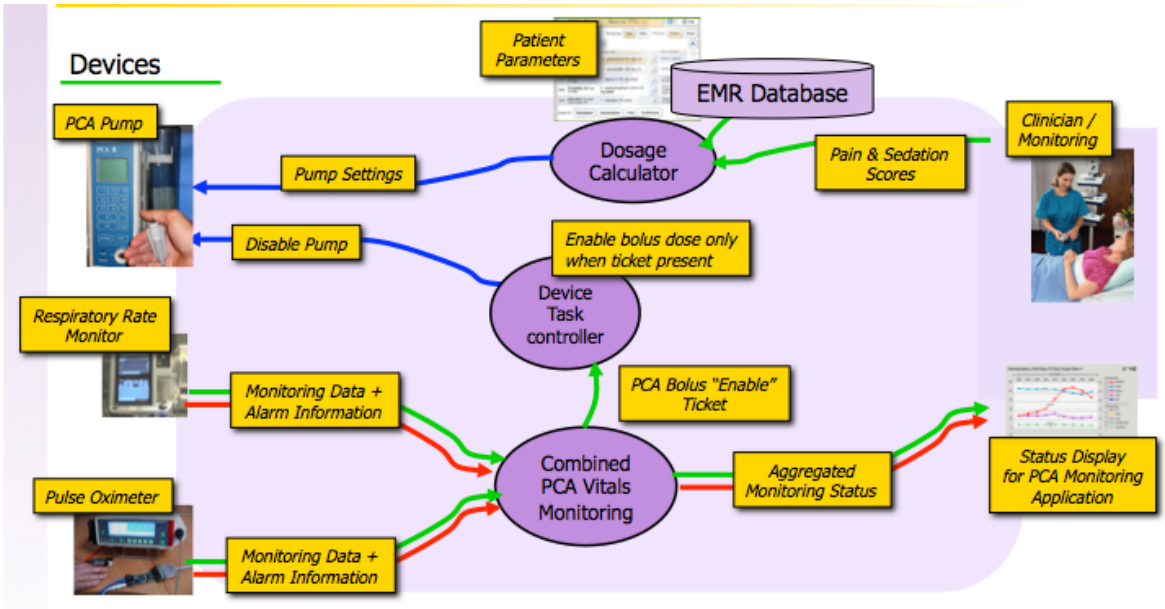

Figure 2.1: ICE Closed Loop Control

2.2 Medical Device Coordination Framework

Medical Device Coordination Framework (MDCF) [HKL+12], jointly developed by SAnToS Laboratory (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. This project is a response to a request from Food and Drug Administration (FDA) to build a prototype of ICE. There is a vision of different medical devices, created by different vendors, connected and working under centralized system. MDCF is designed to illustrate by example issues related to functional concepts, safety, security, verification and certification.

The following comprise the goals of the MDCF project:

- 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

Currently, MDCF use only mock devices, which are Java desktop applications. PCA Pump Prototype, developed in this thesis, aims to be the realistic hardware device targeted specifically for the MDCF.

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

connected to Message Bus, which along with Device Manager and Device Database ensures communication with Application Manager [HKL+12].

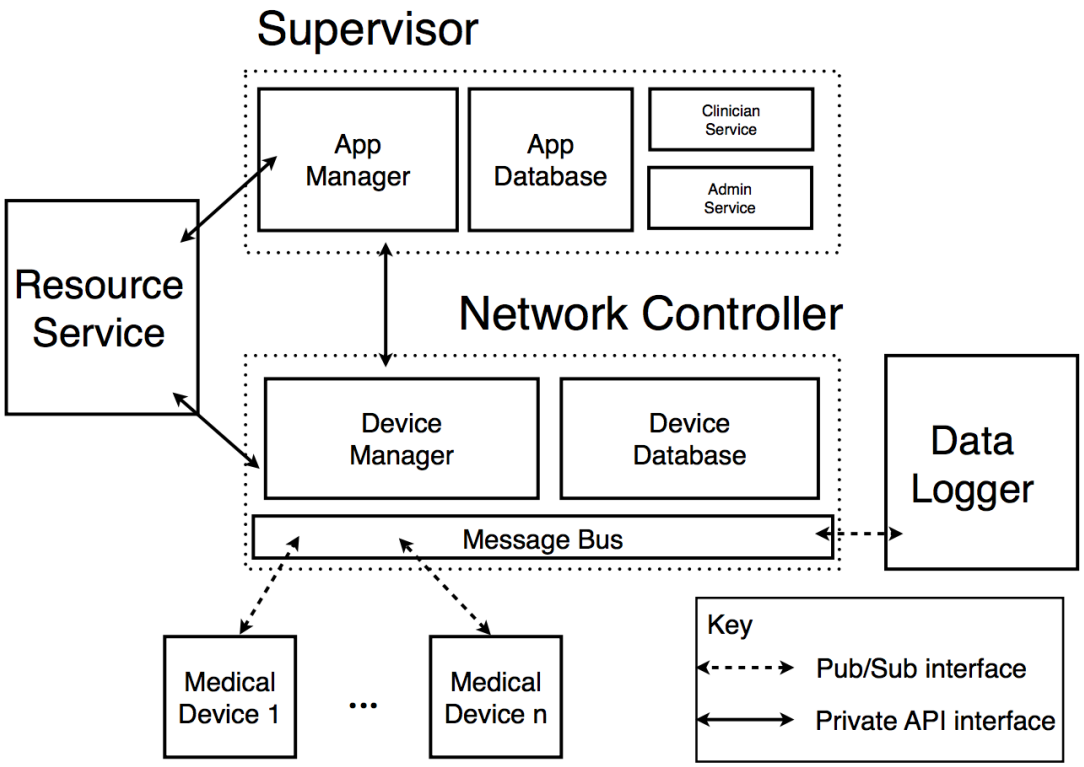

Figure 2.2: MDCF architecture and example app virtual machine (lower right)

2.3 AADL

AADL stands for Architecture Analysis & Design Language. It is used to model embedded and real-time systems. AADL allows for 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, and code generation.

AADL has its roots in DARPA[2] funded research. The first version (1.0) was approved in

[2]http://www.darpa.mil

2004 under technical leadership of Peter Feiler.[3] AADL is develop by SAE AADL standard committee.[4] AADL version 2.0 was published in January 2009. The most recent version (2.1[5]) was published in September 2012.[6]

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

AADL contains entities for modeling software and hardware components, and 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

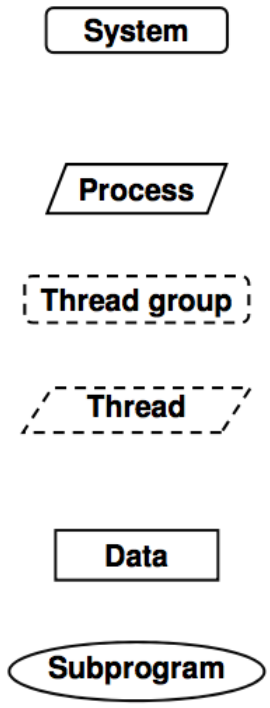

Figure 2.3: AADL Application Software Components

[3] http://wiki.sei.cmu.edu/aadl/index.php/The_Story_of_AADL/
[4] https://wiki.sei.cmu.edu/aadl/index.php/Main_Page
[5] https://wiki.sei.cmu.edu/aadl/images/d/d2/AADL_V2.1_Syntax_Card.pdf
[6] https://wiki.sei.cmu.edu/aadl/index.php/Standardization

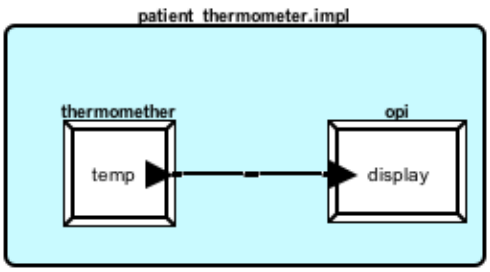

Figure 2.4: AADL model of simple thermometer

An example AADL model is shown in graphical representation, in the Figure 2.4. Its textual representation is presented in the Figure 2.5.

```
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;
```

Figure 2.5: AADL model of simple thermometer

There are several tools for AADL model support, such as: OSATE (see Section 2.3.1), STOOD (AADL design tool),[7] ADELE (graphical editor),[8] Cheddar (real time scheduling

[7] http://www.ellidiss.com/products/stood
[8] https://wiki.sei.cmu.edu/aadl/index.php/Adele

tool),[9] AADLInspector (model processing framework),[10] or Ocarina (see Section 2.7.1).

AADL focuses on architectural modeling, but it can be extended via 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, see Section 2.4)
  - Error-model annex: specifies fault and propagation concerns
  - ARINC653 annex: defines modeling patterns for modeling avionics systems
  - Data-Model annex: describes the modeling of specific data types and structures with AADL

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

AADL is used as a modeling language in this thesis.

2.3.1 OSATE

Open Source AADL Tool Environment (OSATE) is a set of plug-ins on top of the Eclipse platform. It provides a tool set for front-end processing of AADL models. OSATE is developed mainly by SEI (Software Engineering Institute - Carnegie Mellon University).[11] The latest available version of OSATE at the time when this thesis was published is OSATE2.[12]

[9] http://beru.univ-brest.fr/ singhoff/cheddar
[10] http://www.ellidiss.com/products/aadl-inspector
[11] http://www.aadl.info/aadl/currentsite/tool/osate.html
[12] https://wiki.sei.cmu.edu/aadl/index.php/Osate_2

OSATE relies on EMF,[13] UML2 and Xtext.[14] It comprises, e.g., an AADL project wizard, AADL Navigator, and AADL syntax analyzer. OSATE enables the conversion of AADL in textual representation into its standardized graphical representation. There are also plug-ins for OSATE, such as the BLESS[15] and OCARINA[16] plug-ins.

OSATE has been used to develop AADL models for this thesis and work with already existing models.

2.4 BLESS

BLESS (Behavior Language for Embedded Systems with Software) is AADL annex sub-language defining behavior of components for AADL [LCH13]. BLESS comes with a verification framework that enables a developer to build proofs of AADL models of embedded electronic systems with software.

BLESS annex subclauses can be added to AADL models transparently without interfering with other uses of AADL. 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].

BLESS contains three AADL annex sub-languages:

- Assertion - assertions 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

[13] http://www.eclipse.org/modeling/emf/
[14] http://www.eclipse.org/Xtext/
[15] http://bless.santoslab.org/node/5
[16] 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].

In the work for this thesis, subset of BLESS is translated into SPARK contracts and assertions. Detailed overview of supported features can be found in Section 4.1.8.

2.5 SPARK Ada

The Ada programming language was originally designed to meet the US Department of Defense Requirements for programming military applications. Since its first version (Ada 83) it has evolved through multiple versions: Ada 95, Ada 2005 and Ada 2012 (released in December 10, 2012).[17] Ada is actively used in many real-world projects in critical application domains,[18] e.g. Aviation (Boeing[19]), Railway Transportation, Commercial Rockets, Satellites and even Banking. One of the main goals of Ada is to ensure software correctness and safety. Ada includes features that eliminate common errors involving pointers, array bounds violations and unprincipled control flow, in comparison to other programming languages (see Figure 2.6). This is achieved not only by language capabilities, but also by tools for testing and 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 amenable to state analysis and formal methods, by collection of analysis and verification tools. Some Ada constructs are excluded from SPARK to make static analysis feasible [IEC+06]. SPARK 2005 does not include constructs such as pointers, dynamic memory allocation or recursion [IEC+06]. Verification tools (see Section 2.6) produce Verification

[17] http://www.ada2012.org
[18] http://www.seas.gwu.edu/~mfeldman/ada-project-summary.html
[19] http://archive.adaic.com/projects/atwork/boeing.html

Conditions (VCs) to check program correctness. Sample Verification Condition contains checks for:

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

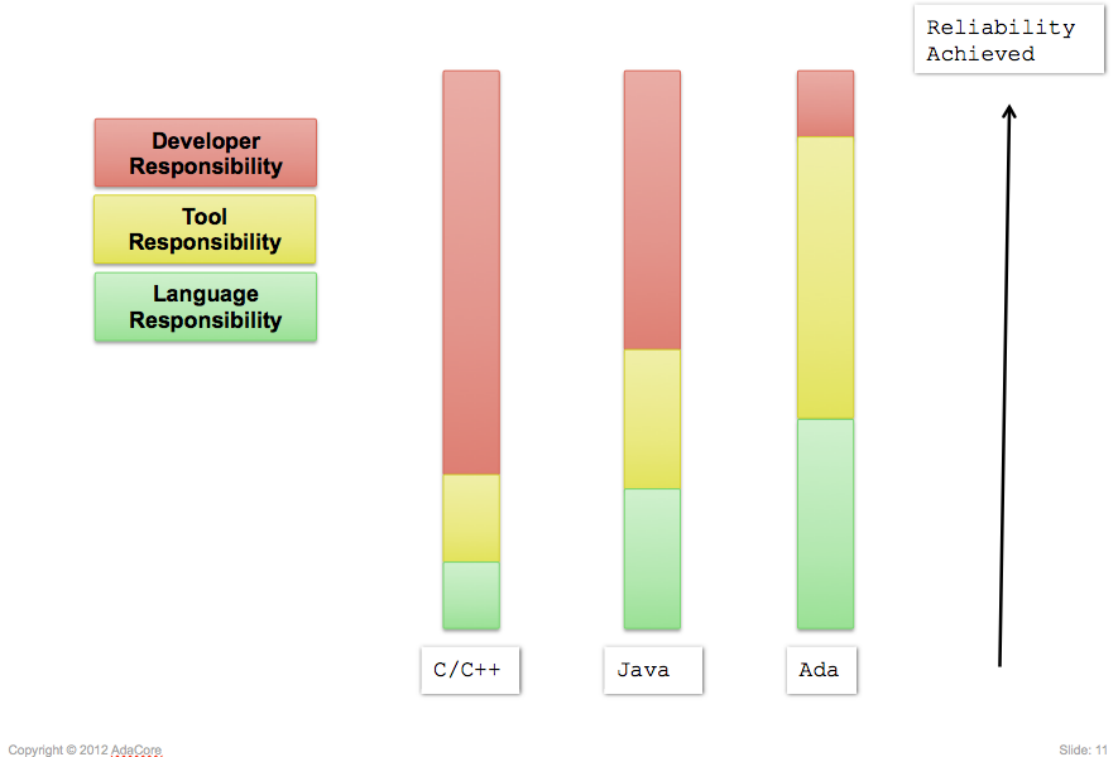

Figure 2.6: Developer responsibility in Ada.[20]

SPARK is used not only for research, but also in industry: aerospace (e.g., EuroFighter Typhoon aircraft,[21] The Lockheed Martin C130J[22] and standard DO-178B[23]), security (e.g.,

[20] http://www.slideshare.net/AdaCore/ada-2012
[21] http://www.eurofighter.com/
[22] http://www.lockheedmartin.com/us/products/c130/c-130j-variants/c-130j-30.html
[23] http://www.adacore.com/gnatpro-safety-critical/avionics/do178b/

MULTi-application Operating System[24]), air traffic management (e.g., iFACTS system[25]) [Bar13]. In practice, because the features of SPARK are limited and because the use of SPARK can be labor intensive, the embedded critical components are written in SPARK while the non-critical components are written in Ada [Cha00].

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 `--#`). This approach allows every SPARK 2005 program to be a valid Ada 2005 program. SPARK annotations contains code contracts (see Table 2.1), which are analyzed by verification tools, but ignored by Ada compiler.

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

Figure 2.7: Sample SPARK procedure with code contracts

Figure 2.7 presents simple procedure with code contracts. It increments variable given as parameter by 1. The `derives` clause specify variable dependency. Its future value depends on its current value. There is precondition saying that the value has to be lower than maximum value of `Integer` type, to avoid overflow. 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).

SPARK 2014[26] (based on Ada 2012) is under development. There is partial tool support (in GNAT Programming Studio), but some language features (such as tasking) are still not

[24]http://www.cardwerk.com/smartcards/MULTOS/
[25]http://www.adacore.com/customers/uks-next-generation-atc-system/
[26]http://www.spark-2014.org

supported. 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]. Some of Ada 2012 features are not allowed in SPARK, e.g.:

- Access types (pointers)
- Exceptions
- Aliasing between variables
- The goto statement
- 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 (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

Continued on next page

Table 2.1 – continued from previous page

SPARK 2005	SPARK 2014	Description
`--# inherit`	not needed	allows to access entities of other packages
`--# pre`	`Pre`	pre condition
`--# post`	`Post`	post condition
`--# assert`	`Assert`	assertion

A sample mapping from SPARK 2005 to 2014 is shown in the Table 2.2. A complete mapping can be found in SPARK 2014 documentation[27] [AL14a].

The previous example (Figure 2.7), translated to SPARK 2014 syntax, is presented in the Figure 2.8.

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

Figure 2.8: 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 can be verified by SPARK 2014 tools. SPARK 2014 does not contains Examiner like SPARK 2005. Instead, proofs are made by GNATprove (see Section 6.5).

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

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 most popular IDE for SPARK Ada is GNAT Programming Studio[28] (see Section 2.5.2). There is also Ada plug-in for Eclipse - GNATbench[29] created by AdaCore.

SPARK Ada is target language for code generation from AADL/BLESS models in this thesis.

2.5.1 GNAT Compiler

The GNAT compiler is an Ada compiler created by AdaCore[30]. 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 and is included in all Linux distributions. GCC is open source, published on GNU General Public License. GCC is divided into a front end and a back end. This architecture enables compiler developers to create new front ends for some language and reuse existing back ends (or vice versa).

[28]http://libre.adacore.com/tools/gps
[29]https://www.adacore.com/gnatpro/toolsuite/gnatbench/
[30]http://www.adacore.com

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 produces object files (`.o`) and Ada Library Information files (`.ali`). Once compilation is done, `gnatmake` invokes `gnatbind` 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. Currently, cross-compilation can only be performed on a 32-bit Linux platform.

GNAT compiler and GNAT cross-compiler has been used to compile SPARK Ada programs created for this thesis.

2.5.2 GNAT Programming Studio (GPS)

GNAT Programming Studio (GPS) is an Integrated Development Environment (IDE) for Ada. It allows to easily manage and compile Ada projects, providing Graphical User Interface as front end for underlaying tools, which have command line interface. Additionally, it enables to create plug-ins using Python and PyGTK.[31] GPS has a plug-ins for SPARK Ada. There is also Sireum Bakar (see Section 2.6.9) plug-in for GPS (developed by SAnToS Laboratory).

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

GPS has been used for creating and editing all SPARK Ada programs created in this thesis.

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

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. Burns, Dobbing, and Vardanega gives the following Ravenscar profile description:

> The Ravenscar Profile is a subset of Ada tasking model, restricted to meet the real-time requirements for safety critical applications such as 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. The concurrency model promoted by the Ravenscar Profile is consistent with the use of tools that allow the static properties of programs to be verified. Potential verification techniques include information flow analysis, schedulability analysis, execution-order analysis and model checking. These techniques allow analysis of a system to be performed throughout its development life cycle, thus avoiding the common problem of finding only during system integration and testing that the design fails to meet its non-functional requirements [BDV04].

Ravenscar profile is available in SPARK 2005, but not yet in SPARK 2014[32] [AL14a]. The default SPARK 2005 profile (sequential) does not enable tasking. In other words, SPARK 2005 tools cannot analyze and reason about concurrent programs if Ravenscar profile flag (`-profile=ravenscar`) is not provided.

To create a task, the task type has to be declared and task variable of this type has to be defined. Ravenscar does not allows dynamic task creation. Thus, all tasks have to exist for the full lifetime of the program [AW01]. Tasks can be declared only in packages, not

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

in subprograms or in other tasks [Bar13]. The priority of each task 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. A lower value indicates lower priority. Figure 2.9 shows sample package with two tasks. Declared tasks have to be implemented in the package body.

```
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;

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

  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;
```

Figure 2.9: Sample tasks

There are two ways to access a variable in different tasks:

- The variable has to be a protected object.
- The variable has to be an atomic type.

A protected object encapsulates a variable in such a way that it is accessible only through protected subprograms. This mechanism uses locking to ensure atomicity. Protected type declaration is similar to task: both a specification and a body has to be defined. Figure 2.10 shows sample tasks with protected type `Integer_Store`, which enables to share an Integer variable between tasks. A protected type has to be declared before tasks that will use it. Otherwise, it will be not visible for them. A protected type body also has to be defined in package body (Figure 2.11).

```
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;
```

Figure 2.10: Sample tasks with protected object

In example given in figures 2.10 and 2.11, `Task1` is writing to `Shared_Var` and `Task2` is reading `Shared_Var`. The highest priority is assigned to the protected object to ensure atomicity during operations on it. The lowest priority is assigned to `Task2`, which is reading `Shared_Var`. Reading

```
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;
```

Figure 2.11: Sample tasks with protected object body

is usually less expensive operation than writing. Thus, to avoid starvation, `Task1` has higher priority than `Task2`. Notice, that `Shared_Var` is declared in the package body, but refined in package specification.

Protected variables may not be used in proof contexts. Thus, if we try to use protected variable in proofs (pre- or postcondition), then SPARK Examiner returns semantic error: `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 requires

other techniques such as model checking and lies outside the scope of SPARK [Bar13]. To preserve the 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 a predefined type such as `Integer`. Thus, a custom type has to be defined. It can be just rename of `Integer` (e.g., `Int32` in the Figure 2.12). Then **pragma** `Atomic` can be applied on this type. Figure 2.12 presents the previous example using atomic types instead of protected objects.

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" [BDV04] and the official Ravenscar Profile documentation (which includes examples) [Tea12] is another good source. The limitations of Tasking in SPARK are reviewed in [AW01].

Ravenscar profile has been used for multitasking applications (including PCA Pump Prototype) created in this thesis.

```
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;
```

Figure 2.12: Sample tasks with atomic type

2.6 SPARK Ada Verification

The goal of software verification is to assure that software satisfies specification and requirements, and to prove the lack of errors. There are two primary types of verification:

- dynamic - performed during the execution of software, e.g. unit tests (by comparison of expected and actual states)
- static - achieved by formal methods, flow analysis, mathematical calculations and logical evaluations (based on formal rendering of specification)

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 that program conforms to a particular class of properties 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 via formal refinement of code from specifications.
- Model checking: enumerate all possible executions and states, and check each state for correctness.

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

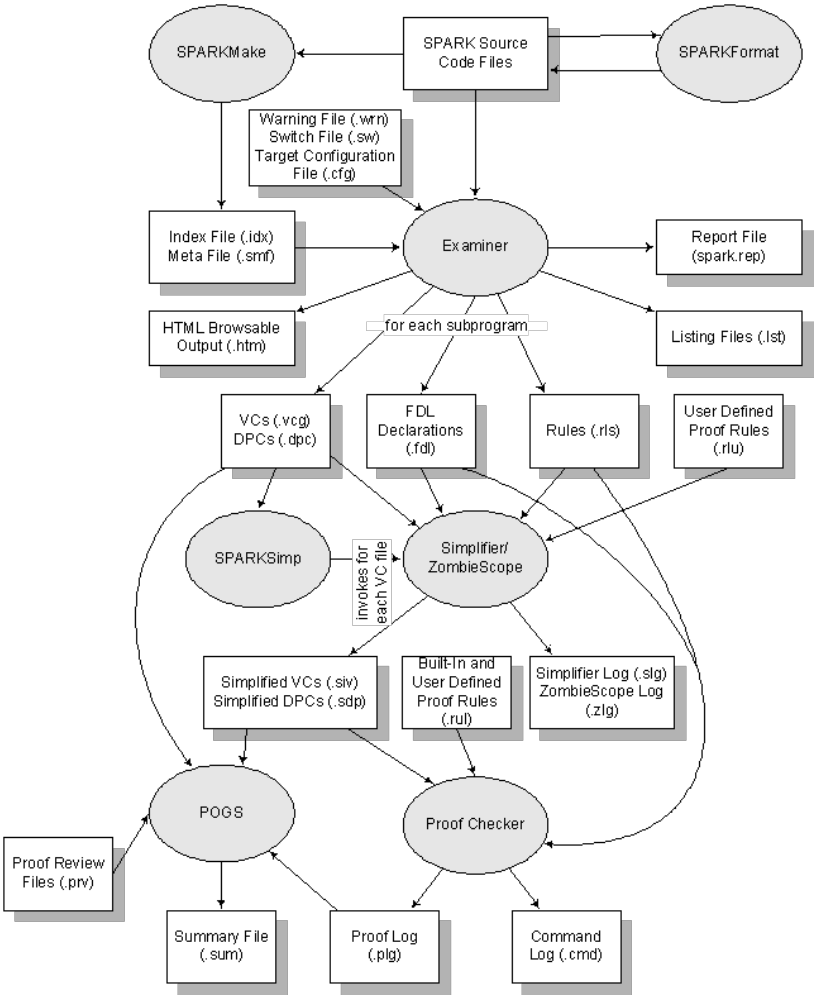

Figure 2.13: Relationship of the Examiner and Proof Tools.[33]

SPARK includes a development and verification tool-set with the following components:

- SPARKMake - generates index file (.idx) and meta file (.smf)
- Examiner - checks syntax, generates Verification Conditions (VCs) and Dead Path Conjectures (DPCs), and discharges (proves) some of them (some might be impossible to discharge)
- Simplifier - simplifies VCs (not discharged by Examiner) and tries to discharge them after simplification process in similar fashion like Examiner

- ZombieScope - finds dead paths
- ViCToR - translates VCs and DPCs to format acceptable by SMT solver and proves correctness using specified SMT solver
- SPARKSimp - runs Simplifier or/and ZombieScope
- POGS - produces verification report
- Proof Checker - discharges VCs or DPCs not discharged by Examiner and Simplifier by carrying out tool-assisted manual proof steps

Relationships between tools and verification flow is presented in the Figure 2.13. 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 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 [Tea11b].

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 the `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 2.14)
- Set SPARK index file (to spark.idx generated by SPARKMake) (Figure 2.15)
- (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), i.e. paths through the code that can never be executed regardless of

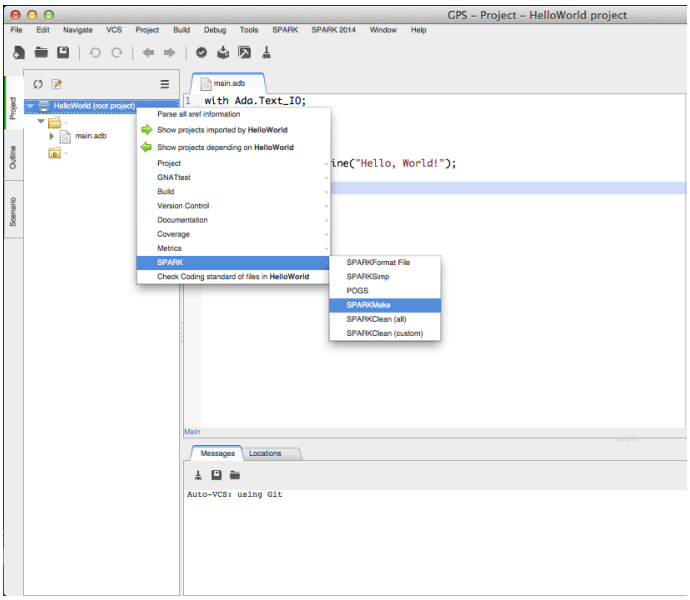

Figure 2.14: Run SPARK Make

input. 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).

Examiner has been used to check syntax and semantics during PCA Pump Prototype development and in verification process described in Chapter 6.

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.

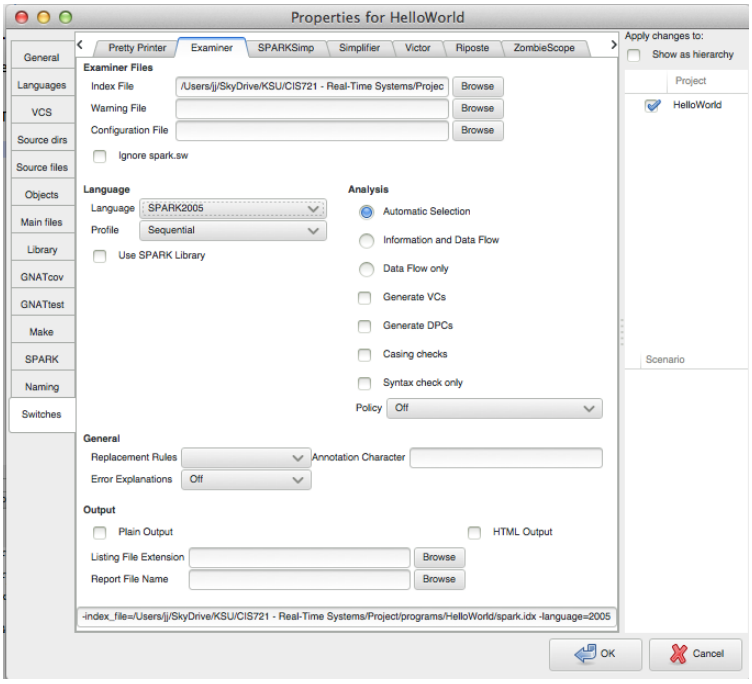

Figure 2.15: Examiner Properties

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

Verification conditions is a set of generated hypothesis, if proven to be true can be concluded that they hold. To generate verification conditions, two kinds of annotations are relevant for Examiner:

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

The notions of pre- and postconditions are based on Hoare logic [HLL+]. More precisely, in the Hoare triple below:

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

`C` is a program that starts in a state satisfying precondition `P`. Program terminates in state satisfying postcondition `Q`. Thus `P` and `Q` are assertions, and `C` is a command (action) performed between them.

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

SPARK functions do not have side effects (as stated in 2.6.1), thus only preconditions are relevant. However, there is annotation `--# return`, which specifies function return value.

Verification Conditions (VCs) are generated depending on commands appearing in the subprogram along path segments. VC generation 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, simplifies and manipulates Verification Conditions (VCs), generated by Examiner, using a number of rules (often referred as rewrite rules). 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 (which contain details about simplification that has been made).

SPARK Simplifier has been used in verification process described in Chapter 6.

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. A program that 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.

ZombieScope has been used for dead paths analysis in verification process described in Chapter 6.

2.6.4 ViCToR

ViCToR is a tool to translate Verification Conditions (VCs), 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.

ViCToR has been used in verification process described in Chapter 6.

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 records the proof steps applied in a `.cmd` file. More details about Proof Checker can be found in chapter 12 of [Bar13].

Proof Checker was not used in this thesis. Instead Bakar Kiasan (see section 2.6.9) has been used.

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 that it finds in a directory tree [Tea10].

SPARKSimp has been used to invoke Simplifier, ZombieScope and ViCToR in verification performed in this thesis (see Chapter 6).

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].

POGS has been used to generate reports for verification performed in this thesis (see Chapter 6).

2.6.8 AUnit

AUnit is a unit test framework for the Ada language. It can be also applied to test SPARK Ada programs. It was inspired by 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 to these related frameworks, it enables simple test cases execution, fixtures, suites, and provides reporting [Fal14].

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 a new project with AUnit tests. The project for which tests are generated is referenced in new generated test 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 an empty (not implemented) test for each subprogram in project. To add/edit/remove tests or rename names, three files have to be edited:

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

Each 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) have to be implemented by hand.

AUnit has been used to create unit test for isolated module of created PCA Pump Prototype (see Section 6.4).

2.6.9 Sireum Bakar

Sireum[34] is a long-term research project conducted by SAnToS Laboratory at Kansas State University. Its goal is to develop an over-arching software analysis platform that incorporates various static analysis techniques such as a 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 an intermediate representation. Any language which can be translated to Pilar can be analyzed by Sireum. For now, there are translators 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

The 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/`.

[34] 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 contracts. Kiasan use symbolic execution technique (Kiasan means "symbolic" in Indonesian). 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 to understand than, e.g., analysis of `.vcg` files generated by SPARK Examiner.

There exists a Kiasan Plug-in for GNAT Programming Studio (GPS). Version 1, for GPS 5, supports SPARK 2005. Version 2, for GPS 6, which supports SPARK 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 the Ravenscar profile. Thus, it can be used only for sequential programs verification. Figure 2.16 presents sample Kiasan analysis result. The Kiasan window in GPS has two parts: (i) a list of units (packages and subprograms), and (ii) analysis cases with pre- and post states. Every unit has the following associated statistics:

- T# - Test cases (expected behavior),
- E# - Exception cases (unexpected behavior),
- Instruction coverage - amount of code that will be executed in execution paths generated by Kiasan analysis,
- Branch coverage - number of branches discovered by Kiasan analysis (0% branch coverage in the case of 100% instruction coverage means that there are no branches in the analyzed unit), and
- Time in which analysis was performed.

After double clicking on some unit, code that 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 the selected (by double clicking) unit. Once some case is selected, code coverage equivalent to this test case is highlighted. Additionally, below the combo box, there are generated execution cases - one for each execution path. The pre-state is listed on the left hand side while the post state is listed on the right hand side. Variables with red font color, in the post-state, are those that are changed as the result of unit execution. Newly created variables (during unit execution) are marked in blue, but there are no such variables in the example presented in Figure 2.16.

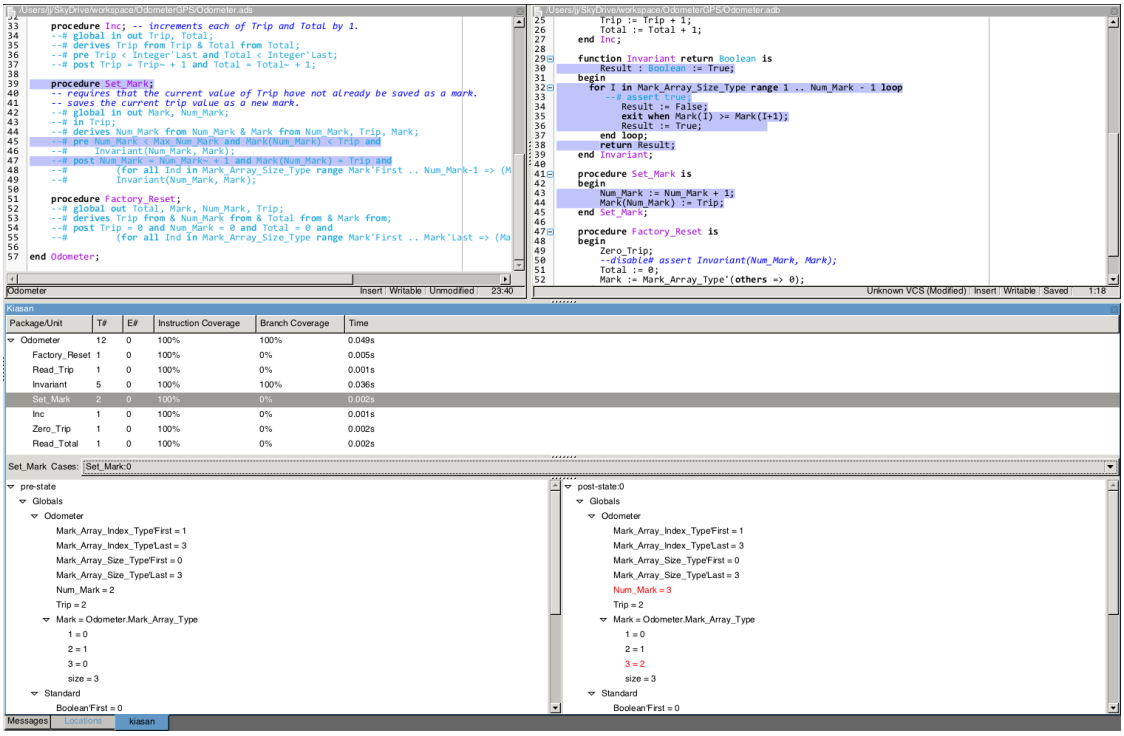

Figure 2.16: Bakar Kiasan report

Bakar Kiasan is useful especially for solving verification issues. It can generate counter examples that give developers greater intuition about problems in the code.

Bakar Kiasan has been used in verification of PCA Pump module (see Section 6.2).

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 analyses in Alir include building the System Dependence Graph (SDG), slicing and chopping on SDG [Thi11].

Bakar Alir has not been used in this thesis, but can potentially be used in the future, to enrich verification process.

2.6.10 GNATprove

GNATprove[35] is a formal verification tool for SPARK 2014 programs, whose input is automatically constructed using GNAT compiler as a front-end. 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 Linux x86, Windows x86 and Linux x86-64.

GNATprove consists of two distinct analyses, flow analysis and proof. Flow analysis checks the correctness of aspects related to data flow (`Global`, `Depends`, `Abstract_State`, `Initializes`, and refinement versions of these), and verifies the initialization of variables. Proof verifies the absence of runtime errors and the correctness of assertions such as `Pre` and `Post` aspects. Using the switch `--mode=<mode>`, whose possible values are `flow`, `prove` and `all`, only one or both

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

of these analyses can be performed (**all** is the default) [AL14b]. GNATprove use Alt-Ergo prover for verification.

GNATprove has been used to verify isolated module of created PCA Pump Prototype, which has been translated to SPARK 2014 (see Section 6.5).

2.7 AADL/BLESS to SPARK Ada code generation

The ultimate goal of the long term research of which this thesis is part is to build an AADL (with BLESS) to SPARK Ada translation. AADL has been used to prototype and fully develop embedded systems for the past 5-7 years [CB09]. Related work in code generation from AADL, but for Java programming language has been done in [PHR]. There are also already existing tools, which performs code generation based on AADL:

- Ocarina
- Ramses

2.7.1 Ocarina

Ocarina [LZPH09] is a tool suite that 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) that supports 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 [CBGP12].

Ocarina has been used as inspirational tool for code generation from AADL models.

2.7.2 RAMSES

RAMSES (Refinement of AADL Models for Synthesis of Embedded Systems) [CBGP12] is a model transformation and code generation tool written in Java. Code generation module produces C code, but does not generate Ada. The approach for code generation is to transform AADL models using a rule-based transformation framework and generate code from transformed (simplified) models. Simplified AADL models contain behavior annex subclauses. RAMSES can be used as OSATE plug-in or standalone application.

RAMSES was initial point of interest, because of its code generation module. However, it has not been used due to its limitation to generate C code only.

3

PCA Pump

"Take risks: if you win, you will be happy; if you lose, you will be wise."

– Unknown

A Patient Controlled Analgesia (PCA) pump[1] is a medical device that allows a 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 administered by a clinician. A continuous infusion mode of the pump (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. A patient can also request additional boluses, but only in specified intervals to

Figure 3.1: Patient Controlled Analgesia (PCA) pump

[1] http://ppahs.org/2012/05/30/patient-controlled-analgesia-pca-pumps-the-basics/

avoid infusion. In addition to basal and patient bolus, clinician can also request a bolus called clinician bolus or square bolus.

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.

A PCA pump is safety-critical device which works in standard process control loop, proposed by Leveson in [Lev12], 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 3.2: Alaris Pump

The PCA pump actuator is a motor that pumps a drug to the patient's vein. The controlled process is dosing the drug. Sensors measure amount of dosed drug. They might be used to double-check if ordered (by controller) that the amount of drug was appropriately delivered. Sometimes there might be some disturbances caused by mechanical issues and environmental conditions. The controller issues appropriate actions based on information from sensors and clinician or patient's commands. A high level overview of PCA Pump is depicted in the Figure 3.4.

One of the problems of using PCA pumps, is that there is inadequate monitoring of patient's blood oxygenation. Nursing staff on general medical units typically track blood

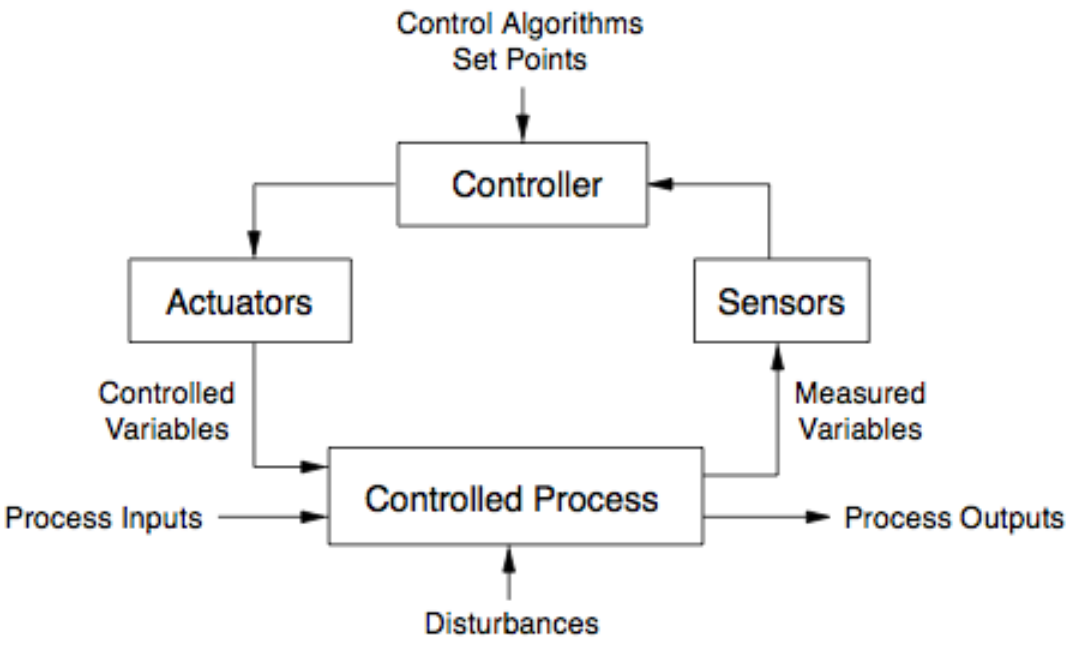

Figure 3.3: Standard Process Control Loop.

oxygenation (SpO_2), heart rate and other vital signs every four hours, which is not enough [OG11]. 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.[2]

Another problem is not adequate resistance to human errors. 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. This caused over infusion that in addition to lack of pulse monitoring resulted in patient's death.[3]

As mentioned in chapter 2, one way to address these problems is through medical devices interoperability. An integrated system can receive input from monitoring devices and disable the pump. In addition, less human error-prone device is needed. It can be assured by using more than one system for their detection.

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

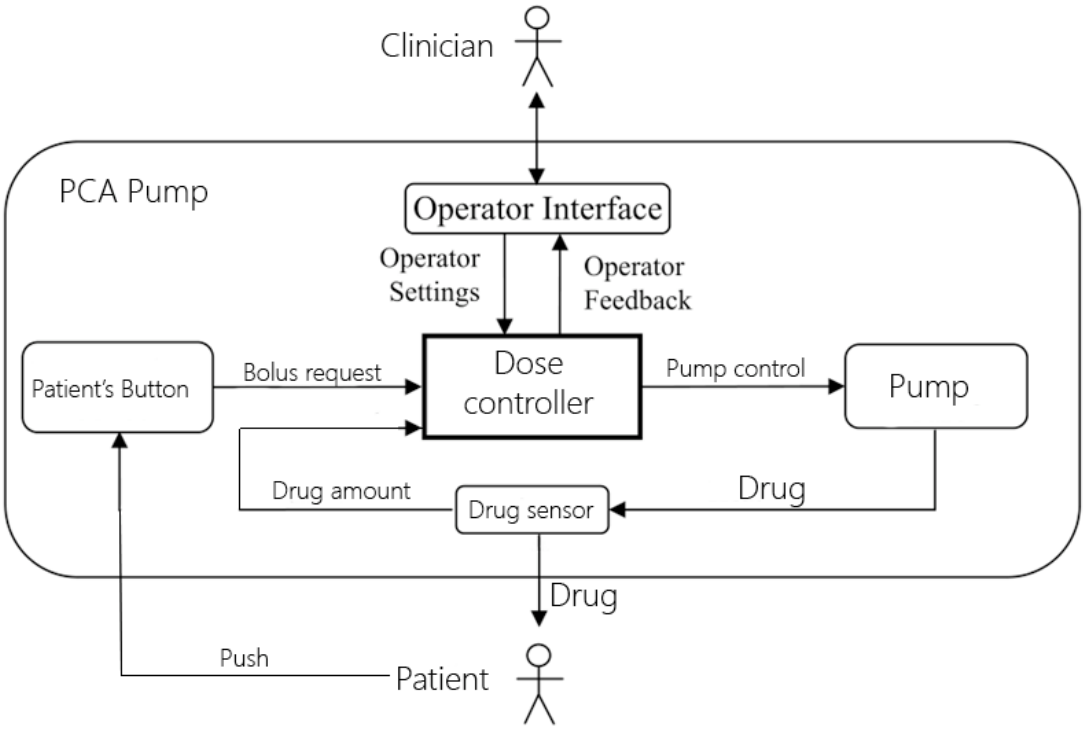

Figure 3.4: PCA Pump system

3.1 PCA Pump Requirements Document

Requirements of "Open Source PCA Pump" [LHC13], on which the work in this thesis is based, are captured in "Open Patient-Controlled Analgesia Infusion Pump System Requirements" document [LH14] created by Brian Larson. The requirements are a rigorously defined 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.

The 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 request from ICE app (based on data from monitoring devices). Such an ICE app could monitor a patient's blood oxygenation and pulse rate, stopping the pump if depressed respiratory function is indicated [LH14].

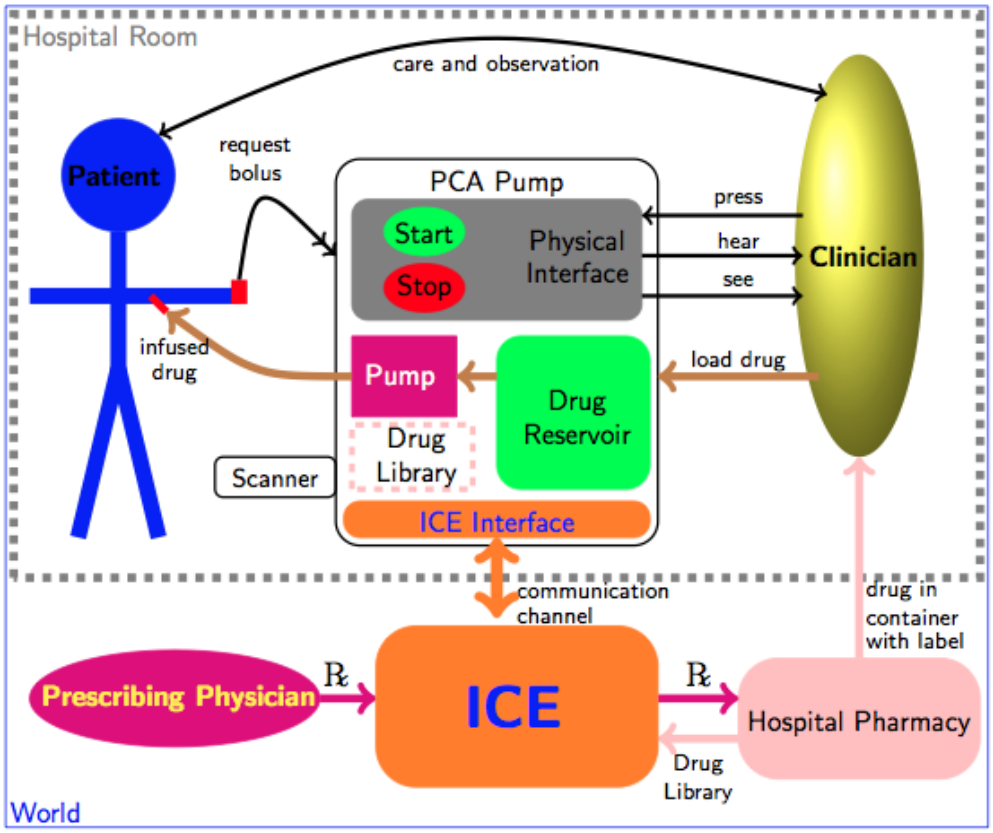

Figure 3.5: Open PCA Pump concept

Additionally, it cooperates with Drug Library, which contains information about drugs and their properties (like concentration). Data needed for pump operation, are captured on 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 [LH14].

The 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 additional dose, equal to VTBI (Volume To Be Infused) 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. KVO is standard mode used in infusion pumps to prevent the vein from closing. Pump switches to KVO rate also when ICE interface requests 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 [LH14], 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 [LH14].

3.2 PCA Pump AADL/BLESS Models

In addition to PCA Pump Requirements Document [LH14], Brian Larson created an AADL model with formal behavioral specifications written in his BLESS framework. The graphical representation of the AADL model is depicted in the Figure 3.6.

The AADL model captures the internal architecture of the device, while BLESS specifications capture its behavior. In Appendix D, thread `Rate_Controller` from the `PCA_Operation` component with BLESS assertions in thread declaration and BLESS behavioral description in thread implementation, is presented. The thread declaration contains input and output ports. Some of them have BLESS assertions attached. These assertions are defined using the BLESS annex in the thread implementation. In addition to assertions, states and transitions defined in thread implementation can potentially be translated into a 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.

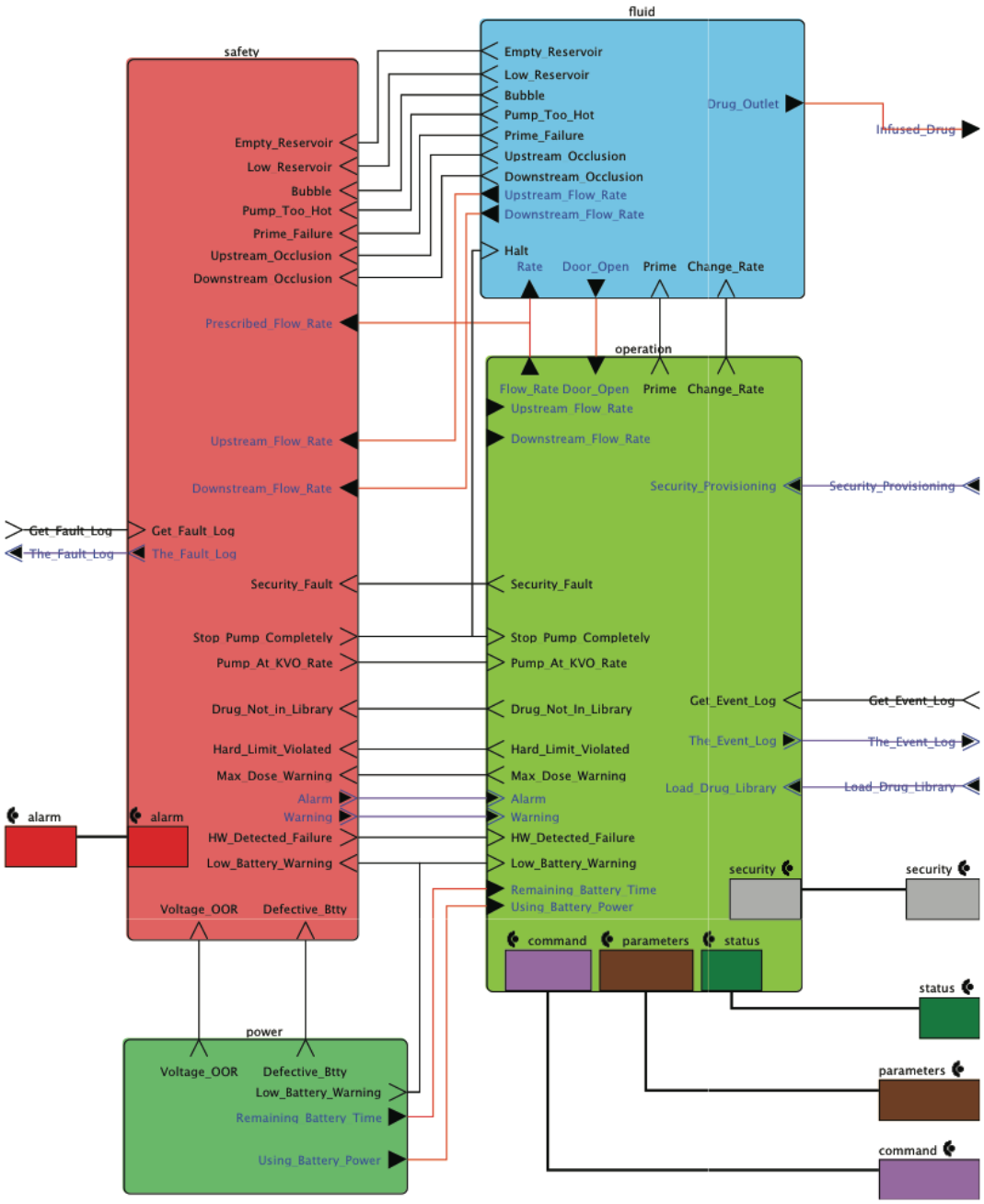

Figure 3.6: Open PCA Pump AADL model

3.3 BeagleBoard-xM

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

BeagleBoard-xM (presented in the Figure 3.7) is an embedded device with an 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.

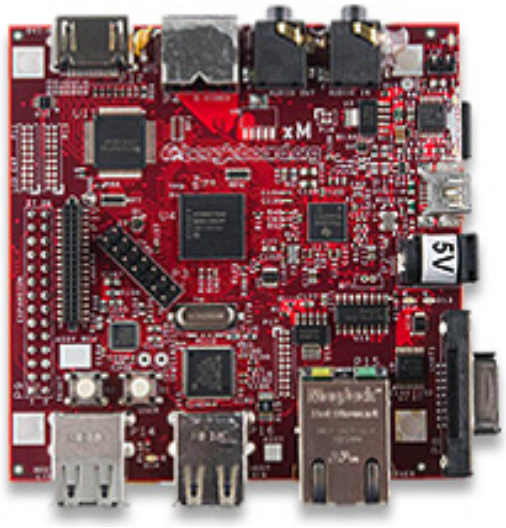

Figure 3.7: BeagleBoard-xM

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 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

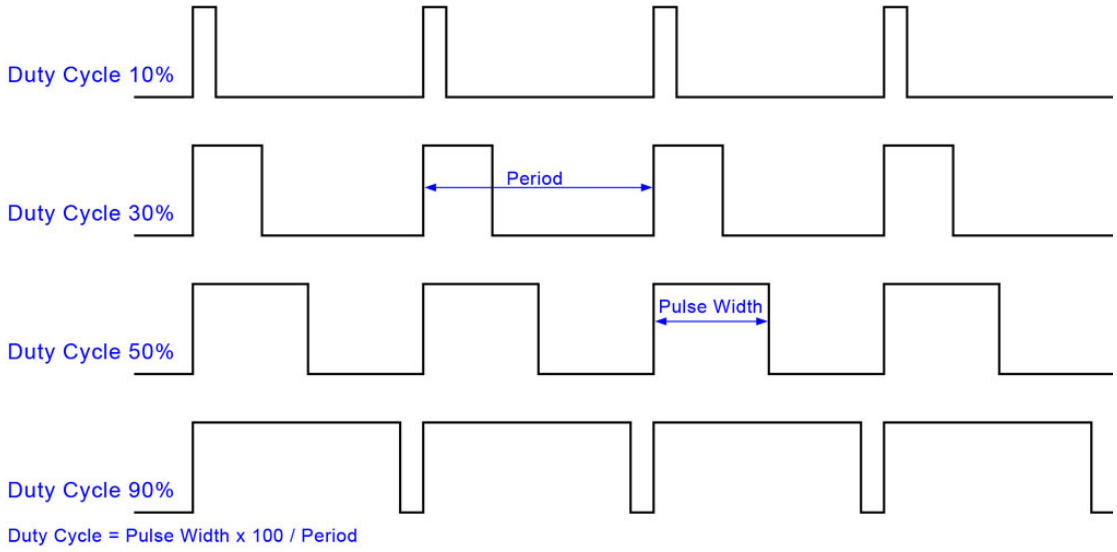

Figure 3.8: An example of PWM duty cycles

was actively developing cross-compiler. They had a working version in 2013, but tested only on their target Android-based device. This version was not working on BeagleBoard-xM platform with Angstrom Linux (configuration used in this thesis). Cooperation with AdaCore, involved bundling and testing a cross-compiler for ARM to produce code for the BeagleBoard-xM, resulted in working cross-compiler. For now, the GNAT cross-compiler works only on Linux 32-bit operating system (as a platform in which cross-compilation has to be performed).

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

4

AADL/BLESS to SPARK Ada Translation

"Don't complain; just work harder."

– *Randy Pausch*

This chapter presents created AADL/BLESS to SPARK Ada translation schemes (4.1), proposed port communication (4.2) and discusses design of an 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])

[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

and its examples[3] were also helpful in understanding of AADL to Ada translation. Mapping of BLESS assertions was created in consultation with Brian Larson (BLESS creator).

4.1.1 Data Types Mapping

One of core AADL packages is `Base_Types`. It defines fundamental data types for AADL. Its definition, without floating and text types, is shown in the Figure 4.1. Every data type has a set of AADL properties (properties are used to define characteristics of an AADL component).

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

The target language for this thesis is SPARK 2005. The SPARK 2014 has been evaluated by thesis author, but determined that, at the time when this thesis was written SPARK 2014 tools were not mature enough and multitasking facilities were not yet included in the language. Types: `Float`, `Character` and `String` are also not part of this thesis, because of the limitations of SPARK 2005 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 the Ravenscar Profile (see section 2.5.3). For every protected type only setter (`Put`) and getter (`Get`) subprograms are defined. The type can be extended by the developer during the development phase. Protected objects can be also removed if they are not needed. The default value for priority, for each generated type is 10. It can be changed during development phase to align with system goals. 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 the Table 4.1.

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

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

Figure 4.1: AADL `Base_Types` package

```
with Interfaces;

package Base_Types is

  type AADL_Boolean is new Standard.Boolean;

  type AADL_Integer is new Standard.Integer;

  type AADL_Natural 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;

end Base_Types;
```

Figure 4.2: Mapping of `Base_Types` for SPARK 2014

Table 4.1: Base AADL types to SPARK mapping.

AADL	SPARK Ada

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

Continued on next page

Table 4.1 – continued from previous page

AADL	SPARK Ada

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

Continued on next page

Table 4.1 – continued from previous page

AADL	SPARK Ada

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

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, the range starts from negative value, for `Unsigned` - from 0. The 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 in the Table 4.2, is straightforward. In addition to simple types, protected types are generated.

Table 4.2: AADL enumeration types to SPARK mapping.

AADL	SPARK Ada

`data Enum_Type`
`  properties`
`  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 that has exactly the same range as an already existing type, especially when it is used for some specific calculations, e.g., measuring speed. Let's say, that `Unsigned_16` was used. Then, during development of next car model, when a larger number of bits are required to hold anticipated values, 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, which is an extension to already existing type in AADL, `extends` clause has to be used. To create, new type, which is based on existing type `Data_Model ::Base_Type` property has to be used. 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

Translation of AADL type created by extension of existing type to SPARK Ada subtype and AADL type created using `Data_Model::Base_Type` property to SPARK Ada derived type is shown in the Table 4.3.

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
  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 the Table 4.4.

Table 4.4: AADL arrays to SPARK Ada mapping

AADL

```
data Some_Array
    properties
      Data_Model::Data_Representation => Array;
      Data_Model::Base_Type => (classifier(
     Base_Types::Integer_32));
      Data_Model::Dimension => (10);
end Some_Array;
```

SPARK Ada

```
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 the Table 4.5.

Table 4.5: AADL struct to SPARK Ada record mapping

AADL	SPARK Ada

`data Some_Record_Type`
`  properties`
`  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, e.g., 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

The proposed ports mapping shown in the 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, i.e., `global` and `derives` annotations to denote global variables usage and variable dependencies. Data types used by ports has to be defined

earlier, to be visible. Moreover, for port communication, protected types are used, to enable concurrency. Simple types are denoted as `Port_Type`, while their protected equivalents as `Port_Type_Store`. Proposed mapping assume single-process application. In order to create distributed system, middle-ware layer has to be created to assure ports communication.

Table 4.6: AADL to SPARK Ada ports mapping.

AADL	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;`

Continued on next page

Table 4.6 – continued from previous page

AADL	SPARK Ada

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;

Port_Name :
  out event port;
	-- spec (.ads)
	procedure Send_Port_Name;
	
	-- body (.adb):
	
	procedure Send_Port_Name
	is
	begin
	  -- TODO: implement sending event
	  -- e.g.:
	  -- Some_Pkg.Put_Port_Name;
	end Send_Port_Name;

Port_Name :
  in event data port Port_Type;
	-- spec (.ads)
	procedure Put_Port_Name(Port_Name_In : Port_Type);
	
	-- body (.adb):
	procedure Put_Port_Name (Port_Name_In : Port_Type)
	is
	begin
	  -- TODO: implement data event handler
	end Put_Port_Name;

Port_Name :
  out event data port Port_Type;
	-- spec (.ads)
	procedure Send_Port_Name;
	
	-- body (.adb):
	procedure Send_Port_Name
	is
	begin
	  -- TODO: implement sending event data
	  -- e.g.:
	  -- Some_Pkg.Put_Port_Name(Port_Name);
	end Send_Port_Name;

4.1.3 Thread to Task Mapping

AADL Threads to SPARK Ada tasks mapping proposed in this thesis is presented in the Table 4.7. Communication between threads is described in Section 4.2.1.

Table 4.7: AADL threads to SPARK Ada tasks mapping.

AADL	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

The mapping of subprograms is also straightforward. However, mapping proposed in this thesis is different than the mapping proposed in "AADL Code Generation Annex" [SCD14]. Flexibility realized by translating appropriate AADL properties is not needed in approach presented in this thesis. Thus **renames** clause is not needed, because it is taken form subprogram name in AADL model. The `Source_Language` property is also not needed, because only one language in targeted (SPARK Ada). For now, the body of subprogram is empty, because behavior (implementation) is not supported by proposed translator. 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	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 the Figure 4.3. Equivalent child packages are shown in the Figure 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 figures 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 the Figure 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;
```

Figure 4.3: Nested packages in SPARK Ada

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

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

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

Figure 4.4: Child packages in SPARK Ada

The thesis author identified a possible approach to create child package and encapsulate one feature group in it. However, SPARK Ada does not allow to access a child package's private part from its parent. Thus, the proposed approach would require to expose feature group internal variables as public, which is undesirable. Thus, a feature group is translated with prefix `Feature_Group_Name_*`. Feature group mapping is presented in Section 4.1.6, in figures 4.5, 4.6 and 4.7. In essence, the feature group is "flatten", i.e., the encapsulation feature of feature groups is removed and elements of feature groups are uniquely identified by using the feature group name as a prefix.

4.1.6 AADL Package to SPARK Ada Package Mapping

Figure 4.5 presents a sample AADL package with a `system` component. It contains all the categories of ports described in Section 4.1.2 as well as 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;
```

Figure 4.5: Sample AADL package with system

For now, only single process SPARK Ada application is considered. Thus, ports are

exposed only on the 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 the ports mapping presented in Section 4.1.2, the translation to a SPARK Ada package is shown in the Figure 4.6 and Figure 4.7.

```
package Some_Pkg
--# own Some_Features_Some_Out_Port : Integer;
--#     Some_Features_Some_In_Port : Integer;
--#     In_Data_Port : Integer;
--#     Out_Data_Port : Integer;
--# initializes Some_Features_Some_Out_Port,
--#             Some_Features_Some_In_Port,
--#             In_Data_Port,
--#             Out_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);

    procedure Send_Out_Event_Data_Port;
end Some_Pkg;
```

Figure 4.6: Translation of sample AADL package from Figure 4.5 - package specification

4.1.7 AADL Property Set to SPARK Ada Package Mapping

In the AADL property set, new properties, types and constants can be defined. There is no equivalent construct 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
  applies to (all);
  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, which indicates the model element(s) to which a property is assigned. It is ignored in the target of the translation. However, future version of the translator might use it, e.g., for automatic generation of **with** clauses or could be translated to comments (to inform developer about modeling assumptions).

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. In such situations, developer 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 in SPARK Ada.

Table 4.10: BLESS to SPARK contracts mapping

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

Continued on next page

Table 4.10 – continued from previous page

AADL/BLESS	SPARK Ada

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;

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;

```
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;

    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
        -- implementation
    end Put_In_Event_Data_Port;

    procedure Send_Out_Event_Data_Port
    is
    begin
        -- implementation
    end Send_Out_Event_Data_Port;

end Some_Pkg;
```

Figure 4.7: Translation of sample AADL package from Figure 4.5 - package body

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 a single-process SPARK Ada application (4.2.1) and concepts of communication between two systems (4.2.2) are presented.

4.2.1 Threads Communication

An example of communication between threads in a single process is depicted in Figure 4.8. There are two threads (`some_thread` and `other_thread`) in one process. The AADL model and its translation to SPARK Ada are presented in the Table 4.11. The connection between threads has to be specified in the process implementation. Based on the mappings from Section 4.1, a protected object is defined, but subprograms are not, because communication takes place only internally. Thus, subprograms are not necessary. The result of translation consists of two tasks and a private global protected object, which enables communication between them. Additionally, both tasks have global annotations (one with **out** mode, other with **in** mode), which indicate the use of a protected object in their bodies.

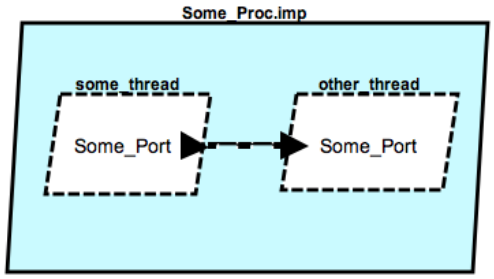

Figure 4.8: Example of port communication between threads

Threads can be also placed in different packages. The same example of two threads within one process, but in different packages is presented in the 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: Translation of AADL threads communication to SPARK Ada

AADL

```
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;
```

SPARK Ada

```
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

```
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	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. Note that no "with" is needed in the first example (Table 4.11), where communication between threads take place in the same package. A modified model of second example, with communication from `Pkg2` to `Pkg1`, is depicted in the Figure 4.9 and presented in the Figure 4.10.

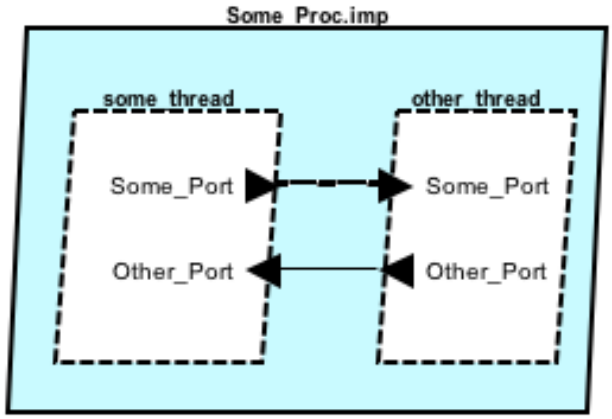

Figure 4.9: Example of two way port communication between threads in different packages

This model, translated to SPARK Ada is presented in the Figure 4.11 and Figure 4.12. 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. Finding an appropriate solution requires further investigation, which is omitted in this thesis.

```
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;
```

Figure 4.10: AADL model of two way port communication threads in different packages

```
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;
```

Figure 4.11: Two way port communication translated to SPARK Ada: package `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;
```

Figure 4.12: Two way port communication translated to SPARK Ada: package `Pkg2TwoWay`

4.2.2 Systems Communication

This Section provides a proposal for handling communication between different systems. An AADL system consists of process(es), and process consists of threads. Ports would be exposed by a package if they are specified in system entity. Communication between two systems can be described by another system. Figure 4.13 presents communication between two systems: panel and pump. AADL model of this system comprises 3 packages: `Main`, `Panel` and `Pump`. They are presented in the figures 4.14, 4.15 and 4.16. The `Panel` package has one thread `Panel_Thread` with two **out** ports: `event port` and `event data port`. Both ports are exposed by the 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`). Connections between these two packages are defined in `Main` package.

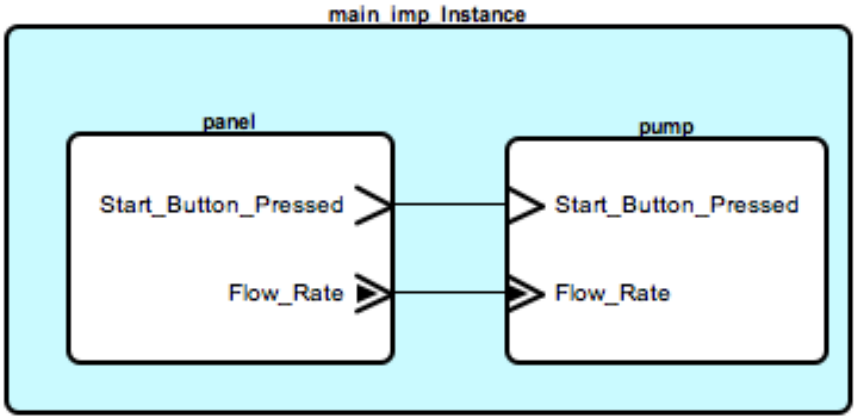

Figure 4.13: Example of port communication between systems

```
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;
```

Figure 4.14: AADL model of port communication between systems: package `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;
```

Figure 4.15: AADL model of port communication between systems: package `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;
```

Figure 4.16: AADL model of port communication between systems: package `Main`

Based on mappings from Section 4.1, conforming SPARK Ada code is presented in the figures 4.17 and 4.18. There are two packages: `Panel` and `Pump`. `Main` package is omitted. Both contain procedures representing port interfaces, according to ports mapping from Section 4.1.2. There is mocked port communication between event data ports. Each package has local variable, which are updated in case of event action. Additionally, procedures responsible for port communication consist appropriate annotations (i.e., `global` and `derives`). Translator should generate this code in case when connection between ports is specified in AADL model. Both packages consist 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 be used to enable not only system to system communication, but also communication with devices. This requires significant effort and was not needed for PCA Pump Prototype created in this thesis, thus it is considered as part of future work described in chapter 8.

```
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;
```

Figure 4.17: Port communication translated to SPARK Ada: package `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;
```

Figure 4.18: Port communication translated to SPARK Ada: package `Pump`

4.3 Towards an Automatic Translator

The ultimate goal is to create translator, which performs translations described in 4.1 and 4.2 automatically. An automatic translator should enable either translation of entire model or parts of the model. An initial implementation strategy might focus on supporting only a 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)

A possible second step would be to introduce BLESS support, specifically, add supported BLESS constructs described in Section 4.1.8:

- assertions for threads
- pre- and postconditions for subprograms

The recommended way to create translator is to parse AADL models, create Abstract Syntax Tree (AST), and emit code using the Visitor pattern. A 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) enables automatic

[4]http://www.antlr.org/
[5]http://tunnelvisionlabs.com/products/demo/antlrworks

AST creation and handles 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 – starting from the translation for the simplest constructs, like data types or single ports, and ending with port communication and BLESS support. First step, would be AADL grammar development. It is recommended to initially specify only the part of required AADL subset and then extend it incrementally. 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. This will help to ensure correctness of translator while working on new features support.

Additionally, the automatic translator should work in two modes:

- Ravenscar: as described above, with protected objects and multiple tasks
- Sequential: single-threaded application, without notion of tasks and protected objects

[6] http://www.eclipse.org/Xtext/index.html
[7] http://www.stringtemplate.org/

5

PCA Pump Prototype Implementation and Code Generation

"Imagination is more important than knowledge.
Knowledge is limited. Imagination encircles the world."

– *Albert Einstein*

This chapter describes running SPARK Ada programs on BeagleBoard-xM platform (3.3), implementation details of PCA pump prototype (5.2)) and code generation from simplified AADL/BLESS models of PCA pump (5.3). All programs presented in this section work the same on an Intel processor (PC or MacBook) and on the BeagleBoard-xM (ARM device).

5.1 Running SPARK Ada Programs on BeagleBoard-xM

To run SPARK Ada program on BeagleBoard-xM, it has to be cross-compiled. As an IDE for SPARK Ada development, GNAT Programming Studio (GPS) is used (see Section 2.5.2). To create a "Hello, World!" application, a new Ada project has been created (choosing

Project/New... from the menu). Then main.adb file, with procedure Main printing "Hello, World!" in standard output, has been added. The code is presented in the Figure 5.1. It is valid Ada 2005 and Ada 2012 code.

```
with Ada.Text_IO;
use Ada.Text_IO;

procedure Main
is
begin
    Put_Line("Hello, World!");
end Main;
```

Figure 5.1: "Hello World" in Ada

The main file has to be always specified in project file (`.gpr`) in order to compile and link the application, which can be runnable. This can be done in Project/Edit Project Properties (Figure 5.2), tab: Main files (Figure 5.3) or directly in project file (`.gpr`).

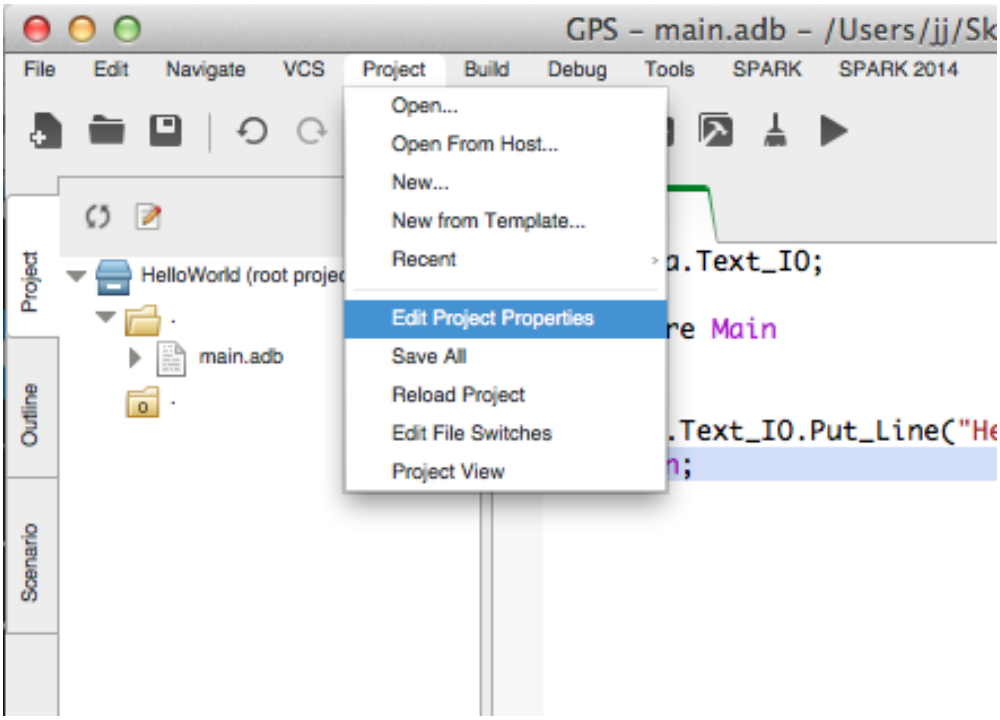

Figure 5.2: Edit Project Properties

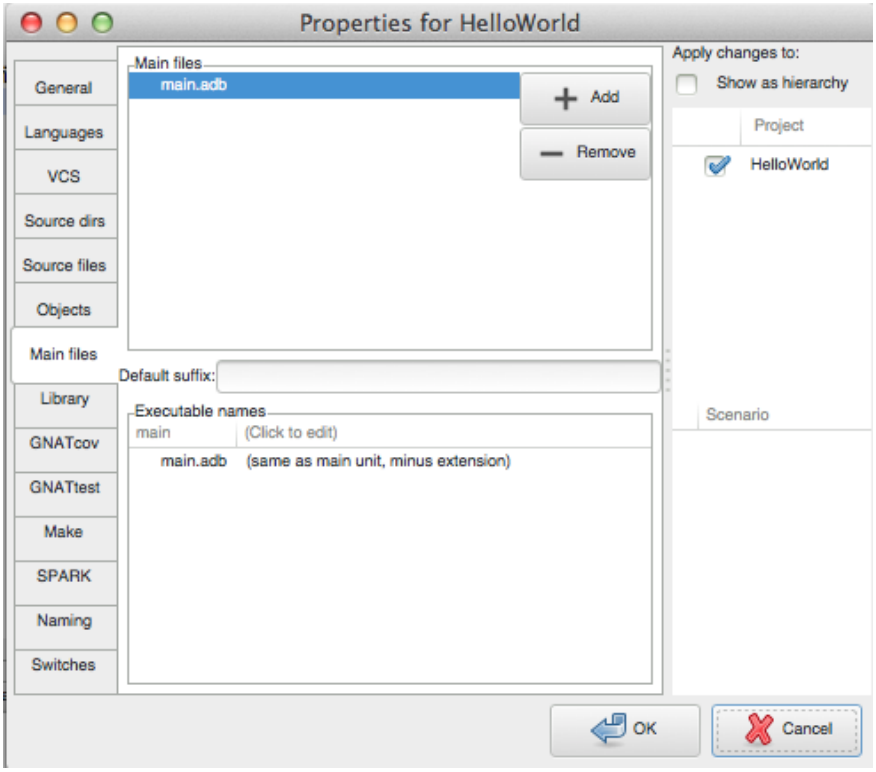

Figure 5.3: Project Main files

To enable cross-compilation, for the current version of cross-compiler, the environmental variable `ENV_PREFIX` has to be set to a directory that contains `/lib` and `/usr` directories. The `/usr` directory should also contain `/usr/lib` and `/usr/include` subdirectories. After these directories are copied into `/home/super/angstrom-arm` directory, the `ENV_PREFIX` is exported with following command: `export ENV_PREFIX=/home/super/angstrom-arm`. The entire project can be compiled and linked with following command: `arm-linux-gnueabi-gnatmake -d -Phelloworld.gpr` (where `helloworld.gpr` is GNAT Programming Studio project file). Additional flags can be specified in the command line or directly in the project file (manually or through GNAT Programming Studio Interface).

A more complex example, which takes advantage of SPARK contracts is presented in Section 5.1.1.

5.1.1 Odometer

The Odometer example is a simple SPARK Ada program, which implements the basic functions of standard odometer. Figure 5.4 shows Odometer in SPARK 2005.

There are 4 subprograms (2 procedures and 2 functions), which are globally available (through other packages and program units):

- `Zero_Trip` procedure - reset Odometer to 0
- `Read_Trip` function - returns current distance
- `Read_Total` function - returns total distance traveled
- `Inc` procedure - increment total and current distance by 1

The given program contains code contracts. Though it does not matter in compilation phase, it is used to illustrate how SPARK verification tools can be applied.

Annotation `global` means that subprogram uses some global variable. This information helps developer to avoid undesired side effects. The `global` annotation has three possible postfixes: (1) **in**, (2) **out** and (3) **in out**, which means that particular variable is read, write and read/write respectively. Annotation `derives` says that some variable value depends on other variables, e.g., in procedure `Inc` variable `Trip` is dependent on its current value (before procedure call). Annotations `pre` and `post` define pre- and postconditions of procedure. We can see that in the `Zero_Trip` procedure, the postcondition requires that variable `Trip` is equal to 0. In procedure `Inc`, postconditions require that variables `Trip` and `Total` are incremented by 1 (tilde appended at the end of variable name is the value of variable when the procedure is called). Annotation `own` exposes private variables for use in specifications for public methods. Annotation `initializes` announce required initialization of the given variables.

In order to test `Odometer` package at runtime, a `Main` procedure has been created. It is presented in the Figure 5.5.

```
package Odometer
--# own
--# Trip,        -- number of meters so far on this trip (can be reset to 0).
--# Total        -- total meters traveled of vehicle since the last factory-reset.
--#   : Natural; -- has range 0 .. Integer'Last.
--# initializes Trip,
--#             Total;
is
    procedure Zero_Trip; -- sets Trip to 0 and clears all saved Trip marks.
    --# global out Trip;
    --# derives Trip from ;
    --# post Trip = 0;

    function Read_Trip return Natural; -- returns value of Trip.
    --# global in Trip;
    --# return Trip;

    function Read_Total return Natural; -- returns value of Total
    --# global in Total;
    --# return Total;

    procedure Inc; -- increments each of Trip and Total by 1.
    --# global in out Trip, Total;
    --# derives Trip from Trip & Total from Total;
    --# pre Trip < Integer'Last and Total < Integer'Last;
    --# post Trip = Trip~ + 1 and Total = Total~ + 1;
end Odometer;

package body Odometer is
    Trip : Natural := 0;
    Total : Natural := 0;

    procedure Zero_Trip is
    begin
        Trip := 0;
    end Zero_Trip;

    function Read_Trip return Natural is
    begin
        return Trip;
    end Read_Trip;

    function Read_Total return Natural is
    begin
        return Total;
    end Read_Total;

    procedure Inc is
    begin
        Trip := Trip + 1;
        Total := Total + 1;
    end Inc;
end Odometer;
```

Figure 5.4: SPARK 2005 code: Odometer

```
with Ada.Text_IO;
with Odometer;

procedure Main
is
begin
    Ada.Text_IO.Put_Line("Trip:  " & Natural'Image(Odometer.Read_Trip));
    Ada.Text_IO.Put_Line("Total: " &  Natural'Image(Odometer.Read_Total));

    Odometer.Inc;

    Ada.Text_IO.Put_Line("Trip:  " & Natural'Image(Odometer.Read_Trip));
    Ada.Text_IO.Put_Line("Total: " &  Natural'Image(Odometer.Read_Total));

    Odometer.Zero_Trip;

    Ada.Text_IO.Put_Line("Trip:  " & Natural'Image(Odometer.Read_Trip));
    Ada.Text_IO.Put_Line("Total: " &  Natural'Image(Odometer.Read_Total));

    Odometer.Inc;

    Ada.Text_IO.Put_Line("Trip:  " & Natural'Image(Odometer.Read_Trip));
    Ada.Text_IO.Put_Line("Total: " &  Natural'Image(Odometer.Read_Total));
end Main;
```

Figure 5.5: Main procedure for `Odometer` package

Odometer in SPARK 2005 works fine on the BeagleBoard-xM using the cross compilation techniques introduced in the previous section. In order to test a SPARK 2014 version of the program, SPARK 2005 annotations have been converted into Ada 2012 contracts. Figure 5.6 presents Odometer in SPARK 2014.

Odometer example was created to check possible limitations and issues related to different platform (ARM-based). No limitations were found.

5.1.2 Multitasking Applications

In Ada World, concurrency is referred as tasking, and the task is the same construct as the thread in other programming languages. In Section 5.1.1, a single-tasking application was tested. This section presents a simple Ada a multitasking application and multitasking version of Odometer in SPARK 2005 from Section 5.1.1. Both applications compile correctly and work as expected on BeagleBoard-xM platform.

```
package Odometer
with Abstract_State => (Trip_State, Total_State)
is
   function Trip_State return Integer with Convention => Ghost, Global => (Input => Trip_State);
   function Total_State return Integer with Convention => Ghost, Global => (Input => Total_State);
   procedure Zero_Trip with Global => (Output => (Trip_State)), Depends => (Trip_State => null), Post =>
     Trip_State = 0;
   function Read_Trip return Natural  with Global => (Input => Trip_State),
      Post => Read_Trip'Result = Trip_State;
   function Read_Total return Natural with Global => (Input => Total_State),
      Post => Read_Total'Result = Total_State;
   procedure Inc with Global => (In_Out => (Trip_State, Total_State)),
      Depends => (Trip_State => Trip_State, Total_State => Total_State),
      Pre => Trip_State < Integer'Last and Total_State < Integer'Last,
      Post => Trip_State = Trip_State'Old + 1 and Total_State = Total_State'Old + 1;
end Odometer;

package body Odometer
with Refined_State => (Trip_State => (Trip), Total_State => (Total))
is
   Trip : Natural;
   Total : Natural;

   function Trip_State return Integer
     with Refined_Global => (Input => Trip) is begin
      return Trip;
   end Trip_State;

   function Total_State return Integer
     with Refined_Global => (Input => Total) is begin
      return Total;
   end Total_State;

   procedure Zero_Trip
     with Refined_Global => (Output => Trip), Refined_Depends => (Trip => null) is begin
      Trip := 0;
   end Zero_Trip;

   function Read_Trip return Natural
     with Refined_Global => (Input => Trip) is begin
      return Trip;
    end Read_Trip;

   function Read_Total return Natural
     with Refined_Global => (Input => Total) is begin
      return Total;
   end Read_Total;

   procedure Inc
     with Refined_Global => (In_Out => (Trip, Total)), Refined_Depends => (Trip => Trip, Total => Total)
   is begin
      Trip := Trip + 1;
      Total := Total + 1;
   end Inc;
end Odometer;
```

Figure 5.6: SPARK 2014 code: Odometer

Ada Multitasking Application

Figure 5.7 presents a simple Ada 2005 multitasking application that prints numbers in different time intervals. It is also valid code for Ada 2012. There are 3 tasks:

- Main task
- S (type: `Seconds`) - simple counter printing numbers form 1 to 10 in every second
- T (type: `Tenth_Seconds`) - simple counter printing numbers from 0.1 to 10 in every 0.1 second

```
with Ada.Text_IO;
use Ada.Text_IO;
with Ada.Float_Text_IO;

procedure Main is
    task type Seconds is
    end Seconds;

    task type Tenth_Seconds is
    end Tenth_Seconds;

    S : Seconds;
    T : Tenth_Seconds;

    task body Seconds is
    begin
        for I in 1..10 loop
            delay Standard.Duration(1);
            Put_Line(Integer'Image(I));
        end loop;
    end Seconds;

    task body Tenth_Seconds is
    begin
        for I in 1..100 loop
            delay 0.1;
            Ada.Float_Text_IO.Put(Float(I)/Float(10), AFT=>2, EXP=>0);
            Put_Line("");
        end loop;
    end Tenth_Seconds;
begin
    Put_Line("Started");
end Main;
```

Figure 5.7: Simple multitasking application in Ada

The program works as expected on BleagleBoard-xM. This is not a valid SPARK program though. As mentioned in Section 2.5.3, tasks can be declared only in packages. Not in subprograms or in other tasks [Bar13].

SPARK Ada multitasking application

As mentioned in Section 2.5.3, in SPARK 2005 multitasking is possible with Ravenscar Profile. Default profile - sequential - does not enable tasking. In other words, SPARK 2005 tools cannot analyze and reason about programs if Ravenscar profile flag is not provided. In SPARK 2014 - for now tasking is not possible. It's part of SPARK 2014 road map to include support for tasking in the future. Thus, only the SPARK 2005 application was tested.

The tested, multitasking application is an extended version of Odometer (presented in the Figure 5.4). It has additional variable `Speed`, procedure `Set_Speed` and new task: `Drive`. Thus, in total it has two tasks:

- Main
- Drive

The `Drive` task increase `Total` and `Trip` variables by `Speed` (m/s) in every second. Extended Odometer is presented in Figure 5.8 and 5.9.

There are two ways to access protected variable in task body:

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

Protected variables may not be used in proof contexts. Thus, if we try to use protected variable in proofs (pre- or postcondition), then we get semantic error: `Trip is a protected own variable`. To preserve pre- and postconditions from original Odometer, atomic types (`Distance` and `Meters_Per_Second`) has been used. The capability to specify pre- and postconditions has been preserved, but now the application is not thread safe.

```
--# inherit Ada.Real_Time;
package Odometer
--# own Trip : Distance;
--#     Total : Distance;
--#     Speed : Meters_Per_Second;
--#     task d : Drive;
--# initializes Trip,
--#             Total,
--#             Speed;
is
    type Distance is range Natural'First .. Natural'Last;
    pragma Atomic (Distance);

    type Meters_Per_Second is range Natural'First .. Natural'Last;
    pragma Atomic(Meters_Per_Second);

    procedure Zero_Trip; -- sets Trip to 0 and clears all saved Trip marks.
    --# global out Trip;
    --# derives Trip from ;
    --# post Trip = 0;

    function Read_Trip return Distance; -- returns value of Trip.
    --# global in Trip;
    --# return Trip;

    function Read_Total return Distance; -- returns value of Total
    --# global in Total;
    --# return Total;

    procedure Inc; -- increments each of Trip and Total by 1.
    --# global in out Trip, Total;
    --# derives Trip from Trip & Total from Total;
    --# pre Trip < Distance'Last and Total < Distance'Last;
    --# post Trip = Trip~ + 1 and Total = Total~ + 1;

    procedure Set_Speed(New_Speed : Meters_Per_Second);
    --# global out Speed;
    --# derives Speed from New_Speed;
    --# post Speed = New_Speed;
private
    task type Drive
    --# global in     Speed;
    --#        in out Trip;
    --#        in out Total;
    --#        in     Ada.Real_Time.ClockTime;
    is
        pragma Priority(10);
    end Drive;
end Odometer;
```

Figure 5.8: Multitasking Odometer specification

```
with Ada.Real_Time;
use type Ada.Real_Time.Time;
package body Odometer is
    Trip : Distance := 0;
    Total : Distance := 0;
    Speed : Meters_Per_Second := 0;
    d : Drive;

    procedure Zero_Trip is
    begin
        Trip := 0;
    end Zero_Trip;

    function Read_Trip return Distance is
    begin
        return Trip;
    end Read_Trip;

    function Read_Total return Distance is
    begin
        return Total;
    end Read_Total;

    procedure Inc is
    begin
        Trip := Trip + 1;
        Total := Total + 1;
    end Inc;

    procedure Set_Speed(New_Speed : Meters_Per_Second)
    is
    begin
        Speed := New_Speed;
    end Set_Speed;

    task body Drive
    is
        Release_Time : Ada.Real_Time.Time;
        Period : constant Integer := 1000; -- update in every second
    begin
        loop
            Release_Time := Ada.Real_Time.Clock + Ada.Real_Time.Milliseconds(Period);
            delay until Release_Time;
            -- each time round, increase Trip and Total
            for I in Meters_Per_Second range 0 .. Speed loop
                Inc;
            end loop;
        end loop;
    end Drive;
end Odometer;
```

Figure 5.9: Multitasking Odometer body

5.1.3 Controlling PCA Pump Actuator

PCA pump prototype created as part of this thesis interacts with external device (physical pump) through General-purpose input/output (GPIO) pin. To control the pump, Pulse width modulation (described in 3.3) is used. BeagleBoard-xM has 28 GPIO pins. Three of them are PWM enable (pin 4 - mapped as `GPIO_144`, pin 6 - `GPIO_146` and pin 10 - `GPIO_145`). All of these pins allow to control external device by specifying frequency and duty cycle. However it requires PWM driver.[1] PWM can be also simulated manually. To run the pump, pin has to be turned on and off with specified frequency. In order to do that, a `sleep` function can be used.

GPIO ports interact with the BeagleBoard platform through memory maps. This means that turning particular pin on or off is achieved by writing values into a memory segment associated with the pin. Memory segment is further mapped into file system. Memory maps are synchronized via continuous refresh loops.

Pin, used for controlling PCA pump, is the pin 14 (mapped as `GPIO_162`). It is mapped into directory `/sys/class/gpio/gpio162/`. To turn pin on, file `/sys/class/gpio/gpio162/value` has to contain '1'. To turn it off - '0'. Pump is also connected to ground (GND). For that purpose pin 28 is used. Figure 5.10 shows simple bash script, which turns pin on and off every second. Before the pin can be used, it has to be opened by writing pin mapping number (in this case: 162 for pin 14) into `/sys/class/gpio/export` file. When communication is over, connection should be closed with writing the same value to file `/sys/class/gpio/unexport`. Setting 'high' (1) for 1 second and 'low' (0) also for 1 second gives 50% duty cycle.

Initial tests of interaction with pump actuator has been made in bash and Java, because it does not require cross-compilation. The bash script runs natively on Angstrom Linux. The Java application runs on the JVM distribution for Angstrom.

[1] http://beagleboard.org/project/PWM+driver+for+Beagle+Board/

```
#!/bin/sh

if [ $# = 0 ]
then
  GPIO=162
else
  GPIO=$1
fi

cleanup() {
  echo $GPIO > /sys/class/gpio/unexport
  exit
}

trap cleanup SIGINT

echo $GPIO > /sys/class/gpio/export
echo "out" > /sys/class/gpio/gpio$GPIO/direction

while [ "1" = "1" ]; do
  echo "1" > /sys/class/gpio/gpio$GPIO/value
  sleep 1
  echo "0" > /sys/class/gpio/gpio$GPIO/value
  sleep 1
done

cleanup
```

Figure 5.10: Turning pin on and off in bash

BeagleBoard-xM with Linux Angstrom allows to install software packages using package manager `opkg`.[2] Packages feeds can be found on `http://feeds.angstrom-distribution.org/feeds` and set in `.conf` files in `/etc/opkg` directory. In this thesis version 2012.01 of Angstrom (with Linux 3.0.14+) has been used and the following feeds:

- `base-feed.conf: src/gz base http://feeds.angstrom-distribution.org/feeds/v2012.05/ipk/eglibc/armv7a/base`
- `beagleboard-feed.conf: src/gz beagleboard http://feeds.angstrom-distribution.org/feeds/v2012.05/ipk/eglibc/armv7a/beagleboard`
- `debug-feed.conf: src/gz debug http://feeds.angstrom-distribution.org/feeds/v2012.05/ipk/eglibc/armv7a/debug`
- `gstreamer-feed.conf: src/gz gstreamer http://feeds.angstrom-distribution.org/feeds/v2012.05/ipk/eglibc/armv7a/gstreamer`
- `noarch-feed.conf: src/gz no-arch http://feeds.angstrom-distribution.org/feeds/v2012.05/ipk/eglibc/armv7a/all`
- `perl-feed.conf: src/gz perl http://feeds.angstrom-distribution.org/feeds/v2012.05/ipk/eglibc/armv7a/perl`
- `python-feed.conf: src/gz python http://feeds.angstrom-distribution.org/feeds/v2012.05/ipk/eglibc/armv7a/python`

Once, feeds are set, it is recommended to update list of available packages with command: `opkg update`. To update all installed packages, following command has to be used: `opkg upgrade`. To install Java runtime-environment (JVM), the following command can be used: `opkg install openjdk-6-java`. Java Development Kit, which contains Java compiler and allows to compile Java programs on BeagleBoard, can be installed with: `opkg install openjdk-6-jdk`.

A program similar to the bash script presented in Figure 5.10, but working for 20 seconds and terminating, written in Java, is presented in Figure 5.11.

[2] http://wiki.openwrt.org/doc/techref/opkg

```
import java.io.*;

public class PcaMain {
  public static void main(String[] args) throws IOException, InterruptedException {
    final String GPIO = "162";
    final String BASE_DIR = "/sys/class/gpio";
    WriteToFile(BASE_DIR+"/export", GPIO);
    WriteToFile(BASE_DIR+"/gpio"+GPIO+"/direction", "out");
    for(int i=0; i<10; ++i) {
            WriteToFile(BASE_DIR+"/gpio"+GPIO+"/value", "1");
            Thread.sleep(1000);
      WriteToFile(BASE_DIR+"/gpio"+GPIO+"/value", "0");
            Thread.sleep(1000);
    }
    WriteToFile(BASE_DIR+"/unexport", GPIO);
  }

  public static void WriteToFile(String filename, String content) throws IOException {
    File file = new File(filename);
    if (!file.exists()) {
      file.createNewFile();
    }
    PrintWriter writer = new PrintWriter(filename, "UTF-8");
    writer.println(content);
    writer.close();
  }
}
```

Figure 5.11: Turning pin on and off in Java

The extended program from Figure 5.11, with procedures to start and stop the pump, written in Ada, is presented in Figure 5.12 and 5.13.

```
with Ada.Real_Time; use type Ada.Real_Time.Time;
package Pca_Pump is
   procedure Start_Pump;
   procedure Stop_Pump;
   procedure Run_Pump(N: in Integer);
   procedure Write_Signal(Signal: in Integer);
end Pca_Pump;
```

Figure 5.12: Simple pump in Ada: package specification

```
with Ada.Strings.Unbounded; use type Ada.Strings.Unbounded;
with Ada.Text_IO.Unbounded_IO; use type Ada.Text_IO;
package body Pca_Pump is
   procedure Start_Pump is
      F    : Ada.Text_IO.File_Type; Data : Unbounded_String := To_Unbounded_String("pumping");
      File_Export : Ada.Text_IO.File_Type;
      File_Direction : Ada.Text_IO.File_Type;
   begin
      Create(File_Export, Ada.Text_IO.Out_File, "/sys/class/gpio/export");
      Put_Line(File_Export, "162");
      Close(File_Export);
      Create(File_Direction, Ada.Text_IO.Out_File, "/sys/class/gpio/gpio162/direction");
      Put_Line(File_Direction, "out");
      Close(File_Direction);
      Create(F, Ada.Text_IO.Out_File, "/home/root/pump_status.txt");
      Unbounded_IO.Put_Line(F, Data); Put_Line("Pumping...");
      Close(F);
   end Start_Pump;

   procedure Stop_Pump is
      F    : Ada.Text_IO.File_Type; Data : Unbounded_String := To_Unbounded_String("IDLE");
      File_Unexport : Ada.Text_IO.File_Type;
   begin
      Create(File_Unexport, Ada.Text_IO.Out_File, "/sys/class/gpio/unexport");
      Put_Line(File_Unexport, "162");
      Close(File_Unexport);
      Create(F, Ada.Text_IO.Out_File, "/home/root/pump_status.txt");
      Unbounded_IO.Put_Line(F, Data); Put_Line("Stopped");
      Close(F);
   end Stop_Pump;

   procedure Run_Pump(N: in Integer) is
      Interval: constant Ada.Real_Time.Time_Span := Ada.Real_Time.Milliseconds(100);
      Next_Time: Ada.Real_Time.Time;
   begin
      Next_Time := Ada.Real_Time.Clock;
      Start_Pump;
      for I in Integer range 1 .. N*1000 loop
     Next_Time := Next_Time + Interval; Write_Signal(1); delay until Next_Time;
         Next_Time := Next_Time + Interval; Write_Signal(0); delay until Next_Time;
      end loop;
      Stop_Pump;
   end Run_Pump;

   procedure Write_Signal(Signal : in Integer) is
      Filename  : String := "/sys/class/gpio/gpio162/value";
      File : Ada.Text_IO.File_Type; Data : Unbounded_String;
   begin
      Ada.Text_IO.Open (File => File, Mode => Ada.Text_IO.Out_File, Name => Filename);
      if Signal = 1 then Data := To_Unbounded_String("1");
      else Data := To_Unbounded_String("0");
      end if;
      Unbounded_IO.Put_Line(File, Data);
      Ada.Text_IO.Close(File);
   end Write_Signal;
end Pca_Pump;
```

Figure 5.13: Simple pump in Ada: package body

5.2 Implementation Based on Requirements Document and AADL Models

In order to confirm that implementation of PCA Pump, specified in Requirements Document, is feasible on BeagleBoard-xM, a simple PCA pump prototype has been created. The implemented prototype is multitasking application (using Ravenscar profile) running on BeagleBoard-xM. The base for implementation was `Pca_Operation` package. Only two AADL threads are implemented: `Rate_Controler` and `Max_Drug_Per_Hour_Watcher`. Thus, the pump has three tasks in total:

- main task - interface for controlling and monitoring the pump
- `Rate_Controller` - control the speed of infusion rate
- `Max_Drug_Per_Hour_Watcher` - control over infusion

The first step was to translate types required by operation module. Strings and float types were skipped to keep verification simple (using only integer types and its subtypes). Besides that, all types from following packages are translated:

- `Base_Types`
- `Bless_Types`
- `Ice_Types`
- `Pca_Types`

The Open PCA pump, according to requirements document [LH14], has 5 operational modes:

- Stopped: $F = 0ml/hr$
- Keep Vein Open (KVO): $F = 0.1ml/hr$
- Basal infusion: $F = F_{Basal}$

- Patient bolus: $F = F_{Basal} + F_{Bolus}$
- Clinician bolus: $F = F_{Basal} + F_{Square_Bolus}$ (square bolus is calculated value: VTBI divided by the duration chosen by the clinician)

The requirements document does not specify implementation details. One of implementation decisions, which had to be made, was to decide how basal infusion will work. One solution was to run actuator continuously on speed calculated based on current flow rate. Another solution was to dose drug in 0.1 ml increments. This is how CADD-Prizm Ambulatory Infusion Pump [Med10] works, and this implementation was chosen. It allows for easier bolus monitoring and calculations. The pump engine controller is in a separate module. It is written in Ada, so it will not be verified with SPARK tools. Using increments, instead of continuous speed allows to issue requests of 0.1 ml dose to the engine module, and it is its responsibility to deliver requested amount of dose. Performing calculations based on speed changes would be much more complicated. For monitoring, amount of drug dosed in last hour (to guard against over infusion), array with size = (60 * 60) has been created. Its elements represents all seconds of last hour. Last element is incremented once request to the engine is issued. This is done in `Rate_Controller` task. `Max_Drug_Per_Hour_Watcher` checks dosed amount by summing all elements. It also shifts the array in every second, so doses older than 1 hour are not take into account anymore.

To avoid using floating point types, internal calculations are in micro liters: 1 micro liter (μl) = 0.001 ml, thus 1 ml = 1000μl.

In real-world applications that use SPARK, 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 or being separated entities on which SPARK tools are not run. `Pca_Engine` package is separated entity, which control the pump actuator. It uses Ada features not present in SPARK, thus it is not verified by SPARK tools.

The implemented PCA pump prototype is a console Ada application with a textual interface, which has following functionalities:

- Entering prescription, which comprises of following parameters:
  - Basal flow rate
  - Volume to be infused (VTBI) during patient or clinician bolus
  - Maximum dose of drug allowed per hour
  - Minimum time between patient's boluses
- Starting the pump
- Stopping the pump
- Monitoring drug dosed in last hour: when maximum allowed dose is exceeded, it switches pump state to KVO rate
- Performing patient bolus:
  - if bolus request too soon (faster than minimum time between bolus) then it is ignored
  - if bolus is requested during clinician bolus, then clinician bolus is paused and patient bolus starts; once patient bolus is done, pump switches back to clinician bolus
- Performing clinician bolus (time has to be specified):
  - bolus requested during previously requested (not finished) clinician bolus is ignored
  - bolus requested during patient bolus is performed right after patient bolus is done

The code of implemented PCA Pump Prototype, along with mapped types, is attached in Appendix B.

5.3 Code Translation from AADL/BLESS Models

The original AADL/BLESS models were simplified and truncated to demonstrate sample translation. 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. Simplified AADL/BLESS models can be found in Appendix E. The translation was performed based on translation schemes from chapter 4. Appendix F contains translated PCA pump code.

Raw, translated code cannot be verified with SPARK tools, because it contains unimplemented parts. One example is the code resulting from translation from BLESS assertions, which are defined but not implemented in models. Once these missing parts will be implemented, code can be verified.

6

Verification

"It had long since come to my attention that people of accomplishment rarely sat back and let things happen to them. They went out and happened to things."

– Leonardo da Vinci

The goal of verification process presented in this chapter is to check for run-time errors and show by example how to fix them with the SPARK verification tools. In the future, the same strategy can be used for verification of requirements specified by BLESS annexes in AADL models. As a reminder to the reader, the SPARK 2005 has been identified (as opposed to SPARK 2014, which is still under development) as the most appropriate and capable for the development and verification needs of this thesis work (at the time when this thesis has been written).

The strategy for Software Verification using SPARK 2005 tools is as follows [Bar13]. First, Examiner generates Verification Conditions (VCs) and Dead Path Conjectures (DPCs). Some VCs that can be discharged by simple rewriting are also simplify and discharged by Examiner. Next, SPARKSimp runs Simplifier to simplify and discharge some (or all) VCs that were not discharged by Examiner. SPARKSimp also runs ZombieScope to analyze DPCs and ViCToR to discharge VCs (not discharged by Examiner nor Simplifier) with

SMT Solver technology. To provide a summary of verification results, a POGS report is generated. To this standard SPARK 2005 tool chain, Bakar Kiasan symbolic execution tools (developed by the Kansas State University SAnToS research group) has been added. Specifically, when not all Verification Conditions are discharged, analysis continues with Bakar Kiasan. After fixes are made with Kiasan help, Examiner and SPARKSimp tools are run again to confirm correctness. This approach is presented in the Figure 6.1. Detailed overview of SPARK verification tools can be found in chapter 12 of SPARK book [Bar13].

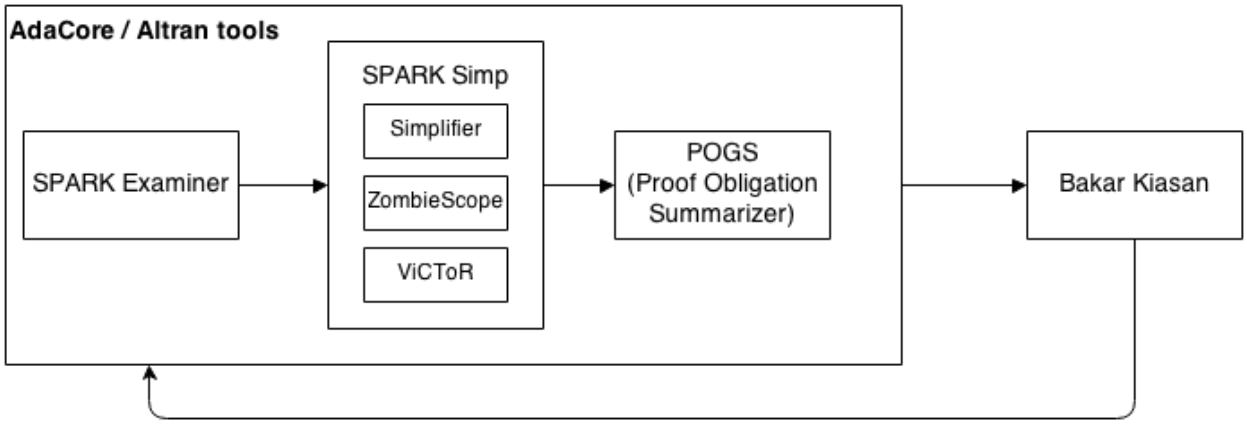

Figure 6.1: Applied Verification strategy

6.1 Verification of Implemented PCA Pump Prototype

During PCA Pump Prototype implementation, program syntax was regularly checked with SPARK Examiner. The complete, manually implemented prototype, which can be found in Appendix B, was verified with the strategy given at the beginning of this chapter (excluding Bakar Kiasan, which does not handle Ravenscar programs). Thus SPARK Examiner, SPARKSimp (Simplifier, ZombieScope and ViCToR) and POGS were used. Package `Pca_Engine` was excluded from verification, using `--# hide` annotation, because it contains Ada code, which is non-valid SPARK. The result of this analysis, in the form of a POGS report

summary, is presented in the Figure 6.2. The full report can be found in Appendix C.

The POGS report shows that 30% (90) of VCs were discharged by Examiner and 61% (183) by Simplifier. There are 29 undischarged VCs. All of them are caused by possible overflows and array index out of bounds. In addition to VCs, DPCs were generated and 32 dead paths were found. Some undischarged VCs and dead paths come from procedures responsible for maximum dose monitoring. As mentioned in chapter 2.6.9, Bakar Kiasan does not support Ravenscar profile. For the demonstration purpose, sequential module for dose monitoring has been created in order to analyze undischarged VCs. Verification process of this module is described in Section 6.2.

6.2 Monitoring Dosed Amount

This section is a case study of verifying the SPARK module responsible for tracking the dosed amount of drug. The module was created in the sequential SPARK 2005 profile, based on implemented PCA prototype presented in Appendix B. The isolated module implementation is presented in the Figure 6.3.

Verification strategy is based on Figure 6.1. First, the program is verified with Examiner, SPARKSimp (Simplifier, ZombieScope and Victor). A Verification report is generated by POGS. In case of any unfinished verification steps, verification is continued with Bakar Kiasan, which gives more user friendly experience that POGS report and generated VC files. It may be preferable to use Bakar Kiasan first, but in this thesis SPARK 2005 verification tools created by AdaCore and Altran were used first to indicate not verified code.

First verification report generated by POGS is presented in the Figure 6.4. It indicates presence of three undischarged (not proved) VCs.

Next, according to verification strategy, instead of VC analysis Bakar Kiasan was run to find out why program is not fully verified. Kiasan report is presented in the Figure 6.5.

```
Summary:

Proof strategies used by subprograms
------------------------------------------------------------------------
Total subprograms with at least one VC proved by examiner:             15
Total subprograms with at least one VC proved by simplifier:           20
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:                  14
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:                                        14
Total subprograms with at least one undischarged VC:                    8  <<<
Total subprograms with at least one false VC:                           0
                                                                       -----
Total subprograms for which VCs have been generated:                   22


ZombieScope Summary:
------------------------------------------------------------------------
Total subprograms for which DPCs have been generated:                  22
Total number subprograms with dead paths found:                         3
Total number of dead paths found:                                      32


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            93         80         12          1
Precondition           12          0         12          0
Check stmnt.            0          0          0          0
Runtime check         187          0        159         28
Refinem. VCs           10         10          0          0
Inherit. VCs            0          0          0          0
============================================================
Totals:                302         90        183         29 <<<
%Totals:                          30%        61%        10%

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

Figure 6.2: Summary of POGS report for PCA Pump prototype

```
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;

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;
```

Figure 6.3: Dose monitor module specification

```
Summary:

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 ========================
```

Figure 6.4: POGS report

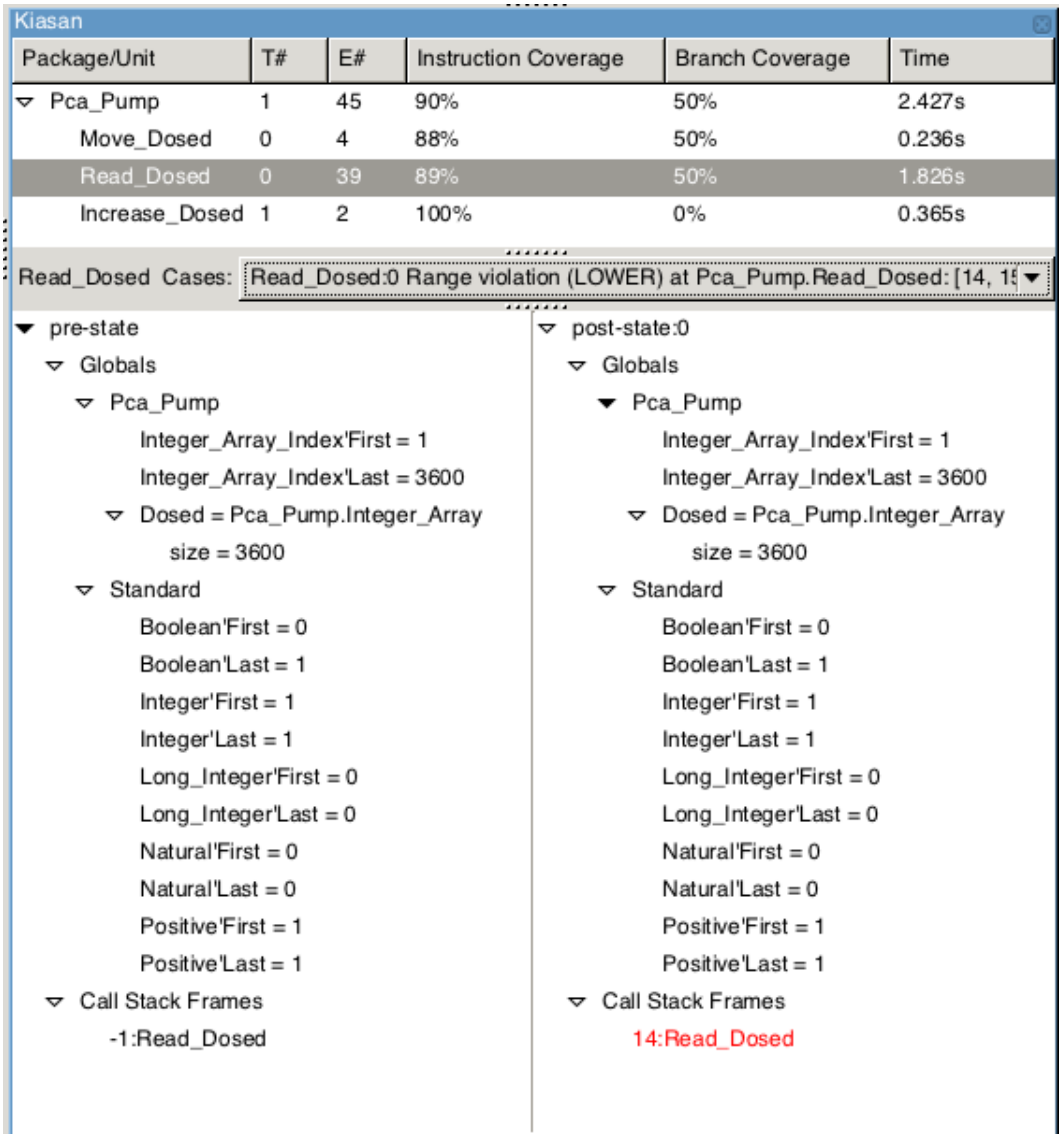

Figure 6.5: Bakar Kiasan verification report

The first issue we can notice is problem with data types' ranges indicated by Exception cases, e.g., `Read_Dosed:0 Range violation (LOWER) at Pca_Pump.Read_Dosed:[14,15]` (presented in Figure 6.5). To solve it (in SPARK 2005) configuration file `Standard.ads` (presented in Figure 6.6), which specifies `Integer` type range, was created. This is information for verification tools, which may helps in verification. The Kiasan verification report generated after that is presented in Figure 6.7. The number of errors is reduced, but now there is possible overflow

```
package Standard is

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

end Standard;
```

Figure 6.6: Configuration file for Bakar Kiasan

violation indicated, e.g., by Exception case 0 for `Increase_Dosed` procedure: `Arithmetic overflow violation (LOWER) at Pca_Pump.Increase_Dosed: [9,90]` (presented in Figure 6.7).

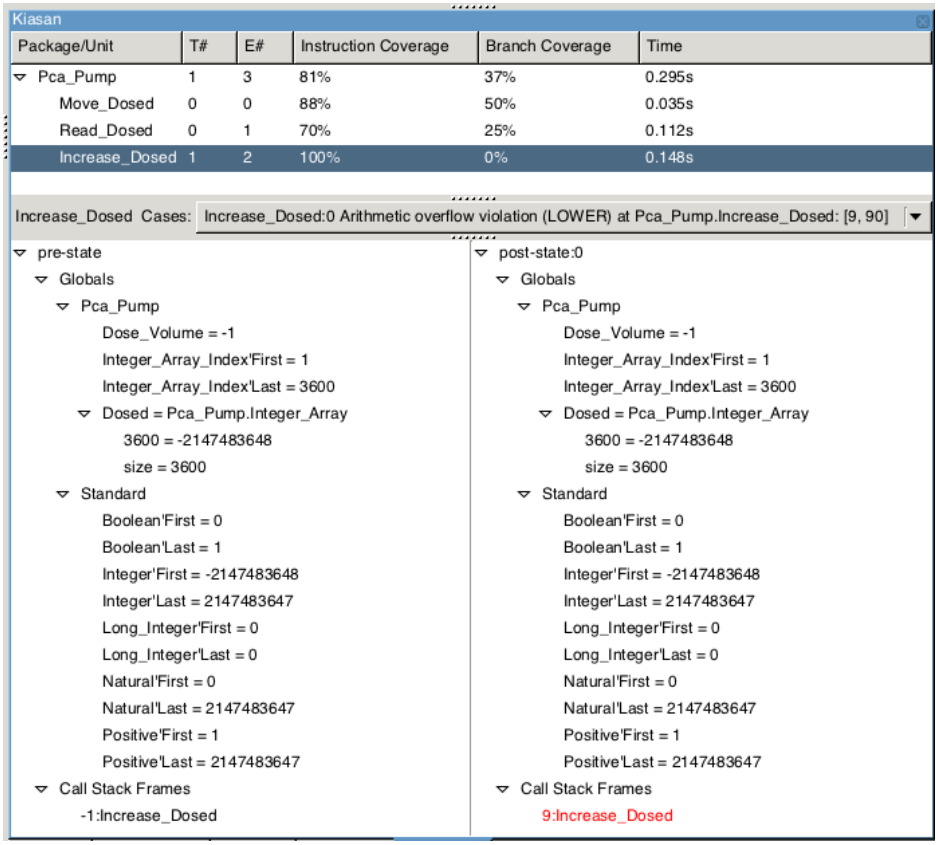

Figure 6.7: Bakar Kiasan verification report, second run

From functional perspective, negative values are not needed it this case, thus new type `Drug_Volume` type was created. `Integer_Array` type was renamed to `Doses_Array` and its type was changed to `Drug_Volume`. Result of Kiasan analysis after this change is presented in Figure 6.8.

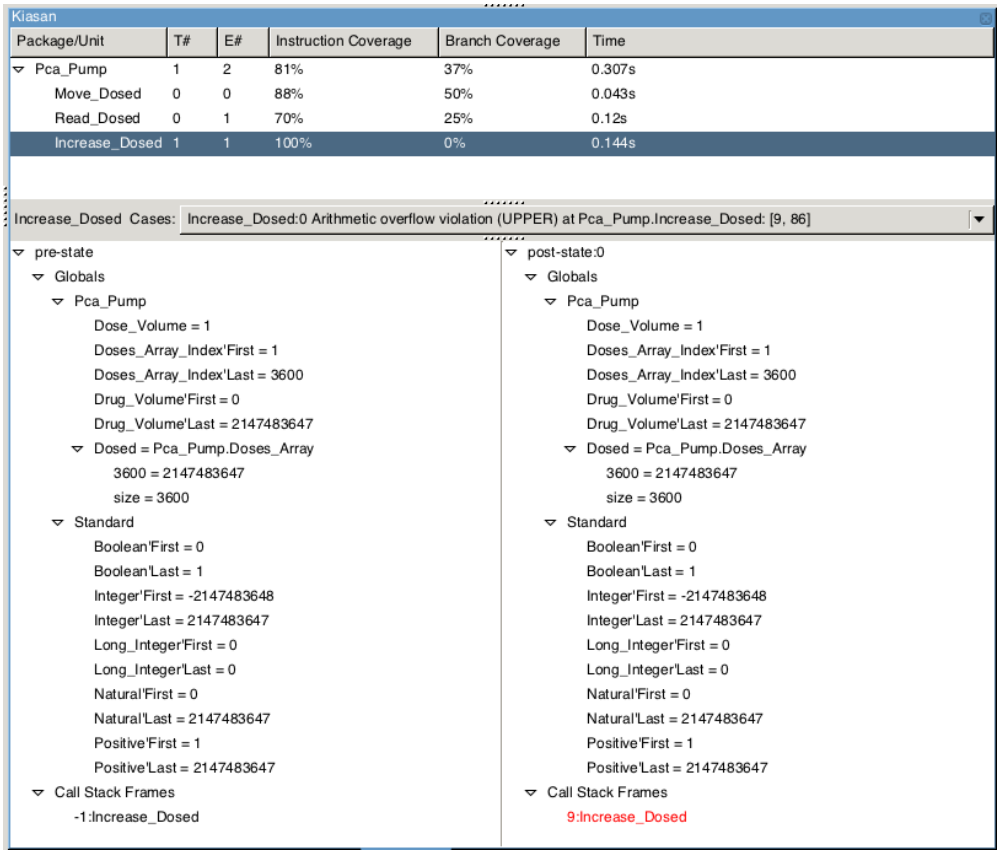

Figure 6.8: Bakar Kiasan verification report, third run

This change eliminated lower overflow, because now negative value cannot be added to any array element. Only upper overflow in `Increase_Dosed` procedure error was left. The fix for this is the introduction of precondition for `Increase_Dosed`: `--# pre Read_Dosed(Dosed) <= Drug_Volume'Last - Dose_Volume;`. Addition of this contract caused semantic error (detected by Examiner): `The identifier Read_Dosed is either undeclared or not visible at this point.` This error is caused by the definition of `Increase_Dosed` procedure before `Read_Dosed` procedure. To fix, this `Read_Dosed` procedure was moved before `Increase_Dosed`. However, after that Examiner returned different error: `Binary operator is not declared for types Drug_Volume and Dose_Volume__type.` To make the operator visible, `Dose_Volume` type has to be declared in `--# own` annotation: `--# Dose_Volume : Drug_Volume;`. After these fixes, Kiasan analysis has be run again. The result is depicted in the Figure 6.9.

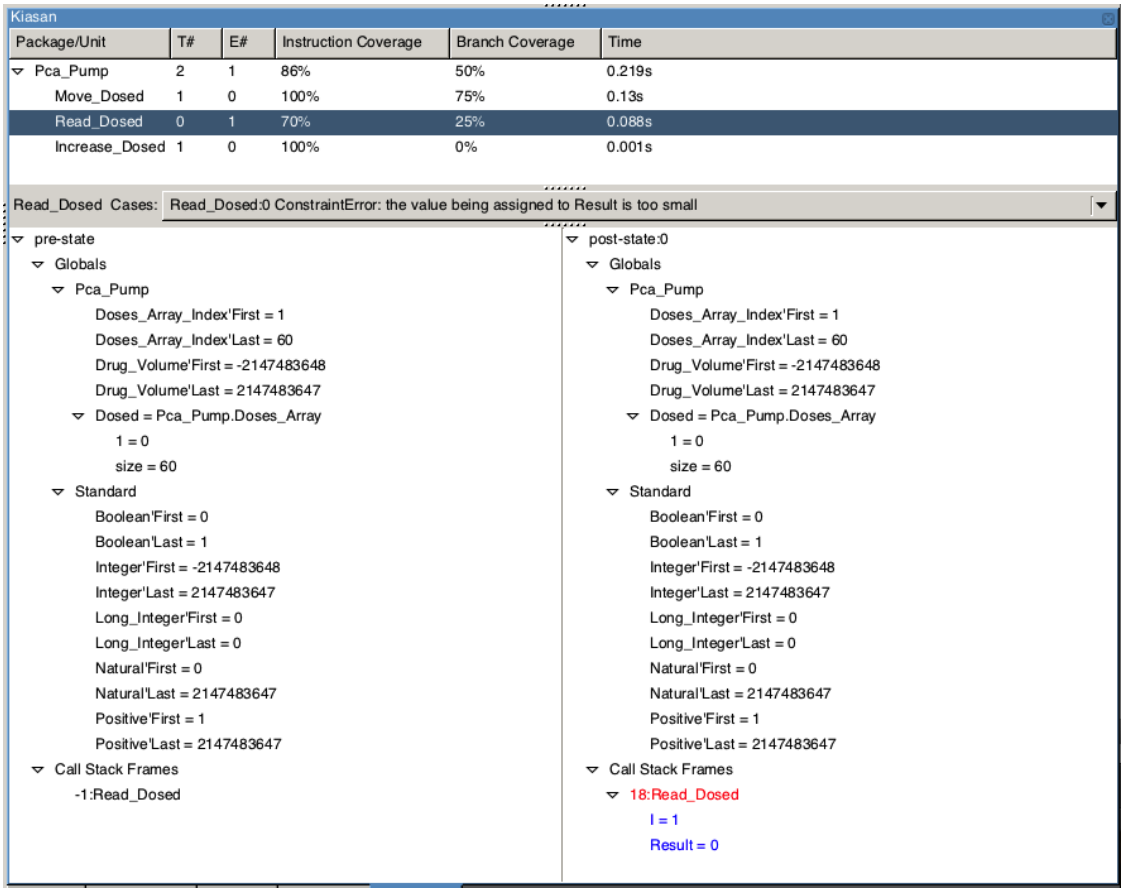

Figure 6.9: Bakar Kiasan verification report, fourth run

There were no error cases in the `Move_Dosed` and the `Increase_Dosed` procedures. The error case in `Read_Dosed` is shown in Figure 6.9. It is `ConstraintError: the value being assigned to Result is too small` . This error is not very informative. After investigation and talks with the Kiasan Developer, it was determined that there is a bug in Kiasan v1 (for SPARK 2005). More precisely: in handling overflows. For the purpose of verification, `Drug_Volume` type range was changed to $0-(2^{15}-1)$. This 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 prototype implementation). 1000000 micro liters is 1000 ml, which is 1 liter. This is an extreme amount of drug in case of PCA pump, according to Requirements Document [LH14]. The bug with type ranges is fixed in Kiasan v2 (for SPARK 2014).

Another problem was the size of `Dosed` array (3600 elements). Kiasan allows the developer to configure the array bound and loop bound. Both had to be increased (from default 10). Another thing was computational complexity. For 3600 elements, state space grows exponentially and it takes a lot of time to analyze it. Thus, for verification purposes, array size was changed to 60 elements along with change to array bounds and loop bounds, also to 60.

After rerunning Kiasan, there is valid test case for `Read_Dose`, but there are also 59 Exception cases: **Range** `violation (UPPER)`, which means possible overflow. One way of fix this problem, was to add an `--# assume` annotation to loop in function body, stating that every sum operation in the loop will not cause overflow, but Kiasan v1 does not support `assume` annotations. Another way was to add precondition that ensures, that the sum of elements is lower than `Drug_Volume'Last`. SPARK does not provide simple library for summing an array (like the Contracts language for Java provides). Thus, this function had to be implemented. Its implementation is the same as `Read_Dosed`. It sums all elements of array. The `Sum` function specification and body is presented in the Figure 6.10. After rerunning Kiasan, only valid test cases were found, which is depicted in the Figure 6.11.

```
-- pca_pump.ads
function Sum(Arr : Doses_Array) return Drug_Volume;


-- pca_pump.adb
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;
```

Figure 6.10: Sum function for summing all elements of array

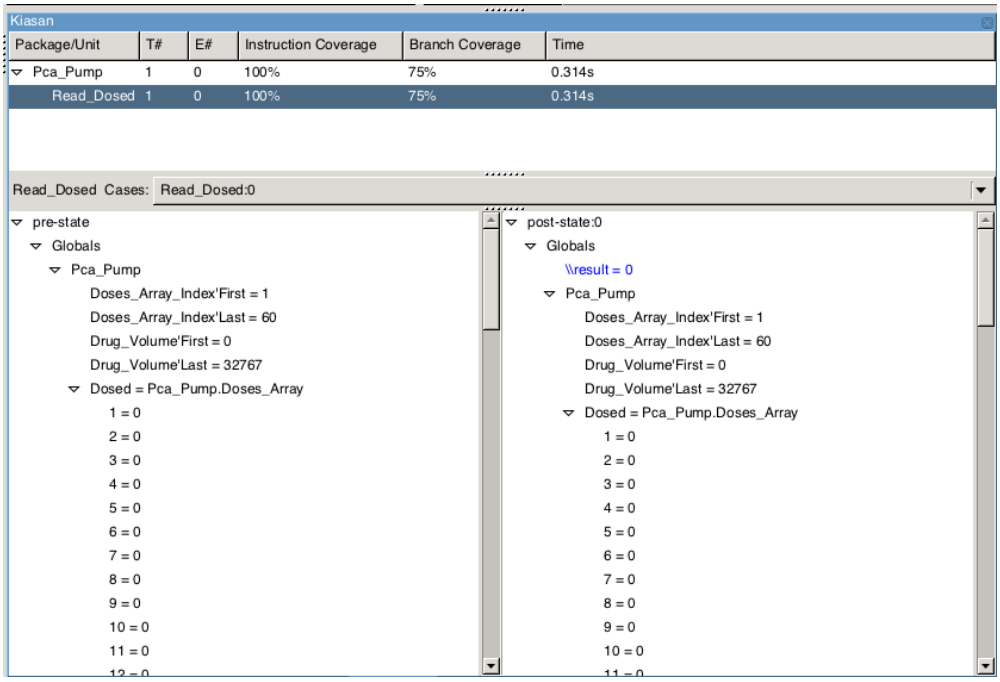

Figure 6.11: Bakar Kiasan verification report, fifth run

The last thing which was improved by code contracts is checking if `Move_Dosed` procedure works as expected. In that purpose three postconditions were added (Figure 6.12). 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));
```

Figure 6.12: 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.

Another way to validate such requirements is to create AUnit tests. In Section 6.4, there is an overview of unit tests created to test behavior described above. Furthermore, symbolic execution technique (used by Kiasan) allows to generate AUnit tests automatically, and this feature is under development in Kiasan v2.

To validate changes made, while working with Kiasan, SPARK Examiner and SPARK-Simp were rerun again. POGS report is presented in the Figure 6.13.

There are 4 undischarged VCs, but total number of generated VCs is 19. In previous run there were only 15. Thus, there are 4 new VCs, and 2 of them are undischarged. The reason is introduction of `Sum` function used by all subprograms. This can be confirmed by examining all undischarged VCs: 1st VC in `increase_dosed.siv` file (Figure 6.14), 9th VC in `move_dosed.siv` file (Figure 6.15), 3rd VC in `read_dosed.vcg` file (Figure 6.16) and 3rd VC in `sum.vcg` file (Figure 6.17). They derived form the subprograms: `Increase_Dosed`, `Move_Dosed`, `Read_Dosed` and `Sum` respectively.

In `Move_Dosed` procedure, the SPARK tools cannot prove the implications in post conditions. Fortunately, it is already proved by Bakar Kiasan. The problem in `Increase_Dosed`, `Read_Dosed` and `Sum` is the same. The SPARK 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. Four exception cases indicating possible overflow are generated. Thus, the only way to discharge the verification obligation of this module is to assume that the proof function `Sum` is correct.

In procedure `Move_Dosed`, there is one dead path found. POGS report gives only information where dead path exists, but not in which circumstances. The information about conditions, in which dead path occurs is stored in `.dpc` file. The file path to concrete file is given in the POGS report just before summary table for procedure `Move_Dosed`. In this case it is `move_dosed.dpc`

```
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   |
 ------------------------------------------------------------------------------

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

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%
```

Figure 6.13: Third POGS report

```
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 .
```

Figure 6.14: 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)) .
```

Figure 6.15: Undischarged Verification Condition from `move_dosed.siv` file

file. Figure 6.13 presents truncated POGS report, but as an example, full POGS report of implemented PCA prototype can be found in Appendix C (e.g. see line 50, which contains DPC analysis for `Start_Pumping` procedure).

The relevant fragment, which applies to the found dead path is presented in Figure 6.18. It is a list of hypothesis, in which hypothesis 10 (H10) states that number of elements in `Doses_Array` is 1 or less. In this case (or more precisely: in this path), **for loop** will not be visited. `Doses_Array` has always 60 elements, thus this path is impossible (dead). It does not mean something bad, because dead path indicate possible issues. In this case it is not issue.

```
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 .
```

Figure 6.16: 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 .
```

Figure 6.17: Undischarged Verification Condition from `sum.siv` file

It is expected behavior. The complete code of the module for dose monitoring, after changes described above, is presented in Figures 6.19 and 6.20.

Code contracts (pre- and postconditions), added during this example verification process, cannot be applied to PCA Pump Prototype implementation, which use RavenSPARK, because they contains protected objects, and - as mentioned in chapter 2.6 - protected objects cannot be used in proof annotations (pre- and postconditions). However, code fixes made in this section can be applied. This shows how code implemented based on translation from AADL/BLESS can be processed by SPARK tools to ensure absence of runtime exceptions.

```
procedure_move_dosed_8.
H1:    for_all(i___1: integer, ((i___1 >= doses_array_index__first) and (
           i___1 <= doses_array_index__last)) -> ((element(
           dosed, [i___1]) >= drug_volume__first) and (element(
           dosed, [i___1]) <= drug_volume__last))) .
H2:    doses_array_index__last - 1 >= integer__first .
H3:    doses_array_index__last - 1 <= integer__last .
H4:    doses_array_index__last - 1 >= integer__base__first .
H5:    doses_array_index__last - 1 <= integer__base__last .
H6:    doses_array_index__first >= integer__first .
H7:    doses_array_index__first <= integer__last .
H8:    (doses_array_index__first <= doses_array_index__last - 1) -> ((
           doses_array_index__last - 1 >= doses_array_index__first) and (
           doses_array_index__last - 1 <= doses_array_index__last)) .
H9:    (doses_array_index__first <= doses_array_index__last - 1) -> ((
           doses_array_index__first >= doses_array_index__first) and (
           doses_array_index__first <= doses_array_index__last)) .
H10:   not (doses_array_index__first <= doses_array_index__last - 1) .
H11:   0 >= drug_volume__first .
H12:   0 <= drug_volume__last .
H13:   doses_array_index__last >= doses_array_index__first .
H14:   doses_array_index__last <= doses_array_index__last .
        ->
C1:    false .
```

Figure 6.18: Dead path in `Move_Dosed` procedure

6.3 Verification of Generated Code

This section presents how SPARK 2005 tools can help with verification and further implementation of automatically generated code from AADL models.

Code translated from simplified PCA Pump AADL models is presented in Appendix F. Verification with Examiner of package `Pca_Operation` specification returns syntax error: `Neither KNOWN_DISCRIMINANT_PART nor TASK_TYPE_ANNOTATION can start with reserved word "IS"`. This means that discriminants or task annotation are expected here. In order to pass Examiner syntax check, at least one annotation has to be declared. For demonstration purposes, `Ada.Real_Time.ClockTime` is used, which announce usage of `ClockTime` variable from `Ada.Real_Time` library. The complete task declaration is presented in the Figure 6.21.

Once annotation is added, `Pca_Operation` package specification passes Examiner syntax check. Verification of package body returns errors, which are caused by non-implemented assertions (translated from BLESS). When all such incomplete assertions are removed, only

```
package Pca_Pump
--# own Dosed : Doses_Array;
--#     Dose_Volume : Drug_Volume;
--# initializes Dosed,
--#             Dose_Volume;
is
    type Drug_Volume is range 0 .. 2**15-1;

    subtype Doses_Array_Index is Positive range 1 .. 60;
    type Doses_Array is array (Doses_Array_Index) of Drug_Volume;

    function Sum(Arr : Doses_Array) return Drug_Volume;

    function Read_Dosed return Drug_Volume;
    --# global in Dosed;
    --# pre Sum(Dosed) <= Drug_Volume'Last;

    procedure Increase_Dosed;
    --# global in out Dosed;
    --#        in Dose_Volume;
    --# derives Dosed from Dosed, Dose_Volume;
    --# pre Read_Dosed(Dosed) <= Drug_Volume'Last - Dose_Volume;

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

end Pca_Pump;
```

Figure 6.19: Dose monitoring module after changes: package specification

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

    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;

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

    function Read_Dosed return Drug_Volume
    is
        Result : Drug_Volume := 0;
    begin
        for I in Doses_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 Doses_Array_Index range Doses_Array_Index'First .. Doses_Array_Index'Last-1 loop
            --# assert I > 1 -> Dosed(I-1) = Dosed(I);
            Dosed(I) := Dosed(I+1);
        end loop;
        Dosed(Doses_Array_Index'Last) := 0;
    end Move_Dosed;

end Pca_Pump;
```

Figure 6.20: Dose monitoring module after changes: package body

```
task type Patient_Bolus_Checker
--# global in Ada.Real_Time.ClockTime;
--# derives null from Ada.Real_Time.ClockTime;
is
    pragma Priority(10);
end Patient_Bolus_Checker;
```

Figure 6.21: Undischarged Verification Condition from `sum.siv` file

```
pca_operation.adb:53:9: Flow Error  30 - The variable Bolus_Duration_In is imported but neither referenced nor
      exported.
pca_operation.adb:72:9: Flow Error  30 - The variable Rx_In is imported but neither referenced nor exported.
pca_operation.adb:82:9: Flow Error  30 - The variable Infusion_Flow_Rate is imported but neither referenced
     nor exported.
pca_operation.adb:92:9: Flow Error  32 - The variable Infusion_Flow_Rate is neither imported nor defined.
pca_operation.adb:92:9: Flow Error  31 - The variable Infusion_Flow_Rate is exported but not (internally)
     defined.
pca_operation.adb:92:9: Flow Error  32 - The variable System_Status is neither imported nor defined.
pca_operation.adb:92:9: Flow Error  31 - The variable System_Status is exported but not (internally) defined.
pca_operation.adb:92:9: Warning 400 - Variable la is declared but not used.
pca_operation.adb:101:9: Flow Error  35 - Importation of the initial value of variable Ada.Real_Time.ClockTime
      is ineffective.
```

Figure 6.22: Flow errors returned by Examiner for `Pca_Operation` package body

flow errors (presented in the Figure 6.22) are found by Examiner.

This is a nice indication of what has to be implemented in particular parts of the program. It is recommended to not use VC and DPC generation until there are some syntax errors. When all errors are fixed, program can be initially verified as described in previous sections.

6.4 AUnit Tests

To prove expected behavior of `Move_Dosed` in Dose monitoring module, presented in Section 6.2, instead of test cases generation, AUnit tests can be created manually. Verification tools can confirm that created unit tests are valid cases or not. 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

```
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;
```

Figure 6.23: AUnit tests for `Move_Dosed` procedure

- `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 in the Figure 6.23. All AUnit tests can be found in Appendix G.

```
package Pca_Pump
with SPARK_Mode,
  Abstract_State => (Dosed_State, Dose_Volume_State),
  Initializes    => (Dosed_State, Dose_Volume_State)
is
   type Drug_Volume is range 0 .. 2**15-1;

   subtype Doses_Array_Index is Integer range 1 .. 60;
   type Doses_Array is array (Doses_Array_Index) of Drug_Volume;

   function Dosed_State return Doses_Array
     with Convention => Ghost,
     Global => (Input => Dosed_State);

   function Dose_Volume_State return Drug_Volume
     with Convention => Ghost,
     Global => (Input => Dose_Volume_State);

   function Sum(Arr : Doses_Array) return Drug_Volume
     with Convention => Ghost;

   function Read_Dosed return Drug_Volume
     with Global  => (Input => (Dosed_State)),
     Pre     => Sum(Dosed_State) <= Drug_Volume'Last;

   procedure Increase_Dosed
     with Global  => (Input  => Dose_Volume_State, In_Out => Dosed_State),
     Depends => (Dosed_State => (Dosed_State, Dose_Volume_State)),
     Pre     => Read_Dosed <= Drug_Volume'Last - Dose_Volume_State;

   pragma Unevaluated_Use_Of_Old (Allow);

   procedure Move_Dosed
     with Global  => (In_Out => Dosed_State),
     Depends => (Dosed_State => Dosed_State),
     Post => (Dosed_State (Doses_Array_Index'Last) = 0),
     Contract_Cases => (Dosed_State(Doses_Array_Index'First) = 0 => Read_Dosed'Old = Read_Dosed,
                        Dosed_State(Doses_Array_Index'First) > 0 => Read_Dosed'Old > Read_Dosed);

end Pca_Pump;
```

Figure 6.24: Sequential module for dose monitoring in SPARK 2014: package specification

6.5 GNATprove

The sequential module for monitoring dosed amount verification presented in Section 6.2 has been converted to SPARK 2014. For conversion, "SPARK 2005 to 2014" translator (created by AdaCore) has been used. Translated code is presented in Figures 6.24 and 6.25.

In SPARK 2014, the `Standard.ads` file with type ranges is not necessary, because it is handled by language. SPARK 2014 introduces notion of ghost functions. They are used to declare functions that are needed only in annotations. Proof function `Sum` is defined as

```
package body Pca_Pump
with SPARK_Mode, Refined_State => (Dosed_State => Dosed, Dose_Volume_State => Dose_Volume)
is
   Dosed : Doses_Array := Doses_Array'(others => 0);
   Dose_Volume : Drug_Volume := 1;

   function Dosed_State return Doses_Array
     with Refined_Global => (Input => Dosed)
   is begin
      return Dosed;
   end Dosed_State;
   function Dose_Volume_State return Drug_Volume
     with Refined_Global => (Input => Dose_Volume)
   is begin
      return Dose_Volume;
   end Dose_Volume_State;

   function Sum(Arr : Doses_Array) return Drug_Volume
   is
      Result : Drug_Volume := 0;
   begin
      for I in Doses_Array_Index loop
         pragma Loop_Invariant (true);
         Result := Result + Arr(I);
      end loop;
      return Result;
   end Sum;

   procedure Increase_Dosed
     with Refined_Global => (Input => Dose_Volume, In_Out => Dosed),
     Refined_Depends => (Dosed => (Dosed, Dose_Volume))
   is begin
      Dosed(Doses_Array_Index'Last) := Dosed(Doses_Array_Index'Last) + Dose_Volume;
   end Increase_Dosed;

   function Read_Dosed return Drug_Volume
     with Refined_Global => (Input => (Dosed))
   is
      Result : Drug_Volume := 0;
   begin
      for I in Doses_Array_Index loop
         pragma Loop_Invariant (if I > 1 then Result >= Dosed (I-1));
         Result := Result + Dosed(I);
      end loop;
      return Result;
   end Read_Dosed;

   procedure Move_Dosed
     with Refined_Global => (In_Out => (Dosed)),
     Refined_Depends => (Dosed => Dosed)
   is begin
      for I in Doses_Array_Index range 1 .. Doses_Array_Index'Last-1 loop
         pragma Loop_Invariant (if I > 1 then Dosed (I-1) = Dosed (I));
         Dosed(I) := Dosed(I+1);
      end loop;
      Dosed(Doses_Array_Index'Last) := 0;
   end Move_Dosed;
end Pca_Pump;
```

Figure 6.25: Sequential module for dose monitoring in SPARK 2014: package body

ghost function. In order to use private, global variables in package specification, abstract refinement and ghost functions (`Dosed_State` and `Dose_Volume_State`) have been used. The **Pragma** `Unevaluated_Use_Of_Old` is used to avoid the error: `prefix of attribute "Old" that is potentially unevaluated must denote an entity.`

Above code has been verified with GNATprove tool. Data and information flow analysis did not return any warnings nor errors. Proof analysis was performed with the following parameters:

- Proof strategy: One proof per path
- Prover timeout: 60
- Do not treat warnings as errors: `--warnings=continue` flag
- Report checks proved

All above parameters gives following command: `gnatprove -P\%PP --ide-progress-bar -U --proof=per_path --timeout=60 --warnings=continue --report=all` (where `\%PP` is path of verified project file `.gpr`).

Proof analysis can be run from GPS (menu: SPARK 2014 / Prove All). There is GUI for options customization (see Figure 6.26).

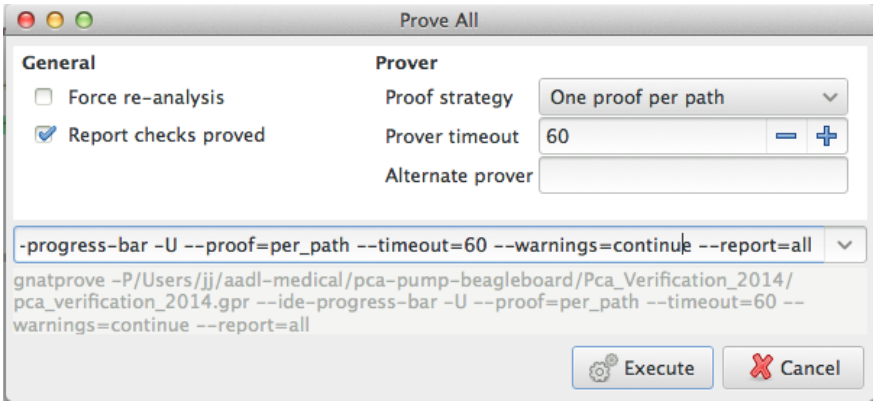

Figure 6.26: GNATprove settings

```
analyzing Pca_Pump, 1 checks
analyzing Pca_Pump.Dosed_State, 0 checks
analyzing Pca_Pump.Dose_Volume_State, 0 checks
analyzing Pca_Pump.Sum, 3 checks
analyzing Pca_Pump.Read_Dosed, 4 checks
analyzing Pca_Pump.Increase_Dosed, 3 checks
analyzing Pca_Pump.Move_Dosed, 12 checks
pca_pump.adb:5:39: info: length check proved
pca_pump.adb:27:10: info: loop invariant initialization proved
pca_pump.adb:27:10: info: loop invariant preservation proved
pca_pump.adb:28:27: warning: overflow check might fail
pca_pump.adb:38:70: warning: overflow check might fail
pca_pump.adb:47:10: info: loop invariant initialization proved
pca_pump.adb:47:10: info: loop invariant preservation proved
pca_pump.adb:47:65: info: index check proved
pca_pump.adb:48:27: warning: overflow check might fail
pca_pump.adb:59:10: info: loop invariant initialization proved
pca_pump.adb:59:10: info: loop invariant preservation proved
pca_pump.adb:59:55: info: index check proved
pca_pump.ads:29:17: info: precondition proved
pca_pump.ads:29:48: info: overflow check proved
pca_pump.ads:36:14: warning: postcondition might fail, requires Dosed_State (Doses_Array_Index'Last) = 0
pca_pump.ads:37:6: info: disjoint contract cases proved
pca_pump.ads:37:6: info: complete contract cases proved
pca_pump.ads:37:66: warning: contract case might fail
pca_pump.ads:37:69: info: precondition proved
pca_pump.ads:37:86: info: precondition proved
pca_pump.ads:38:66: warning: contract case might fail
pca_pump.ads:38:69: info: precondition proved
pca_pump.ads:38:86: info: precondition proved
```

Figure 6.27: GNATprove verification summary of module for dose monitoring in SPARK 2014

Summary of proof analysis is presented in the Figure 6.27. Proof analysis returned three warnings: `overflow check might fail` and one warning: `contract case might fail`. It indicates the same problem like in verification with SPARK 2005 tools: potential for overflow. Additionally, there is a warning (`postcondition might fail`) caused by tools limitations, which are not able to infer dependency between ghost function `Dosed_State` and array `Dosed`. If state refinement is not used (i.e. refined variables are defined in package specification), and actual array is used in the postcondition (instead of ghost function), this warning does not occur. The same program without abstract state is presented in the figures 6.28 and 6.29. Its verification summary is shown in the Figure 6.30.

```
package Pca_Pump_No_Refinement
  with SPARK_Mode
is
   type Drug_Volume is range 0 .. 2**15-1;

   subtype Doses_Array_Index is Integer range 1 .. 60;
   type Doses_Array is array (Doses_Array_Index) of Drug_Volume;

   Dosed : Doses_Array := Doses_Array'(others => 0);
   Dose_Volume : Drug_Volume := 1;

   function Sum(Arr : Doses_Array) return Drug_Volume
     with Convention => Ghost;

   function Read_Dosed return Drug_Volume
     with Global  => (Input => (Dosed)),
     Pre     => Sum(Dosed) <= Drug_Volume'Last;

   procedure Increase_Dosed
     with Global  => (Input  => Dose_Volume, In_Out => Dosed),
     Depends => (Dosed => (Dosed, Dose_Volume)),
     Pre     => Read_Dosed <= Drug_Volume'Last - Dose_Volume;

   pragma Unevaluated_Use_Of_Old (Allow);

   procedure Move_Dosed
     with Global  => (In_Out => Dosed),
     Depends => (Dosed => Dosed),
     Post => (Dosed(Doses_Array_Index'Last) = 0),
     Contract_Cases => (Dosed(Doses_Array_Index'First) = 0 => Read_Dosed'Old = Read_Dosed,
                        Dosed(Doses_Array_Index'First) > 0 => Read_Dosed'Old > Read_Dosed);
end Pca_Pump_No_Refinement;
```

Figure 6.28: Sequential module for dose monitoring in SPARK 2014 without variable refinement: package specification

```
package body Pca_Pump_No_Refinement
  with SPARK_Mode
is
   function Sum(Arr : Doses_Array) return Drug_Volume
   is
      Result : Drug_Volume := 0;
   begin
      for I in Doses_Array_Index loop
         pragma Loop_Invariant (true);
         Result := Result + Arr(I);
      end loop;
      return Result;
   end Sum;

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

   function Read_Dosed return Drug_Volume
   is
      Result : Drug_Volume := 0;
   begin
      for I in Doses_Array_Index loop
         pragma Loop_Invariant (if I > 1 then Result >= Dosed (I-1));
         Result := Result + Dosed(I);
      end loop;
      return Result;
   end Read_Dosed;

   procedure Move_Dosed
   is
   begin
      for I in Doses_Array_Index range 1 .. Doses_Array_Index'Last-1 loop
         pragma Loop_Invariant (if I > 1 then Dosed (I-1) = Dosed (I));
         Dosed(I) := Dosed(I+1);
      end loop;
      Dosed(Doses_Array_Index'Last) := 0;
   end Move_Dosed;
end Pca_Pump_No_Refinement;
```

Figure 6.29: Sequential module for dose monitoring in SPARK 2014 without variable refinement: package body

```
analyzing Pca_Pump_No_Refinement, 1 checks
analyzing Pca_Pump_No_Refinement.Sum, 3 checks
analyzing Pca_Pump_No_Refinement.Read_Dosed, 4 checks
analyzing Pca_Pump_No_Refinement.Increase_Dosed, 3 checks
analyzing Pca_Pump_No_Refinement.Move_Dosed, 12 checks
pca_pump_no_refinement.adb:9:10: info: loop invariant initialization proved
pca_pump_no_refinement.adb:9:10: info: loop invariant preservation proved
pca_pump_no_refinement.adb:10:27: warning: overflow check might fail
pca_pump_no_refinement.adb:18:70: warning: overflow check might fail
pca_pump_no_refinement.adb:26:10: info: loop invariant initialization proved
pca_pump_no_refinement.adb:26:10: info: loop invariant preservation proved
pca_pump_no_refinement.adb:26:65: info: index check proved
pca_pump_no_refinement.adb:27:27: warning: overflow check might fail
pca_pump_no_refinement.adb:36:10: info: loop invariant initialization proved
pca_pump_no_refinement.adb:36:10: info: loop invariant preservation proved
pca_pump_no_refinement.adb:36:55: info: index check proved
pca_pump_no_refinement.ads:9:39: info: length check proved
pca_pump_no_refinement.ads:22:17: info: precondition proved
pca_pump_no_refinement.ads:22:48: info: overflow check proved
pca_pump_no_refinement.ads:29:14: info: postcondition proved
pca_pump_no_refinement.ads:30:6: info: disjoint contract cases proved
pca_pump_no_refinement.ads:30:6: info: complete contract cases proved
pca_pump_no_refinement.ads:30:60: warning: contract case might fail
pca_pump_no_refinement.ads:30:63: info: precondition proved
pca_pump_no_refinement.ads:30:80: info: precondition proved
pca_pump_no_refinement.ads:31:60: warning: contract case might fail
pca_pump_no_refinement.ads:31:63: info: precondition proved
pca_pump_no_refinement.ads:31:80: info: precondition proved
```

Figure 6.30: GNATprove verification summary of module for dose monitoring in SPARK 2014 without variable refinement

6.6 Assessment

The verification approach presented in this chapter allowed to detect potential run-time exceptions (e.g., overflow). In the future, this approach can be used also for the verification of requirements specified by BLESS annexes in AADL models. The SPARK Examiner was helpful not only for the verification, but also during the implementation, in flow errors detection, which indicates when package implementation does not conform to its specification. In the demonstrated example (of the translated PCA Pump from AADL models) this means just lack of the implementation, and was a suggestion for the developer regarding parts of the system that are not complete.

Bakar Kiasan was used extensively in resolving possible run-time errors. Test cases generated by this tool gave very intuitive overview of faced problems. As a complementary to test cases generation, the AUnit tests were created manually to cross-check the obtained results.

The presented verification approach might be also helpful in verifying systems that use the run time assertions. The verification can detect assertions that can potentially fail. This should lead to tweaking the code, to avoid undesired behavior or handling assertions fails.

7

Summary

"Success is determined not by whether or not you face obstacles, but by your reaction to them. And if you look at these obstacles as a containing fence, they become your excuse for failure. If you look at them as a hurdle, each one strengthens you for the next."

– Ben Carson

In this thesis PCA Pump prototype, in SPARK 2005 with Ravenscar profile, has been created. It runs on BeagleBoard-xM platform and control physical device. Furthermore, AADL/BLESS to SPARK Ada translation is proposed. Based on that sample translation from simplified AADL models of PCA pump has been performed. At the end, example verification (targeting absence of runtime exceptions) of created PCA Pump Prototype, isolated module for dose monitoring and translated code has been shown.

All work done in this thesis targets SPARK 2005. SPARK 2014 and its tools (such as GNATprove) were not completed at the time, when this thesis was written. However, an example verification (of dose monitoring module, which has been translated to SPARK 2014) was presented.

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 is required to isolate some functionalities or implement them in different way. Another issue was lack of resources and SPARK code samples - especially realistic medical devices code examples, which are keep secretly as intellectual property by companies. Available resources are usually small examples used in research or reference manuals, which were created a number of years ago. Although still valid, these have not been updated or expanded for years.

Furthermore, BLESS and SPARK are still under development. Thus, it was very hard to take advantage of all desirable capabilities (most of features are not yet implemented). An example may be lack of support for pre- and post conditions in RavenSPARK.

In addition to that, community working with above technologies is very small. On StackOverflow[1] there is 728 question related to Ada[2] and only 3 to SPARK.[3] In the same time, C# has 673,721 questions[4] and Java - 682,308.[5]

Proposed mapping from AADL to SPARK Ada is not consulted with industry engineers. Thus, it would be first thing to do, in order to continue this research. A lot of work can be done in this topic, as is described in Chapter 8.

[1] http://stackoverflow.com
[2] http://stackoverflow.com/questions/tagged/ada
[3] http://stackoverflow.com/questions/tagged/spark-ada
[4] http://stackoverflow.com/questions/tagged/c%23
[5] http://stackoverflow.com/questions/tagged/java

8

Future Work

"If you fail to plan, you plan to fail."

– Benjamin Franklin

The following are possible extensions for work done in this thesis:

- The most important thing, which would be extremely helpful to proceed with work done in this thesis, would be to review it by some industry expert and experienced engineer. Especially, how particular functionalities (like monitoring external sensors or controlling pump actuator) are implemented, and how looks communication between components modeled in AADL.

- Creation of automatic translator described in Section 4.3 would be good validation of created translation schemes. It may reveal some issues not present for manual translation.

- Currently AADL thread properties are not taken into account in thread to task mapping, in Section 4.1.3. Properties like priority or period could be mapped pretty straightforward to SPARK Ada. For now, former is hard-coded as 10 and latter simply skipped, which requires developer to handle it. However, given properties that are

modeled and analyzed in AADL models, should be translated automatically to maintain synchronization between models and the code. AADL properties are described in [FG13], in the Appendix A.

- Data types translation presented in Section 4.1.1, in addition to straightforward type mapping, includes protected types. However, all protected types have the same set of subprograms (`Put` and `Get`). It is worth to consider introduction of generics, which will allow to specify generic protected type and then reuse it for all types.

- In feature groups translation (Section 4.1.5), idea of child or nested packages was abandoned. However, it would be good to reconsider it for the purposeof providing encapsulation for grouped features. It may be useful to introduce getter functions in parent package or some other technique that would allow for better separation and decomposition.

- AADL property set mapping in Section 4.1.7 handles only `aadlinteger` type. Thus, it requires extension for handling other more complex constructs.

- Current translation schemes cause creation of pretty big packages, which will become bigger after adding implementation. Thus, some decomposition is desired. The following techniques can be considered:

  - partitioning of packages
  - taking advantage of child packages
  - separation of threads to different packages (e.g. one thread per child package and all common functionalities in parent package)
  - simple package separation

- The mappings for BLESS are limited only to a small subset. Development of translations for BLESS state machines (states and transitions) would be good addition. this

would support behavior translation. A good point to start is the `Rate_Controller` thread, which can be found in `PCA_Operation_Threads` package in original AADL models created by Brian Larson. The semantics of BLESS contain notions of time that make translation to SPARK difficult. This problem occurs in state machine models. Finding solutions for that is needed. Maybe even, by changing BLESS semantics.

- When this thesis was written, SPARK 2014 did not support multitasking. However, there were plans to introduce it into SPARK 2014 following an approach similar to SPARK 2005. Once multitasking support is present, translations for SPARK 2014 will be possible.

- There is an issue with two way communication between SPARK packages caused by circular dependency. It is described in Section 4.2.1.

- The port communication presented in Section 4.2 captures only 1:1 connections between ports of the same type and opposite direction. In AADL there are also inter-port connections and one-to-many or many-to-one connections. [FG13] They should be taken into AADL subset for medical devices modeling and translation.

- The created PCA pump prototype contains only basic functionalities. Some parameters (like drug concentration) are ignored. The next step is its development should be to include funcionality that has been omitted. In addition to that, interaction with external modules such as sensors for monitoring drug flow or communication with ICE through Ethernet port is desired. This requires creation of communication channel between BeagleBoard (SPARK Ada application) and these systems.

Bibliography

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

[AL14a] AdaCore and Altran UK Ltd. Spark 2014 reference manual. URL: http://docs.adacore.com/spark2014-docs/html/lrm, 2011-2014.

[AL14b] AdaCore and Altran UK Ltd. Spark 2014 toolset user's guide. URL: http://docs.adacore.com/spark2014-docs/html/ug, 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.

[BDV04] Alan Burns, Brian Dobbing, and Tullio Vardanega. Guide for the use of the ada ravenscar profile in high integrity systems. *ACM SIGAda Ada Letters*, 24(2):1–74, Juin 2004.

[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.

[CBGP12] Fabien Cadoret, Etienne Borde, Sébastien Gardoll, and Laurent Pautet. Design patterns for rule-based refinement of safety critical embedded systems models. In *International Conference on Engineering of Complex Computer Systems (ICECCS'12)*, pages 67 – 76, Paris (France), 2012. IEEE.

[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.

[FWH] P. Feiler, L. Wrage, and J. Hansson. System architecture virtual integration: A case study. In *Embedded Real-time Software and Systems Conference*.

[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 *Proceedings of the 2012 International Conference on Cyber-Physical Systems*, pages 3 – 12. IEEE, 2012.

[HLL+] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. In *ACM Computing Surveys (CSUR)*.

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

[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.

[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.

[LH14] Brian R. Larson and John Hatlicff. Open patient-controlled analgesia infusion pump system requirements draft 0.11. URL: http://santoslab.org/pub/open-pca-pump/artifacts/Open-PCA-Pump-Requirements.pdf, Juin 2014.

[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.

[OG11] Frank J. Overdyk and Jesse J. Guerra. Improving outcomes in med-surg patients with opioid-induced respiratory depression. *American Nurse Today*, 6(11), Novembre 2011.

[PHR] Sam Procted, John Hatcliff, and Robby. Towards an aadl-based definition of app architecture for medical application platforms. In *Proceedings of the 2014 Software Engineering in Health-care (SEHC) Workshop at the International Symposium on Foundations of Health Information Engineering and Systems (FHIES 2014)*.

[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.

[SVC06] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, 2006.

[Tea] SPARK Team. Victor wrapper user manual. URL: http://docs.adacore.com/sparkdocs-docs/VictorWrapper_UM.htm.

[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

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
- **SAnToS Laboratory** - Laboratory for Specification, Analysis, and Transformation of Software

Appendix B

PCA pump prototype - simple, implemented, working pump

This appendix contains implemented, simple version of PCA Pump, created based on 3.1 and AADL models created by Brian Larson. Data types used by this pump are the same like translated from AADL models presented in appendix F.

```
1  with Base_Types;
2  with Pca_Types;
3  with Ice_Types;
4  --# inherit Ada.Real_Time,
5  --#         Ada.Synchronous_Task_Control,
6  --#         Base_Types,
7  --#         Pca_Types,
8  --#         Ice_Types,
9  --#         Pca_Engine;
10 package Pca_Operation
11 --# own protected Operate (suspendable);
12 --#     protected Fluid_Pulses : Integer_Array_Store(Priority => 10);
13 --#     protected Prescription : Pca_Types.Prescription_Store(Priority => 10);
14 --#     protected State : Pca_Types.Status_Type_Store(Priority => 10);
15 --#     protected Clinician_Bolus_Paused : Base_Types.Boolean_Store(Priority => 10);
16 --#     protected Clinician_Bolus_Duration : Ice_Types.Minute_Store(Priority => 10);
17 --#     task rc : Rate_Controller;
18 --#     task mdphw : Max_Drug_Per_Hour_Watcher;
19 --#     Last_Patient_Bolus;
20 --# initializes Last_Patient_Bolus;
21 is
22
23     subtype Integer_Array_Index is Integer range 1 .. 60*60;
24     type Integer_Array is array (Integer_Array_Index) of Integer;
25
26     protected type Integer_Array_Store
27     is
```

```
28          pragma Priority (10);
29
30          function Get(Ind : in Integer) return Integer;
31          --# global in Integer_Array_Store;
32
33          procedure Put(Ind : in Integer; Val : in Integer);
34          --# global in out Integer_Array_Store;
35          --# derives Integer_Array_Store from Integer_Array_Store, Ind, Val;
36
37          procedure Inc(Ind : in Integer);
38          --# global in out Integer_Array_Store;
39          --# derives Integer_Array_Store from Integer_Array_Store, Ind;
40
41          function Sum return Integer;
42          --# global in Integer_Array_Store;
43
44          procedure Pulse;
45          --# global in out Integer_Array_Store;
46          --# derives Integer_Array_Store from Integer_Array_Store;
47      private
48          TheStoredData : Integer_Array := Integer_Array'(others => 0);
49      end Integer_Array_Store;
50
51      function Get_Volume_Infused return Integer; -- microliters
52      --# global in     Fluid_Pulses;
53
54      function Get_State return Pca_Types.Status_Type;
55      --# global in     State;
56
57      procedure Panel_Set_Basal_Flow_Rate(Flow_Rate : Pca_Types.Flow_Rate);
58      --# global in out Prescription;
59
60      function Panel_Get_Basal_Flow_Rate return Pca_Types.Flow_Rate;
61      --# global in     Prescription;
62
63      procedure Panel_Set_Vtbi(Vtbi : Pca_Types.Drug_Volume);
64      --# global in out Prescription;
65
66      function Panel_Get_Vtbi return Pca_Types.Drug_Volume;
67      --# global in     Prescription;
68
69      procedure Panel_Set_Max_Drug_Per_Hour(Max_Drug_Per_Hour : Pca_Types.Drug_Volume);
70      --# global in out Prescription;
71
72      function Panel_Get_Max_Drug_Per_Hour return Pca_Types.Drug_Volume;
73      --# global in     Prescription;
74
75      procedure Panel_Set_Minimum_Time_Between_Bolus(Minimum_Time_Between_Bolus : Ice_Types.Minute);
76      --# global in out Prescription;
77
78      function Panel_Get_Minimum_Time_Between_Bolus return Ice_Types.Minute;
79      --# global in     Prescription;
80
81      procedure StartPump;
82      --# global out    Operate;
83      --#        out    State;
84      --# derives Operate from &
85      --#          State from ;
86
87      procedure StopPump;
88      --# global out    Operate;
89      --#        out    State;
90      --# derives Operate from &
91      --#          State from ;
92
```

```
93      procedure PatientBolus;
94      --# global in out State;
95      --#        in out Last_Patient_Bolus;
96      --#        in out Clinician_Bolus_Paused;
97      --#        in     Prescription;
98      --#        in     Ada.Real_Time.ClockTime;
99      --# derives State from State, Last_Patient_Bolus, Prescription, Ada.Real_Time.ClockTime &
100     --#         Last_Patient_Bolus from Last_Patient_Bolus, Prescription, Ada.Real_Time.ClockTime &
101     --#         Clinician_Bolus_Paused from Clinician_Bolus_Paused, State, Last_Patient_Bolus, Prescription,
        Ada.Real_Time.ClockTime;
102
103     procedure ClinicianBolus(Cb_Duration : in     Ice_Types.Minute);
104     --# global in out State;
105     --#        in out Clinician_Bolus_Duration;
106     --#        in out Clinician_Bolus_Paused;
107     --# pre Cb_Duration <= 6 * 60; -- from Requirements 4.3.5
108
109
110 private
111     task type Rate_Controller
112     --# global out Operate;
113     --#        in out Fluid_Pulses;
114     --#        in out Clinician_Bolus_Paused;
115     --#        in     Prescription;
116     --#        in out State;
117     --#        in     Ada.Real_Time.ClockTime;
118     --#        in     Clinician_Bolus_Duration;
119     --# derives Operate       from &
120     --#         Fluid_Pulses from Fluid_Pulses, State, Clinician_Bolus_Paused, Ada.Real_Time.ClockTime,
        Prescription, Clinician_Bolus_Duration &
121     --#         Clinician_Bolus_Paused from State, Clinician_Bolus_Paused, Ada.Real_Time.ClockTime,
        Prescription, Clinician_Bolus_Duration &
122     --#         State        from State, Prescription, Ada.Real_Time.ClockTime, Clinician_Bolus_Duration,
        Clinician_Bolus_Paused;
123     --# declare suspends => Operate;
124     is
125         pragma Priority (9);
126     end Rate_Controller;
127
128     task type Max_Drug_Per_Hour_Watcher
129     --# global in out Fluid_Pulses;
130     --#        in     Prescription;
131     --#        in out State;
132     --#        in     Ada.Real_Time.ClockTime;
133     --# derives Fluid_Pulses  from Fluid_Pulses &
134     --#         State         from Prescription, Fluid_Pulses, State &
135     --#         null          from Ada.Real_Time.ClockTime;
136     is
137         pragma Priority (9);
138     end Max_Drug_Per_Hour_Watcher;
139
140 end Pca_Operation;
141
142 with Ada.Synchronous_Task_Control,
143      Ada.Real_Time,
144      Base_Types,
145      Pca_Types,
146      Pca_Engine;
147 use type Ada.Real_Time.Time;
148 use type Pca_Types.Status_Type;
149
150 package body Pca_Operation
151 is
152     Operate : Ada.Synchronous_Task_Control.Suspension_Object;
153     rc : Rate_Controller;
```

```
154     mdphw : Max_Drug_Per_Hour_Watcher;
155
156     Fluid_Pulses : Integer_Array_Store;
157
158     State : Pca_Types.Status_Type_Store;
159
160     Prescription : Pca_Types.Prescription_Store;
161
162     Fluid_Pulse_Volume : constant Natural := 100; -- in microliters
163
164     Kvo_Rate : constant Pca_Types.Flow_Rate := 1; -- in mililiters
165
166     Bolus_Flow_Rate : constant Pca_Types.Flow_Rate := 100; -- in mililiters
167
168     Last_Patient_Bolus : Ada.Real_Time.Time := Ada.Real_Time.Time_First;
169
170     Clinician_Bolus_Duration : Ice_Types.Minute_Store;
171
172     Clinician_Bolus_Paused : Base_Types.Boolean_Store;
173
174     protected body Integer_Array_Store
175     is
176         function Get(Ind : in Integer) return Integer
177         --# global in TheStoredData;
178         is
179         begin
180             return TheStoredData(Ind);
181         end Get;
182
183
184         procedure Put(Ind : in Integer; Val : in Integer)
185           --# global in out TheStoredData;
186           --# derives TheStoredData from TheStoredData, Ind, Val;
187         is
188         begin
189             TheStoredData(Ind) := Val;
190         end Put;
191
192         procedure Inc(Ind : in Integer)
193           --# global in out TheStoredData;
194           --# derives TheStoredData from TheStoredData, Ind;
195         is
196         begin
197             TheStoredData(Ind) := TheStoredData(Ind) + Fluid_Pulse_Volume;
198         end Inc;
199
200         function Sum return Integer
201         --# global in TheStoredData;
202         is
203             Result : Integer := 0;
204         begin
205             for I in Integer_Array_Index loop
206                 --# assert I > 1 -> Result >= TheStoredData(I-1);
207                 Result := Result + TheStoredData(I);
208             end loop;
209             return Result;
210         end Sum;
211
212         procedure Pulse
213         --# global in out TheStoredData;
214         --# derives TheStoredData from TheStoredData;
215         is
216         begin
217             for I in Integer_Array_Index range 1 .. Integer_Array_Index'Last-1 loop
218                 --# assert I > 1 -> TheStoredData(I-1) = TheStoredData(I);
```

```
219                 TheStoredData(I) := TheStoredData(I+1);
220             end loop;
221             TheStoredData(Integer_Array_Index'Last) := 0;
222         end Pulse;
223
224     end Integer_Array_Store;
225
226     function Get_Time_Between_Activations(Flow_Rate : in Pca_Types.Flow_Rate) return Natural
227     is
228         Result : Natural;
229         Flow_Rate_In_Microliters : Natural;
230         Activations_Per_Hour : Natural;
231     begin
232         Flow_Rate_In_Microliters := Natural(Flow_Rate) * 1000; -- convert mL to uL
233         Activations_Per_Hour := Flow_Rate_In_Microliters / Fluid_Pulse_Volume;
234
235         -- miliseconds between activations
236         Result := ((60 * 60) * 1000) / Activations_Per_Hour;
237         return Result;
238     end Get_Time_Between_Activations;
239
240
241     function Get_Volume_Infused return Integer
242     is
243     begin
244         return Fluid_Pulses.Sum;
245     end Get_Volume_Infused;
246
247     function Get_State return Pca_Types.Status_Type
248     is
249         Current_State : Pca_Types.Status_Type;
250     begin
251         Current_State := State.Get;
252         return Current_State;
253     end Get_State;
254
255     procedure Panel_Set_Basal_Flow_Rate(Flow_Rate : Pca_Types.Flow_Rate)
256     is
257     begin
258         Prescription.Set_Basal_Flow_Rate(Flow_Rate);
259     end Panel_Set_Basal_Flow_Rate;
260
261     function Panel_Get_Basal_Flow_Rate return Pca_Types.Flow_Rate
262     is
263     begin
264         return Prescription.Get_Basal_Flow_Rate;
265     end Panel_Get_Basal_Flow_Rate;
266
267     procedure Panel_Set_Vtbi(Vtbi : Pca_Types.Drug_Volume)
268     is
269     begin
270         Prescription.Set_Vtbi(Vtbi);
271     end Panel_Set_Vtbi;
272
273     function Panel_Get_Vtbi return Pca_Types.Drug_Volume
274     is
275     begin
276         return Prescription.Get_Vtbi;
277     end Panel_Get_Vtbi;
278
279     procedure Panel_Set_Max_Drug_Per_Hour(Max_Drug_Per_Hour : Pca_Types.Drug_Volume)
280     is
281     begin
282         Prescription.Set_Max_Drug_Per_Hour(Max_Drug_Per_Hour);
283     end Panel_Set_Max_Drug_Per_Hour;
```

```
284
285     function Panel_Get_Max_Drug_Per_Hour return Pca_Types.Drug_Volume
286     is
287     begin
288         return Prescription.Get_Max_Drug_Per_Hour;
289     end Panel_Get_Max_Drug_Per_Hour;
290
291     procedure Panel_Set_Minimum_Time_Between_Bolus(Minimum_Time_Between_Bolus : Ice_Types.Minute)
292     is
293     begin
294         Prescription.Set_Minimum_Time_Between_Bolus(Minimum_Time_Between_Bolus);
295     end Panel_Set_Minimum_Time_Between_Bolus;
296
297     function Panel_Get_Minimum_Time_Between_Bolus return Ice_Types.Minute
298     is
299     begin
300         return Prescription.Get_Minimum_Time_Between_Bolus;
301     end Panel_Get_Minimum_Time_Between_Bolus;
302
303     procedure StartPump
304     is
305     begin
306         Ada.Synchronous_Task_Control.Set_True (Operate);
307         State.Put(Pca_Types.Basal);
308     end StartPump;
309
310     procedure StopPump
311     is
312     begin
313         Ada.Synchronous_Task_Control.Set_False (Operate);
314         State.put(Pca_Types.Stopped);
315     end StopPump;
316
317     procedure PatientBolus
318     is
319         Minimum_Time_Between_Bolus : Ice_Types.Minute;
320         Time_Now : Ada.Real_Time.Time;
321         Current_State : Pca_Types.Status_Type;
322     begin
323         Minimum_Time_Between_Bolus := Prescription.Get_Minimum_Time_Between_Bolus;
324         Time_Now := Ada.Real_Time.Clock;
325         if Last_Patient_Bolus + Ada.Real_Time.Milliseconds(Natural(Minimum_Time_Between_Bolus)*(60*1000)) <=
     Time_Now then
326             Last_Patient_Bolus := Time_Now;
327             Current_State := State.Get;
328             if Current_State = Pca_Types.Square_Bolus then
329                 Clinician_Bolus_Paused.Put(True);
330             end if;
331             State.Put(Pca_Types.Bolus);
332         end if;
333     end PatientBolus;
334
335     procedure ClinicianBolus(Cb_Duration : in     Ice_Types.Minute)
336     is
337         Current_State : Pca_Types.Status_Type;
338     begin
339         Current_State := State.Get;
340         if Current_State = Pca_Types.Basal then
341             Clinician_Bolus_Duration.Put(Cb_Duration);
342             State.Put(Pca_Types.Square_Bolus);
343         elsif Current_State = Pca_Types.Bolus then
344             Clinician_Bolus_Duration.Put(Cb_Duration);
345             Clinician_Bolus_Paused.Put(True);
346         end if;
347     end ClinicianBolus;
```

```
348
349
350     task body Rate_Controller
351     is
352         --Release_Time : Ada.Real_Time.Time;
353         Now : Ada.Real_Time.Time;
354         Period : Natural;
355         Last_Basal_Pulse : Ada.Real_Time.Time := Ada.Real_Time.Time_First; -- Ada.Real_Time.Clock - Ada.
        Real_Time.Milliseconds(1000 * 60 * 60);
356         Last_Kvo_Pulse : Ada.Real_Time.Time := Ada.Real_Time.Time_First; -- Ada.Real_Time.Clock - Ada.
        Real_Time.Milliseconds(1000 * 60 * 60);
357         Last_Patient_Bolus_Pulse : Ada.Real_Time.Time := Ada.Real_Time.Time_First; -- Ada.Real_Time.Clock -
        Ada.Real_Time.Milliseconds(1000 * 60 * 60);
358         Last_Clinician_Bolus_Pulse : Ada.Real_Time.Time := Ada.Real_Time.Time_First; -- Ada.Real_Time.Clock -
         Ada.Real_Time.Milliseconds(1000 * 60 * 60);
359         Patient_Bolus_Volume_Left : Natural := 0;
360         Clinicaian_Bolus_Vtbi : Pca_Types.Drug_Volume;
361         Clinician_Bolus_Volume_Left : Natural := 0;
362         Current_State : Pca_Types.Status_Type;
363         Flow_Rate : Pca_Types.Flow_Rate;
364         Drug_Volume : Pca_Types.Drug_Volume;
365         Clinician_Bolus_Paused_Temp : Boolean;
366         Clinician_Bolus_Duration_Temp : Ice_Types.Minute;
367     begin
368         loop
369             Ada.Synchronous_Task_Control.Suspend_Until_True (Operate); -- wait until user allows Pump to
        operate
370             Ada.Synchronous_Task_Control.Set_True (Operate); -- Keep the task running, the previous call will
         have set Operate to False.
371
372             Now := Ada.Real_Time.Clock;
373             Current_State := State.Get;
374
375             --# assert true;
376
377             case Current_State is
378             when Pca_Types.Stopped =>
379                 null;
380             when Pca_Types.KVO =>
381                 Period := Get_Time_Between_Activations(Kvo_Rate);
382                 if Last_Kvo_Pulse + Ada.Real_Time.Milliseconds(Period) <= Now then
383                     Last_Kvo_Pulse := Now;
384                     Fluid_Pulses.Inc(Integer_Array_Index'Last); -- each time round, update the volume infused
385                     Pca_Engine.Run_Pumping(100); -- and pump 0.1 ml
386                 end if;
387             when Pca_Types.Basal =>
388                 Flow_Rate := Prescription.Get_Basal_Flow_Rate;
389                 Period := Get_Time_Between_Activations(Flow_Rate);
390                 if Last_Basal_Pulse + Ada.Real_Time.Milliseconds(Period) <= Now then
391                     Last_Basal_Pulse := Now;
392                     Fluid_Pulses.Inc(Integer_Array_Index'Last); -- each time round, update the volume infused
393                     Pca_Engine.Run_Pumping(100); -- and pump 0.1 ml
394                 end if;
395             when Pca_Types.Bolus =>
396                 -- basal
397                 Flow_Rate := Prescription.Get_Basal_Flow_Rate;
398                 Period := Get_Time_Between_Activations(Flow_Rate);
399                 if Last_Basal_Pulse + Ada.Real_Time.Milliseconds(Period) <= Now then
400                     Last_Basal_Pulse := Now;
401                     Fluid_Pulses.Inc(Integer_Array_Index'Last); -- each time round, update the volume infused
402                     Pca_Engine.Run_Pumping(100); -- and pump 0.1 ml
403                 end if;
404
405                 -- patient
406                 Period := Get_Time_Between_Activations(Bolus_Flow_Rate);
```

```
407                 if Last_Patient_Bolus_Pulse + Ada.Real_Time.Milliseconds(Period) <= Now then
408                     Last_Patient_Bolus_Pulse := Now;
409
410                     if Patient_Bolus_Volume_Left = 0 then
411                         Drug_Volume := Prescription.Get_Vtbi;
412                         Patient_Bolus_Volume_Left := Natural(Drug_Volume);
413                         Patient_Bolus_Volume_Left := Patient_Bolus_Volume_Left * 1000; -- convert to
    microliters
414                     end if;
415
416                     Fluid_Pulses.Inc(Integer_Array_Index'Last); -- each time round, update the volume infused
417                     Pca_Engine.Run_Pumping(100); -- and pump 0.1 ml
418
419                     Patient_Bolus_Volume_Left := Patient_Bolus_Volume_Left - 100;
420                     if Patient_Bolus_Volume_Left = 0 then
421                         Clinician_Bolus_Paused_Temp := Clinician_Bolus_Paused.Get;
422                         if Clinician_Bolus_Paused_Temp then
423                             State.Put(Pca_Types.Square_Bolus);
424                             Clinician_Bolus_Paused.Put(False);
425                         else
426                             State.Put(Pca_Types.Basal);
427                         end if;
428                     end if;
429                 end if;
430             when Pca_Types.Square_Bolus =>
431                 -- basal
432                 Flow_Rate := Prescription.Get_Basal_Flow_Rate;
433                 Period := Get_Time_Between_Activations(Flow_Rate);
434                 if Last_Basal_Pulse + Ada.Real_Time.Milliseconds(Period) <= Now then
435                     Last_Basal_Pulse := Now;
436                     Fluid_Pulses.Inc(Integer_Array_Index'Last); -- each time round, update the volume infused
437                     Pca_Engine.Run_Pumping(100); -- and pump 0.1 ml
438                 end if;
439
440                 -- clinician
441                 Clinician_Bolus_Duration_Temp := Clinician_Bolus_Duration.Get;
442                 Clinicaian_Bolus_Vtbi := Prescription.Get_Vtbi;
443                 Period := Get_Time_Between_Activations(Pca_Types.Flow_Rate(Natural(Clinicaian_Bolus_Vtbi) *
    (60/Natural(Clinician_Bolus_Duration_Temp))));
444                 if Last_Clinician_Bolus_Pulse + Ada.Real_Time.Milliseconds(Period) <= Now then
445                     Last_Clinician_Bolus_Pulse := Now;
446
447                     if Clinician_Bolus_Volume_Left = 0 then
448                         Clinicaian_Bolus_Vtbi := Prescription.Get_Vtbi;
449                         Clinician_Bolus_Volume_Left := Natural(Clinicaian_Bolus_Vtbi) * 1000;  -- in
    microliters
450                     end if;
451
452                     Fluid_Pulses.Inc(Integer_Array_Index'Last); -- each time round, update the volume infused
453                     Pca_Engine.Run_Pumping(100); -- and pump 0.1 ml
454
455                     Clinician_Bolus_Volume_Left := Clinician_Bolus_Volume_Left - 100;
456                     if Clinician_Bolus_Volume_Left = 0 then
457                         State.Put(Pca_Types.Basal);
458                     end if;
459                 end if;
460             end case;
461         end loop;
462     end Rate_Controller;
463
464     task body Max_Drug_Per_Hour_Watcher is
465         Release_Time : Ada.Real_Time.Time;
466         Period : constant Integer := 1000; -- update in every second
467         Volume_Infused : Integer;
468         Max_Drug_Per_Hour : Pca_Types.Drug_Volume;
```

```
469     begin
470         loop
471             --# assert true;
472 
473             Release_Time := Ada.Real_Time.Clock; -- must be simple assignment
474             Release_Time := Release_Time + Ada.Real_Time.Milliseconds(Period);
475             delay until Release_Time;
476             Fluid_Pulses.Pulse; -- each time round, update the volume infused moving window
477             Max_Drug_Per_Hour := Prescription.Get_Max_Drug_Per_Hour;
478             Volume_Infused := Get_Volume_Infused;
479             if Volume_Infused > (Integer(Max_Drug_Per_Hour)*1000) then -- convert to microliters
480                 State.Put(Pca_Types.KVO);
481             end if;
482         end loop;
483     end Max_Drug_Per_Hour_Watcher;
484 end Pca_Operation;
```

Listing B.1: `Pca_Operation` package

```
1  with Ada.Real_Time;
2  use type Ada.Real_Time.Time;
3  --# inherit Ada.Real_Time;
4  package Pca_Engine
5  is
6      procedure Start_Pumping;
7  
8      procedure Stop_Pumping;
9  
10     procedure Run_Pumping(Microliters : in Natural);
11     --# global in Ada.Real_Time.ClockTime;
12     --# derives null from Microliters, Ada.Real_Time.ClockTime;
13     --# pre Microliters > 0;
14 
15 end Pca_Engine;
16 
17 with Ada.Strings.Unbounded;
18 use Ada.Strings.Unbounded;
19 with Ada.Text_IO.Unbounded_IO;
20 use Ada.Text_IO;
21 
22 package body Pca_Engine
23 --# hide Pca_Engine
24 is
25     GPIO_Path : constant String := "/sys/class/gpio/";
26     Status_File_Path : constant String := "/home/root/pump_status.txt";
27 
28     GPIO_Export_File_Path : constant String := GPIO_Path & "export";
29     GPIO_Unexport_File_Path : constant String := GPIO_Path & "unexport";
30     GPIO162_Direction_File_Path : constant String := GPIO_Path & "gpio162/direction";
31     GPIO162_Value_File_Path : constant String := GPIO_Path & "gpio162/value";
32 
33     procedure Start_Pumping
34     is
35         F    : Ada.Text_IO.File_Type;
36         Data : Unbounded_String := To_Unbounded_String("pumping");
37         File_Export : Ada.Text_IO.File_Type;
38         File_Direction : Ada.Text_IO.File_Type;
39     begin
40         Create(File_Export, Ada.Text_IO.Out_File, GPIO_Export_File_Path);
41         Put_Line(File_Export, "162");
42         Close(File_Export);
43 
44         Create(File_Direction, Ada.Text_IO.Out_File, GPIO162_Direction_File_Path);
```

```
45          Put_Line(File_Direction, "out");
46          Close(File_Direction);
47
48          Create(F, Ada.Text_IO.Out_File, Status_File_Path);
49          Unbounded_IO.Put_Line(F, Data);
50          Put_Line("Pumping...");
51          Close(F);
52      end Start_Pumping;
53
54
55      procedure Stop_Pumping is
56          F    : Ada.Text_IO.File_Type;
57          Data : Unbounded_String := To_Unbounded_String("IDLE");
58          File_Unexport : Ada.Text_IO.File_Type;
59      begin
60          Create(File_Unexport, Ada.Text_IO.Out_File, GPIO_Unexport_File_Path);
61          Put_Line(File_Unexport, "162");
62          Close(File_Unexport);
63
64          Create(F, Ada.Text_IO.Out_File, Status_File_Path);
65          Unbounded_IO.Put_Line(F, Data);
66          Put_Line("Idle...");
67          Close(F);
68      end Stop_Pumping;
69
70      procedure Write_Signal(Signal : in Integer) is
71          File : Ada.Text_IO.File_Type;
72          Data : Unbounded_String;
73      begin
74          Ada.Text_IO.Open (File => File,
75                            Mode => Ada.Text_IO.Out_File,
76                            Name => GPIO162_Value_File_Path);
77          if Signal = 1 then
78              Data := To_Unbounded_String("1");
79          else
80              Data := To_Unbounded_String("0");
81          end if;
82          Unbounded_IO.Put_Line(File, Data);
83          Ada.Text_IO.Close(File);
84      end Write_Signal;
85
86      procedure Run_Pumping(Microliters : in Natural)
87      is
88          Interval_High: constant Ada.Real_Time.Time_Span := Ada.Real_Time.Microseconds(9000);
89          Interval_Low: constant Ada.Real_Time.Time_Span := Ada.Real_Time.Microseconds(1000);
90          Next_Time: Ada.Real_Time.Time;
91      begin
92          Next_Time := Ada.Real_Time.Clock;
93          Start_Pumping;
94          for I in Integer range 1 .. (15*Microliters) loop
95              Next_Time := Next_Time + Interval_High;
96              Write_Signal(1);
97              delay until Next_Time;
98              Next_Time := Next_Time + Interval_Low;
99              Write_Signal(0);
100             delay until Next_Time;
101         end loop;
102         Stop_Pumping;
103     end Run_Pumping;
104
105 end Pca_Engine;
```

Listing B.2: `Pca_Engine` package

```
with Pca_Operation;
with Pca_Types;
with Ice_Types;
with Ada.Text_IO;
with Ada.Float_Text_IO;
--# inherit Pca_Operation,
--#         Ada.Real_Time,
--#         Pca_Types,
--#         Ice_Types;
--# main_program;
procedure Main
is
    pragma Priority (10);
    Input : String(1..10) := (others => ' ');
    Input_Last : Integer;
    Option : Integer;
    use Ada.Text_IO;
begin

    Put_Line("Menu: ");
    Put_Line("0 - Enter prescription");
    Put_Line("1 - Start PCA Pump");
    Put_Line("2 - Stop PCA Pump");
    Put_Line("3 - Display Volume Infused");
    Put_Line("4 - Display Prescription");
    Put_Line("5 - Set Basal Flow Rate");
    Put_Line("6 - Patient bolus");
    Put_Line("7 - Clinician bolus");
    Put_Line("8 - Display Current State");
    loop
        Input := (others => ' ');
        Get_Line(Input, Input_Last);
        Option := Integer'Value(Input);
        case Option is
            when 0 =>
                Put_Line("Enter Basal Flow Rate (ml/hr): ");
                Input := (others => ' ');
                Get_Line(Input, Input_Last);
                Pca_Operation.Panel_Set_Basal_Flow_Rate(Pca_Types.Flow_Rate(Integer'Value(Input)));
                Put_Line("Enter Volume to be infused during patient bolus (ml): ");
                Input := (others => ' ');
                Get_Line(Input, Input_Last);
                Pca_Operation.Panel_Set_Vtbi(Pca_Types.Drug_Volume(Integer'Value(Input)));
                Put_Line("Enter Max Drug Per Hour (ml): ");
                Input := (others => ' ');
                Get_Line(Input, Input_Last);
                Pca_Operation.Panel_Set_Max_Drug_Per_Hour(Pca_Types.Drug_Volume(Integer'Value(Input)));
                Put_Line("Enter Minimum Time Between Boluses (min.): ");
                Input := (others => ' ');
                Get_Line(Input, Input_Last);
                Pca_Operation.Panel_Set_Minimum_Time_Between_Bolus(Ice_Types.Minute(Integer'Value(Input)));
            when 1 =>
                Pca_Operation.StartPump;
                Put_Line("Pump Started");
            when 2 =>
                Pca_Operation.StopPump;
                Put_Line("Pump Stopped");
            when 3 =>
                Put("Volume Infused (ml):");
                Ada.Float_Text_IO.Put(Float(Pca_Operation.Get_Volume_Infused) / Float(1000), AFT=>2, EXP=>0);
                Put_Line("");
            when 4 =>
                Put_Line("Current Basal Flow Rate (ml/hr): " & Integer'Image(Integer(Pca_Operation.
    Panel_Get_Basal_Flow_Rate)));
```

```
64                Put_Line("Current Volume to be Infused (ml): " & Integer'Image(Integer(Pca_Operation.
      Panel_Get_Vtbi)));
65                Put_Line("Current Max Drug Per Hour (ml): " & Integer'Image(Integer(Pca_Operation.
      Panel_Get_Max_Drug_Per_Hour)));
66                Put_Line("Current minimum time between bolus (min): " & Integer'Image(Integer(Pca_Operation.
      Panel_Get_Minimum_Time_Between_Bolus)));
67             when 5 =>
68                Put_Line("Enter Basal Flow Rate (ml/hr): ");
69                Input := (others => ' ');
70                Get_Line(Input, Input_Last);
71                Pca_Operation.Panel_Set_Basal_Flow_Rate(Pca_Types.Flow_Rate(Integer'Value(Input)));
72             when 6 =>
73                Pca_Operation.PatientBolus;
74             when 7 =>
75                Put_Line("Enter Duration (min): ");
76                Input := (others => ' ');
77                Get_Line(Input, Input_Last);
78                Pca_Operation.ClinicianBolus(Ice_Types.Minute'Value(Input));
79             when 8 =>
80                case Pca_Operation.Get_State is
81                   when Pca_Types.Stopped =>
82                      Put_Line("Stopped");
83                   when Pca_Types.Bolus =>
84                      Put_Line("Bolus");
85                   when Pca_Types.Basal =>
86                      Put_Line("Basal");
87                   when Pca_Types.KVO =>
88                      Put_Line("KVO");
89                   when Pca_Types.Square_Bolus =>
90                      Put_Line("Square Bolus");
91                end case;
92             when others => exit;
93          end case;
94       end loop;
95       Put_Line("Program end");
96 end Main;
```

Listing B.3: `Main` procedure

Appendix C

PCA pump prototype verification - POGS report

```
1  ------------------------------------------------------------------------------
2                            Semantic Analysis Summary
3                                  POGS GPL 2012
4              Copyright (C) 2012 Altran Praxis Limited, Bath, U.K.
5  ------------------------------------------------------------------------------
6
7  Summary of:
8
9  Verification Condition files (.vcg)
10 Simplified Verification Condition files (.siv)
11 Victor result files (.vct)
12 Riposte result files (.rsm)
13 Proof Logs (.plg)
14 Dead Path Conjecture files (.dpc)
15 Summary Dead Path files (.sdp)
16
17 "status" column keys:
18     1st character:
19         '-' - No VC
20         'S' - No SIV
21         'U' - Undischarged
22         'E' - Proved by Examiner
23         'I' - Proved by Simplifier by Inference
24         'X' - Proved by Simplifier by Contradiction
25         'P' - Proved by Simplifier using User Defined Proof Rules
26         'V' - Proved by Victor
27         'O' - Proved by Riposte
28         'C' - Proved by Checker
29         'R' - Proved by Review
30         'F' - VC is False
31     2nd character:
32         '-' - No DPC
33         'S' - No SDP
34         'U' - Unchecked
```

```
        'D' - Dead path
        'L' - Live path

in the directory:
/Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar

Summary produced: 25-JUL-2014 14:16:50.13

File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/clinicianbolus.vcg
procedure Pca_Operation.ClinicianBolus

VCs generated 25-JUL-2014 14:16:24

VCs simplified 25-JUL-2014 14:16:28

File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/clinicianbolus.dpc
DPCs generated 25-JUL-2014 14:16:24

DPC ZombieScoped 25-JUL-2014  14:16:2

VCs for procedure_clinicianbolus :
 ------------------------------------------------------------------------------
| #   | From  | To                  | Proved By          | Dead Path | Status |
|------------------------------------------------------------------------------
| 1   | start | rtc check @ 198     | Inference          | Unchecked |   IU   |
| 2   | start | rtc check @ 200     | Inference          | Unchecked |   IU   |
| 3   | start | rtc check @ 201     | Inference          | Unchecked |   IU   |
| 4   | start | rtc check @ 203     | Inference          | Unchecked |   IU   |
| 5   | start |    assert @ finish  | Examiner           | Live      |   EL   |
| 6   | start |    assert @ finish  | Examiner           | Live      |   EL   |
| 7   | start |    assert @ finish  | Examiner           | Live      |   EL   |
 ------------------------------------------------------------------------------


File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/get_state.vcg
function Pca_Operation.Get_State

VCs generated 25-JUL-2014 14:16:24

VCs simplified 25-JUL-2014 14:16:28

File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/get_state.dpc
DPCs generated 25-JUL-2014 14:16:24

DPC ZombieScoped 25-JUL-2014  14:16:2

VCs for function_get_state :
 ------------------------------------------------------------------------------
| #   | From  | To                  | Proved By          | Dead Path | Status |
|------------------------------------------------------------------------------
| 1   | start | rtc check @ 110     | Inference          | Unchecked |   IU   |
| 2   | start |    assert @ finish  | Inference          | Live      |   IL   |
 ------------------------------------------------------------------------------


File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/get_time_between_activations.vcg
function Pca_Operation.Get_Time_Between_Activations

VCs generated 25-JUL-2014 14:16:24

VCs simplified 25-JUL-2014 14:16:28

File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/get_time_between_activations.dpc
DPCs generated 25-JUL-2014 14:16:24

```

```
DPC ZombieScoped 25-JUL-2014  14:16:2

VCs for function_get_time_between_activations :
 ------------------------------------------------------------------------------
| #   | From  | To                  | Proved By          | Dead Path | Status |
|------------------------------------------------------------------------------
| 1   | start | rtc check @ 91      | Undischarged       | Unchecked |   UU   |
| 2   | start | rtc check @ 92      | Inference          | Unchecked |   IU   |
| 3   | start | rtc check @ 95      | Undischarged       | Unchecked |   UU   |
| 4   | start |    assert @ finish  | Inference          | Live      |   IL   |
 ------------------------------------------------------------------------------


File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/get_volume_infused.vcg
function Pca_Operation.Get_Volume_Infused

VCs generated 25-JUL-2014 14:16:24

VCs simplified 25-JUL-2014 14:16:29

File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/get_volume_infused.dpc
DPCs generated 25-JUL-2014 14:16:24

DPC ZombieScoped 25-JUL-2014  14:16:2

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


File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/integer_array_store/get.vcg
function Pca_Operation.Integer_Array_Store.Get

VCs generated 25-JUL-2014 14:16:24

VCs simplified 25-JUL-2014 14:16:29

File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/integer_array_store/get.dpc
DPCs generated 25-JUL-2014 14:16:24

DPC ZombieScoped 25-JUL-2014  14:16:2

VCs for function_get :
 ------------------------------------------------------------------------------
| #   | From  | To                  | Proved By          | Dead Path | Status |
|------------------------------------------------------------------------------
| 1   | start | rtc check @ 39      | Undischarged       | Unchecked |   UU   |
| 2   | start |    assert @ finish  | Inference          | Live      |   IL   |
| 3   |       | refinement          | Examiner           | No DPC    |   E-   |
| 4   |       | refinement          | Examiner           | No DPC    |   E-   |
 ------------------------------------------------------------------------------


File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/integer_array_store/inc.vcg
procedure Pca_Operation.Integer_Array_Store.Inc

VCs generated 25-JUL-2014 14:16:24

VCs simplified 25-JUL-2014 14:16:29

File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/integer_array_store/inc.dpc
DPCs generated 25-JUL-2014 14:16:24
```

```
DPC ZombieScoped 25-JUL-2014  14:16:2

VCs for procedure_inc :
 ------------------------------------------------------------------------------
| #   | From  | To                  | Proved By          | Dead Path | Status |
|------------------------------------------------------------------------------
| 1   | start | rtc check @ 56      | Undischarged       | Unchecked |   UU   |
| 2   | start |    assert @ finish  | Examiner           | Live      |   EL   |
| 3   |       | refinement          | Examiner           | No DPC    |   E-   |
| 4   |       | refinement          | Examiner           | No DPC    |   E-   |
 ------------------------------------------------------------------------------


File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/integer_array_store/pulse.vcg
procedure Pca_Operation.Integer_Array_Store.Pulse

VCs generated 25-JUL-2014 14:16:24

VCs simplified 25-JUL-2014 14:16:29

File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/integer_array_store/pulse.dpc
DPCs generated 25-JUL-2014 14:16:24

DPC ZombieScoped 25-JUL-2014  14:16:2

VCs for procedure_pulse :
 ------------------------------------------------------------------------------
| #   | From  | To                  | Proved By          | Dead Path | Status |
|------------------------------------------------------------------------------
| 1   | start | rtc check @ 76      | Inference          | Unchecked |   IU   |
| 2   | start | rtc check @ 76      | Inference          | Unchecked |   IU   |
| 3   | start |    assert @ 77      | Inference          | Live      |   IL   |
| 4   | 77    |    assert @ 77      | Inference          | Live      |   IL   |
| 5   | 77    | rtc check @ 78      | Inference          | Unchecked |   IU   |
| 6   | start | rtc check @ 80      | Inference          | Unchecked |   IU   |
| 7   | 77    | rtc check @ 80      | Inference          | Unchecked |   IU   |
| 8   | start |    assert @ finish  | Examiner           | Dead      |   ED   |
| 9   | 77    |    assert @ finish  | Examiner           | Live      |   EL   |
| 10  |       | refinement          | Examiner           | No DPC    |   E-   |
| 11  |       | refinement          | Examiner           | No DPC    |   E-   |
 ------------------------------------------------------------------------------


File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/integer_array_store/put.vcg
procedure Pca_Operation.Integer_Array_Store.Put

VCs generated 25-JUL-2014 14:16:24

VCs simplified 25-JUL-2014 14:16:29

File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/integer_array_store/put.dpc
DPCs generated 25-JUL-2014 14:16:24

DPC ZombieScoped 25-JUL-2014  14:16:2

VCs for procedure_put :
 ------------------------------------------------------------------------------
| #   | From  | To                  | Proved By          | Dead Path | Status |
|------------------------------------------------------------------------------
| 1   | start | rtc check @ 48      | Undischarged       | Unchecked |   UU   |
| 2   | start |    assert @ finish  | Examiner           | Live      |   EL   |
| 3   |       | refinement          | Examiner           | No DPC    |   E-   |
| 4   |       | refinement          | Examiner           | No DPC    |   E-   |
 ------------------------------------------------------------------------------
```

```
File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/integer_array_store/sum.vcg
function Pca_Operation.Integer_Array_Store.Sum

VCs generated 25-JUL-2014 14:16:24

VCs simplified 25-JUL-2014 14:16:30

File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/integer_array_store/sum.dpc
DPCs generated 25-JUL-2014 14:16:24

DPC ZombieScoped 25-JUL-2014  14:16:3

VCs for function_sum :
 ------------------------------------------------------------------------------
| #   | From  | To                  | Proved By          | Dead Path | Status |
|------------------------------------------------------------------------------
| 1   | start |    assert @ 65      | Inference          | Live      |   IL   |
| 2   | 65    |    assert @ 65      | Undischarged       | Live      |   UL   |
| 3   | 65    | rtc check @ 66      | Undischarged       | Unchecked |   UU   |
| 4   | 65    |    assert @ finish  | Inference          | Live      |   IL   |
| 5   |       | refinement          | Examiner           | No DPC    |   E-   |
| 6   |       | refinement          | Examiner           | No DPC    |   E-   |
 ------------------------------------------------------------------------------


File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/max_drug_per_hour_watcher.vcg
task_type Pca_Operation.Max_Drug_Per_Hour_Watcher

VCs generated 25-JUL-2014 14:16:25

VCs simplified 25-JUL-2014 14:16:30

File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/max_drug_per_hour_watcher.dpc
DPCs generated 25-JUL-2014 14:16:25

DPC ZombieScoped 25-JUL-2014  14:16:3

VCs for task_type_max_drug_per_hour_watcher :
 ------------------------------------------------------------------------------
| #   | From  | To                  | Proved By          | Dead Path | Status |
|------------------------------------------------------------------------------
| 1   | start |    assert @ 330     | Examiner           | Live      |   EL   |
| 2   | 330   |    assert @ 330     | Examiner           | Live      |   EL   |
| 3   | 330   |    assert @ 330     | Examiner           | Live      |   EL   |
| 4   | 330   | rtc check @ 332     | Inference          | Unchecked |   IU   |
| 5   | 330   | rtc check @ 333     | Inference          | Unchecked |   IU   |
| 6   | 330   | rtc check @ 333     | Undischarged       | Unchecked |   UU   |
| 7   | 330   | rtc check @ 336     | Inference          | Unchecked |   IU   |
| 8   | 330   | rtc check @ 337     | Inference          | Unchecked |   IU   |
| 9   | 330   | rtc check @ 338     | Inference          | Unchecked |   IU   |
| 10  | 330   | rtc check @ 339     | Inference          | Unchecked |   IU   |
| 11  | 330   |    assert @ finish  | Examiner           | Dead      |   ED   |
| 12  | 330   |    assert @ finish  | Examiner           | Dead      |   ED   |
 ------------------------------------------------------------------------------


File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/panel_get_basal_flow_rate.vcg
function Pca_Operation.Panel_Get_Basal_Flow_Rate

VCs generated 25-JUL-2014 14:16:24

VCs simplified 25-JUL-2014 14:16:30
```

```
File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/panel_get_basal_flow_rate.dpc
DPCs generated 25-JUL-2014 14:16:24

DPC ZombieScoped 25-JUL-2014  14:16:3

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


File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/panel_get_max_drug_per_hour.vcg
function Pca_Operation.Panel_Get_Max_Drug_Per_Hour

VCs generated 25-JUL-2014 14:16:24

VCs simplified 25-JUL-2014 14:16:30

File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/panel_get_max_drug_per_hour.dpc
DPCs generated 25-JUL-2014 14:16:24

DPC ZombieScoped 25-JUL-2014  14:16:3

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


File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/
     panel_get_minimum_time_between_bolus.vcg
function Pca_Operation.Panel_Get_Minimum_Time_Between_Bolus

VCs generated 25-JUL-2014 14:16:24

VCs simplified 25-JUL-2014 14:16:31

File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/
     panel_get_minimum_time_between_bolus.dpc
DPCs generated 25-JUL-2014 14:16:24

DPC ZombieScoped 25-JUL-2014  14:16:3

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


File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/panel_get_vtbi.vcg
function Pca_Operation.Panel_Get_Vtbi

VCs generated 25-JUL-2014 14:16:24

VCs simplified 25-JUL-2014 14:16:31

File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/panel_get_vtbi.dpc
DPCs generated 25-JUL-2014 14:16:24

```

```
DPC ZombieScoped 25-JUL-2014  14:16:3

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


File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/panel_set_basal_flow_rate.vcg
procedure Pca_Operation.Panel_Set_Basal_Flow_Rate

VCs generated 25-JUL-2014 14:16:24

VCs simplified 25-JUL-2014 14:16:31

File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/panel_set_basal_flow_rate.dpc
DPCs generated 25-JUL-2014 14:16:24

DPC ZombieScoped 25-JUL-2014  14:16:3

VCs for procedure_panel_set_basal_flow_rate :
 ------------------------------------------------------------------------------
| #   | From  | To                 | Proved By          | Dead Path | Status |
|------------------------------------------------------------------------------
| 1   | start | rtc check @ 117     | Inference          | Unchecked |   IU   |
| 2   | start |    assert @ finish  | Examiner           | Live      |   EL   |
 ------------------------------------------------------------------------------


File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/panel_set_max_drug_per_hour.vcg
procedure Pca_Operation.Panel_Set_Max_Drug_Per_Hour

VCs generated 25-JUL-2014 14:16:24

VCs simplified 25-JUL-2014 14:16:31

File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/panel_set_max_drug_per_hour.dpc
DPCs generated 25-JUL-2014 14:16:24

DPC ZombieScoped 25-JUL-2014  14:16:3

VCs for procedure_panel_set_max_drug_per_hour :
 ------------------------------------------------------------------------------
| #   | From  | To                 | Proved By          | Dead Path | Status |
|------------------------------------------------------------------------------
| 1   | start | rtc check @ 141     | Inference          | Unchecked |   IU   |
| 2   | start |    assert @ finish  | Examiner           | Live      |   EL   |
 ------------------------------------------------------------------------------


File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/
     panel_set_minimum_time_between_bolus.vcg
procedure Pca_Operation.Panel_Set_Minimum_Time_Between_Bolus

VCs generated 25-JUL-2014 14:16:24

VCs simplified 25-JUL-2014 14:16:31

File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/
     panel_set_minimum_time_between_bolus.dpc
DPCs generated 25-JUL-2014 14:16:24

DPC ZombieScoped 25-JUL-2014  14:16:3
```

```
VCs for procedure_panel_set_minimum_time_between_bolus :
 ------------------------------------------------------------------------------
| #   | From  | To                  | Proved By          | Dead Path | Status |
|------------------------------------------------------------------------------
| 1   | start | rtc check @ 153     | Inference          | Unchecked |   IU   |
| 2   | start |    assert @ finish  | Examiner           | Live      |   EL   |
 ------------------------------------------------------------------------------


File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/panel_set_vtbi.vcg
procedure Pca_Operation.Panel_Set_Vtbi

VCs generated 25-JUL-2014 14:16:24

VCs simplified 25-JUL-2014 14:16:31

File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/panel_set_vtbi.dpc
DPCs generated 25-JUL-2014 14:16:24

DPC ZombieScoped 25-JUL-2014  14:16:3

VCs for procedure_panel_set_vtbi :
 ------------------------------------------------------------------------------
| #   | From  | To                  | Proved By          | Dead Path | Status |
|------------------------------------------------------------------------------
| 1   | start | rtc check @ 129     | Inference          | Unchecked |   IU   |
| 2   | start |    assert @ finish  | Examiner           | Live      |   EL   |
 ------------------------------------------------------------------------------


File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/patientbolus.vcg
procedure Pca_Operation.PatientBolus

VCs generated 25-JUL-2014 14:16:24

VCs simplified 25-JUL-2014 14:16:32

File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/patientbolus.dpc
DPCs generated 25-JUL-2014 14:16:24

DPC ZombieScoped 25-JUL-2014  14:16:3

VCs for procedure_patientbolus :
 ------------------------------------------------------------------------------
| #   | From  | To                  | Proved By          | Dead Path | Status |
|------------------------------------------------------------------------------
| 1   | start | rtc check @ 182     | Inference          | Unchecked |   IU   |
| 2   | start | rtc check @ 183     | Inference          | Unchecked |   IU   |
| 3   | start | rtc check @ 184     | Inference          | Unchecked |   IU   |
| 4   | start | rtc check @ 184     | Undischarged       | Unchecked |   UU   |
| 5   | start | rtc check @ 185     | Inference          | Unchecked |   IU   |
| 6   | start | rtc check @ 186     | Inference          | Unchecked |   IU   |
| 7   | start | rtc check @ 190     | Inference          | Unchecked |   IU   |
| 8   | start | rtc check @ 190     | Inference          | Unchecked |   IU   |
| 9   | start |    assert @ finish  | Examiner           | Live      |   EL   |
| 10  | start |    assert @ finish  | Examiner           | Live      |   EL   |
| 11  | start |    assert @ finish  | Examiner           | Live      |   EL   |
 ------------------------------------------------------------------------------


File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/rate_controller.vcg
task_type Pca_Operation.Rate_Controller

VCs generated 25-JUL-2014 14:16:24
```

VCs simplified 25-JUL-2014 14:16:39

File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/rate_controller.dpc
DPCs generated 25-JUL-2014 14:16:24

DPC ZombieScoped 25-JUL-2014 14:16:3

VCs **for** task_type_rate_controller :

#	From	To	Proved By	Dead Path	Status
1	start	rtc check @ 231	Inference	Unchecked	IU
2	234	rtc check @ 231	Inference	Unchecked	IU
3	234	rtc check @ 231	Inference	Unchecked	IU
4	234	rtc check @ 231	Inference	Unchecked	IU
5	234	rtc check @ 231	Inference	Unchecked	IU
6	234	rtc check @ 231	Inference	Unchecked	IU
7	234	rtc check @ 231	Inference	Unchecked	IU
8	234	rtc check @ 231	Inference	Unchecked	IU
9	234	rtc check @ 231	Inference	Unchecked	IU
10	234	rtc check @ 231	Inference	Unchecked	IU
11	234	rtc check @ 231	Inference	Unchecked	IU
12	234	rtc check @ 231	Inference	Unchecked	IU
13	234	rtc check @ 231	Inference	Unchecked	IU
14	234	rtc check @ 231	Inference	Unchecked	IU
15	234	rtc check @ 231	Inference	Unchecked	IU
16	234	rtc check @ 231	Inference	Unchecked	IU
17	234	rtc check @ 231	Inference	Unchecked	IU
18	234	rtc check @ 231	Inference	Unchecked	IU
19	234	rtc check @ 231	Inference	Unchecked	IU
20	234	rtc check @ 231	Inference	Unchecked	IU
21	234	rtc check @ 231	Inference	Unchecked	IU
22	234	rtc check @ 231	Inference	Unchecked	IU
23	234	rtc check @ 231	Inference	Unchecked	IU
24	234	rtc check @ 231	Inference	Unchecked	IU
25	234	rtc check @ 231	Inference	Unchecked	IU
26	234	rtc check @ 231	Inference	Unchecked	IU
27	234	rtc check @ 231	Inference	Unchecked	IU
28	234	rtc check @ 231	Inference	Unchecked	IU
29	234	rtc check @ 231	Inference	Unchecked	IU
30	234	rtc check @ 231	Inference	Unchecked	IU
31	start	rtc check @ 232	Inference	Unchecked	IU
32	234	rtc check @ 232	Inference	Unchecked	IU
33	234	rtc check @ 232	Inference	Unchecked	IU
34	234	rtc check @ 232	Inference	Unchecked	IU
35	234	rtc check @ 232	Inference	Unchecked	IU
36	234	rtc check @ 232	Inference	Unchecked	IU
37	234	rtc check @ 232	Inference	Unchecked	IU
38	234	rtc check @ 232	Inference	Unchecked	IU
39	234	rtc check @ 232	Inference	Unchecked	IU
40	234	rtc check @ 232	Inference	Unchecked	IU
41	234	rtc check @ 232	Inference	Unchecked	IU
42	234	rtc check @ 232	Inference	Unchecked	IU
43	234	rtc check @ 232	Inference	Unchecked	IU
44	234	rtc check @ 232	Inference	Unchecked	IU
45	234	rtc check @ 232	Inference	Unchecked	IU
46	234	rtc check @ 232	Inference	Unchecked	IU
47	234	rtc check @ 232	Inference	Unchecked	IU
48	234	rtc check @ 232	Inference	Unchecked	IU
49	234	rtc check @ 232	Inference	Unchecked	IU
50	234	rtc check @ 232	Inference	Unchecked	IU
51	234	rtc check @ 232	Inference	Unchecked	IU
52	234	rtc check @ 232	Inference	Unchecked	IU
53	234	rtc check @ 232	Inference	Unchecked	IU

54	234	rtc check @ 232	Inference	Unchecked	IU
55	234	rtc check @ 232	Inference	Unchecked	IU
56	234	rtc check @ 232	Inference	Unchecked	IU
57	234	rtc check @ 232	Inference	Unchecked	IU
58	234	rtc check @ 232	Inference	Unchecked	IU
59	234	rtc check @ 232	Inference	Unchecked	IU
60	234	rtc check @ 232	Inference	Unchecked	IU
61	start	assert @ 234	Examiner	Live	EL
62	234	assert @ 234	Examiner	Live	EL
63	234	assert @ 234	Examiner	Live	EL
64	234	assert @ 234	Examiner	Live	EL
65	234	assert @ 234	Examiner	Live	EL
66	234	assert @ 234	Examiner	Live	EL
67	234	assert @ 234	Examiner	Live	EL
68	234	assert @ 234	Examiner	Live	EL
69	234	assert @ 234	Examiner	Live	EL
70	234	assert @ 234	Examiner	Live	EL
71	234	assert @ 234	Examiner	Live	EL
72	234	assert @ 234	Examiner	Live	EL
73	234	assert @ 234	Examiner	Live	EL
74	234	assert @ 234	Examiner	Live	EL
75	234	assert @ 234	Examiner	Live	EL
76	234	assert @ 234	Examiner	Live	EL
77	234	assert @ 234	Examiner	Live	EL
78	234	assert @ 234	Examiner	Live	EL
79	234	assert @ 234	Examiner	Live	EL
80	234	assert @ 234	Examiner	Live	EL
81	234	assert @ 234	Examiner	Live	EL
82	234	assert @ 234	Examiner	Live	EL
83	234	assert @ 234	Examiner	Live	EL
84	234	assert @ 234	Examiner	Live	EL
85	234	assert @ 234	Examiner	Live	EL
86	234	assert @ 234	Examiner	Live	EL
87	234	assert @ 234	Examiner	Live	EL
88	234	assert @ 234	Examiner	Live	EL
89	234	assert @ 234	Examiner	Live	EL
90	234	assert @ 234	Examiner	Live	EL
91	234	rtc check @ 240	Inference	Unchecked	IU
92	234	rtc check @ 240	Inference	Unchecked	IU
93	234	rtc check @ 241	Inference	Unchecked	IU
94	234	rtc check @ 241	Undischarged	Unchecked	UU
95	234	rtc check @ 242	Inference	Unchecked	IU
96	234	rtc check @ 243	Inference	Unchecked	IU
97	234	pre check @ 244	Inference	Unchecked	IU
98	234	rtc check @ 247	Inference	Unchecked	IU
99	234	rtc check @ 248	Inference	Unchecked	IU
100	234	rtc check @ 248	Inference	Unchecked	IU
101	234	rtc check @ 249	Inference	Unchecked	IU
102	234	rtc check @ 249	Undischarged	Unchecked	UU
103	234	rtc check @ 250	Inference	Unchecked	IU
104	234	rtc check @ 251	Inference	Unchecked	IU
105	234	pre check @ 252	Inference	Unchecked	IU
106	234	rtc check @ 256	Inference	Unchecked	IU
107	234	rtc check @ 257	Inference	Unchecked	IU
108	234	rtc check @ 257	Inference	Unchecked	IU
109	234	rtc check @ 258	Inference	Unchecked	IU
110	234	rtc check @ 258	Undischarged	Unchecked	UU
111	234	rtc check @ 259	Inference	Unchecked	IU
112	234	rtc check @ 260	Inference	Unchecked	IU
113	234	pre check @ 261	Inference	Unchecked	IU
114	234	rtc check @ 265	Inference	Unchecked	IU
115	234	rtc check @ 265	Inference	Unchecked	IU
116	234	rtc check @ 265	Inference	Unchecked	IU
117	234	rtc check @ 265	Inference	Unchecked	IU
118	234	rtc check @ 266	Inference	Unchecked	IU

119	234	rtc check @ 266	Inference	Unchecked	IU
120	234	rtc check @ 266	Undischarged	Unchecked	UU
121	234	rtc check @ 266	Undischarged	Unchecked	UU
122	234	rtc check @ 267	Inference	Unchecked	IU
123	234	rtc check @ 267	Inference	Unchecked	IU
124	234	rtc check @ 270	Inference	Unchecked	IU
125	234	rtc check @ 270	Inference	Unchecked	IU
126	234	rtc check @ 271	Undischarged	Unchecked	UU
127	234	rtc check @ 271	Undischarged	Unchecked	UU
128	234	rtc check @ 272	Inference	Unchecked	IU
129	234	rtc check @ 272	Inference	Unchecked	IU
130	234	rtc check @ 275	Inference	Unchecked	IU
131	234	rtc check @ 275	Inference	Unchecked	IU
132	234	rtc check @ 275	Inference	Unchecked	IU
133	234	rtc check @ 275	Inference	Unchecked	IU
134	234	pre check @ 276	Inference	Unchecked	IU
135	234	pre check @ 276	Inference	Unchecked	IU
136	234	pre check @ 276	Inference	Unchecked	IU
137	234	pre check @ 276	Inference	Unchecked	IU
138	234	rtc check @ 278	Undischarged	Unchecked	UU
139	234	rtc check @ 278	Undischarged	Unchecked	UU
140	234	rtc check @ 278	Undischarged	Unchecked	UU
141	234	rtc check @ 278	Undischarged	Unchecked	UU
142	234	rtc check @ 282	Inference	Unchecked	IU
143	234	rtc check @ 282	Inference	Unchecked	IU
144	234	rtc check @ 282	Inference	Unchecked	IU
145	234	rtc check @ 282	Inference	Unchecked	IU
146	234	rtc check @ 285	Inference	Unchecked	IU
147	234	rtc check @ 285	Inference	Unchecked	IU
148	234	rtc check @ 285	Inference	Unchecked	IU
149	234	rtc check @ 285	Inference	Unchecked	IU
150	234	rtc check @ 291	Inference	Unchecked	IU
151	234	rtc check @ 292	Inference	Unchecked	IU
152	234	rtc check @ 292	Inference	Unchecked	IU
153	234	rtc check @ 293	Inference	Unchecked	IU
154	234	rtc check @ 293	Undischarged	Unchecked	UU
155	234	rtc check @ 294	Inference	Unchecked	IU
156	234	rtc check @ 295	Inference	Unchecked	IU
157	234	pre check @ 296	Inference	Unchecked	IU
158	234	rtc check @ 300	Inference	Unchecked	IU
159	234	rtc check @ 300	Inference	Unchecked	IU
160	234	rtc check @ 301	Inference	Unchecked	IU
161	234	rtc check @ 301	Inference	Unchecked	IU
162	234	rtc check @ 302	Undischarged	Unchecked	UU
163	234	rtc check @ 302	Undischarged	Unchecked	UU
164	234	rtc check @ 302	Inference	Unchecked	IU
165	234	rtc check @ 302	Inference	Unchecked	IU
166	234	rtc check @ 303	Inference	Unchecked	IU
167	234	rtc check @ 303	Inference	Unchecked	IU
168	234	rtc check @ 303	Undischarged	Unchecked	UU
169	234	rtc check @ 303	Undischarged	Unchecked	UU
170	234	rtc check @ 304	Inference	Unchecked	IU
171	234	rtc check @ 304	Inference	Unchecked	IU
172	234	rtc check @ 307	Inference	Unchecked	IU
173	234	rtc check @ 307	Inference	Unchecked	IU
174	234	rtc check @ 308	Inference	Unchecked	IU
175	234	rtc check @ 308	Inference	Unchecked	IU
176	234	rtc check @ 311	Inference	Unchecked	IU
177	234	rtc check @ 311	Inference	Unchecked	IU
178	234	rtc check @ 311	Inference	Unchecked	IU
179	234	rtc check @ 311	Inference	Unchecked	IU
180	234	pre check @ 312	Inference	Unchecked	IU
181	234	pre check @ 312	Inference	Unchecked	IU
182	234	pre check @ 312	Inference	Unchecked	IU
183	234	pre check @ 312	Inference	Unchecked	IU

```
| 184 | 234   | rtc check @ 314     | Undischarged       | Unchecked |   UU   |
| 185 | 234   | rtc check @ 314     | Undischarged       | Unchecked |   UU   |
| 186 | 234   | rtc check @ 314     | Undischarged       | Unchecked |   UU   |
| 187 | 234   | rtc check @ 314     | Undischarged       | Unchecked |   UU   |
| 188 | 234   | rtc check @ 316     | Inference          | Unchecked |   IU   |
| 189 | 234   | rtc check @ 316     | Inference          | Unchecked |   IU   |
| 190 | 234   | rtc check @ 316     | Inference          | Unchecked |   IU   |
| 191 | 234   | rtc check @ 316     | Inference          | Unchecked |   IU   |
| 192 | 234   |    assert @ finish  | Examiner           | Dead      |   ED   |
| 193 | 234   |    assert @ finish  | Examiner           | Dead      |   ED   |
| 194 | 234   |    assert @ finish  | Examiner           | Dead      |   ED   |
| 195 | 234   |    assert @ finish  | Examiner           | Dead      |   ED   |
| 196 | 234   |    assert @ finish  | Examiner           | Dead      |   ED   |
| 197 | 234   |    assert @ finish  | Examiner           | Dead      |   ED   |
| 198 | 234   |    assert @ finish  | Examiner           | Dead      |   ED   |
| 199 | 234   |    assert @ finish  | Examiner           | Dead      |   ED   |
| 200 | 234   |    assert @ finish  | Examiner           | Dead      |   ED   |
| 201 | 234   |    assert @ finish  | Examiner           | Dead      |   ED   |
| 202 | 234   |    assert @ finish  | Examiner           | Dead      |   ED   |
| 203 | 234   |    assert @ finish  | Examiner           | Dead      |   ED   |
| 204 | 234   |    assert @ finish  | Examiner           | Dead      |   ED   |
| 205 | 234   |    assert @ finish  | Examiner           | Dead      |   ED   |
| 206 | 234   |    assert @ finish  | Examiner           | Dead      |   ED   |
| 207 | 234   |    assert @ finish  | Examiner           | Dead      |   ED   |
| 208 | 234   |    assert @ finish  | Examiner           | Dead      |   ED   |
| 209 | 234   |    assert @ finish  | Examiner           | Dead      |   ED   |
| 210 | 234   |    assert @ finish  | Examiner           | Dead      |   ED   |
| 211 | 234   |    assert @ finish  | Examiner           | Dead      |   ED   |
| 212 | 234   |    assert @ finish  | Examiner           | Dead      |   ED   |
| 213 | 234   |    assert @ finish  | Examiner           | Dead      |   ED   |
| 214 | 234   |    assert @ finish  | Examiner           | Dead      |   ED   |
| 215 | 234   |    assert @ finish  | Examiner           | Dead      |   ED   |
| 216 | 234   |    assert @ finish  | Examiner           | Dead      |   ED   |
| 217 | 234   |    assert @ finish  | Examiner           | Dead      |   ED   |
| 218 | 234   |    assert @ finish  | Examiner           | Dead      |   ED   |
| 219 | 234   |    assert @ finish  | Examiner           | Dead      |   ED   |
| 220 | 234   |    assert @ finish  | Examiner           | Dead      |   ED   |
 ---------------------------------------------------------------------------


File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/startpump.vcg
procedure Pca_Operation.StartPump

VCs generated 25-JUL-2014 14:16:24

VCs simplified 25-JUL-2014 14:16:46

File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/startpump.dpc
DPCs generated 25-JUL-2014 14:16:24

DPC ZombieScoped 25-JUL-2014  14:16:4

VCs for procedure_startpump :
 ---------------------------------------------------------------------------
| #   | From  | To                | Proved By          | Dead Path | Status |
|---------------------------------------------------------------------------
| 1   | start | rtc check @ 166     | Inference          | Unchecked |   IU   |
| 2   | start |    assert @ finish  | Examiner           | Live      |   EL   |
 ---------------------------------------------------------------------------


File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/stoppump.vcg
procedure Pca_Operation.StopPump

VCs generated 25-JUL-2014 14:16:24
```

```
VCs simplified 25-JUL-2014 14:16:46

File /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/stoppump.dpc
DPCs generated 25-JUL-2014 14:16:24

DPC ZombieScoped 25-JUL-2014  14:16:4

VCs for procedure_stoppump :
 ------------------------------------------------------------------------------
| #   | From  | To                  | Proved By          | Dead Path | Status |
|------------------------------------------------------------------------------
| 1   | start | rtc check @ 173     | Inference          | Unchecked |   IU   |
| 2   | start |    assert @ finish  | Examiner           | Live      |   EL   |
 ------------------------------------------------------------------------------


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

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

   2  /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/get_time_between_activations.
     vcg
   1  /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/integer_array_store/get.vcg
   1  /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/integer_array_store/inc.vcg
   1  /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/integer_array_store/put.vcg
   2  /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/integer_array_store/sum.vcg
   1  /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/max_drug_per_hour_watcher.vcg
   1  /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/patientbolus.vcg
  20  /Users/jj/aadl-medical/pca-pump-beagleboard/pca_ravenscar/pca_operation/rate_controller.vcg

Proof strategies used by subprograms
------------------------------------------------------------------------------
Total subprograms with at least one VC proved by examiner:             15
Total subprograms with at least one VC proved by simplifier:           20
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:                  14
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:                                        14
Total subprograms with at least one undischarged VC:                    8  <<<
Total subprograms with at least one false VC:                           0
                                                                       -----
Total subprograms for which VCs have been generated:                   22


ZombieScope Summary:
------------------------------------------------------------------------------
Total subprograms for which DPCs have been generated:                  22
```

```
Total number subprograms with dead paths found:                          3
Total number of dead paths found:                                       32


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             93         80         12          1
Precondition            12          0         12          0
Check stmnt.             0          0          0          0
Runtime check          187          0        159         28
Refinem. VCs            10         10          0          0
Inherit. VCs             0          0          0          0
=============================================================
Totals:                302         90        183         29 <<<
%Totals:                           30%        61%        10%

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

Listing C.1: POGS report for PCA Pump prototype

Appendix D

Rate controller thread from PCA pump AADL models

This appendix presents `Rate_Controller` thread from `PCA_Operation` module, from AADL/BLESS models of PCA pump, created by Brian Larson.

```
1  thread Rate_Controller
2  features
3      Infusion_Flow_Rate: out data port PCA_Types::Flow_Rate
4        {BLESS::Assertion => "<<:=PUMP_RATE()>>";};
5      System_Status: out event data port PCA_Types::Status_Type;
6      Begin_Infusion: in event port
7        {BLESS::Assertion => "<<Rx_APPROVED()>>";};
8      Begin_Priming: in event port;
9      End_Priming: in event port;
10     Halt_Infusion: in event port;
11     Square_Bolus_Rate: in data port PCA_Types::Flow_Rate
12       {BLESS::Assertion => "<<:=SQUARE_BOLUS_RATE>>";};
13     Patient_Bolus_Rate: in data port PCA_Types::Flow_Rate
14       {BLESS::Assertion => "<<:=PATIENT_BOLUS_RATE>>";};
15     Basal_Rate: in data port PCA_Types::Flow_Rate
16       {BLESS::Assertion => "<<:=BASAL_RATE>>";};
17     VTBI: in data port PCA_Types::Drug_Volume
18       {BLESS::Assertion => "<<:=VTBI>>";};
19     HW_Detected_Failure: in event port;
20     Stop_Pump_Completely: in event port;
21     Pump_At_KVO_Rate: in event port;
22     Alarm : in event data port PCA_Types::Alarm_Type;
23     Warning : in event data port PCA_Types::Warning_Type;
24     Patient_Request_Not_Too_Soon: in event port
25       {BLESS::Assertion => "<<:=PATIENT_REQUEST_NOT_TOO_SOON(now)>>";};
26     Door_Open: in data port Base_Types::Boolean;
27     Pause_Infusion: in event port;
28     Resume_Infusion: in event port;
29     CP_Clinician_Request_Bolus: in event port;
```

```
30     CP_Bolus_Duration: in event data port ICE_Types::Minute;
31     Near_Max_Drug_Per_Hour: in event port  --near maximum drug infused in any hour
32       {BLESS::Assertion => "<<PATIENT_NEAR_MAX_DRUG_PER_HOUR()>>";};
33     Over_Max_Drug_Per_Hour: in event port  --over maximum drug infused in any hour
34       {BLESS::Assertion => "<<PATIENT_OVER_MAX_DRUG_PER_HOUR()>>";};
35     ICE_Stop_Pump: in event port;
36   properties
37     Thread_Properties::Dispatch_Protocol => Aperiodic;
38 end Rate_Controller;
39
40 thread implementation Rate_Controller.imp
41 annex BLESS
42 {**
43 assert
44 <<HALT : :(la=SafetyPumpStop) or (la=StopButton) or (la=EndPriming)>>  --pump at 0 if stop button, or safety
       architecture says, or done priming
45 <<KVO_RATE : :(la=KVOcommand) or (la=KVOalarm) or (la=TooMuchJuice)>>  --pump at KVO rate when commanded,
       some alarms, or excedded hourly limit
46 <<PB_RATE : :la=PatientButton>>  --patient button pressed, and allowed
47 <<CCB_RATE : :(la=StartSquareBolus) or (la=ResumeSquareBolus)>>  --clinician-commanded bolus start or
       resumption after patient bolus
48 <<PRIME_RATE : :la=StartPriming>>  --priming pump
49 <<BASAL_RATE : :(la=StartButton) or (la=ResumeBasal) or (la=SquareBolusDone)>>  --regular infusion
50 <<PUMP_RATE : :=
51   (HALT()) -> 0,                                    --no flow
52   (KVO_RATE()) -> PCA_Properties::KVO_Rate,         --KVO rate
53   (PB_RATE()) -> Patient_Bolus_Rate,  --maximum infusion upon patient request
54   (CCB_RATE()) -> Square_Bolus_Rate,                --square bolus rate=VTBI/duration, from data port
55   (PRIME_RATE()) -> PCA_Properties::Prime_Rate,     --pump priming
56   (BASAL_RATE()) -> Basal_Rate                      --basal rate, from data port
57 >>
58 invariant <<true>>
59 variables
60   --time of last action
61   tla :BLESS_Types::Time := 0;
62   la :   --last action
63     enumeration (
64       SafetyStopPump,   --safety architecture found a problem
65       StopButton,       --clinician pressed stop button
66       KVOcommand,       --from control panel (clinician) or ICE (app) to pump Keep-vein-open rate
67       KVOalarm,         --some alarms should pump at KVO rate
68       TooMuchJuice,     --exceeded max drug per hour, pump at KVO until prescription and patient are re-
       authenticated
69       PatientButton,    --patient requested drug
70       ResumeSquareBolus,--infusion of VTBI finished, resume clinician-commanded bolus
71       ResumeBasal,      --infusion of VTBI finished, resume basal-rate
72       StartSquareBolus, --begin clinician-commanded bolus
73       SquareBolusDone,  --infusion of VTBI finished
74       StartPriming,     --begin pump/line priming, pressed "prime" button
75       EndPriming,       --end priming, pressed "prime" button again, or time-out
76       StartButton       --start pumping at basal rate
77     );
78   pb_duration :BLESS_Types::Time  --patient button duration = VTBI/Patient_Bolus_Rate
79     <<PB_DURATION : :pb_duration=(VTBI/Patient_Bolus_Rate)>>;
80 states
81   PowerOn : initial state;        --power-on
82   WaitForRx : complete state;   --wait for valid prescription
83   CheckPBR : state  --check Patient_Bolus_Rate is positive
84     <<Rx_APPROVED()>>;
85   RxApproved : complete state   --prescription verified
86     <<Rx_APPROVED() and PB_DURATION()>>;
87   Priming : complete state      --priming the pump, 1 ml in 6 sec
88     <<(la=StartPriming) and (Infusion_Flow_Rate@now = PCA_Properties::Prime_Rate) and PB_DURATION()>>;
89   WaitForStart : complete state   --wait for clinician to press 'start' button
90     <<HALT() and (Infusion_Flow_Rate@now=0) and PB_DURATION()>>;
```

```
91   PumpBasalRate : complete state  --pumping at basal rate
92     <<((la=StartButton) or (la=ResumeBasal)) and (Infusion_Flow_Rate@now=Basal_Rate@now) and PB_DURATION()>>;
93   PumpPatientButtonVTBI : complete state  --pumping patient-requested bolus
94     <<(la=PatientButton) and PB_DURATION()
95       and (Infusion_Flow_Rate@now=Patient_Bolus_Rate)>>;
96   PumpCCBRate : complete state    --pumping at clinician-commanded bolus rate
97     <<((la=StartSquareBolus) or (la=ResumeSquareBolus)) and (Infusion_Flow_Rate@now=Square_Bolus_Rate@now)
       and PB_DURATION()>>;
98   PumpKVORate : complete state    --pumping at keep-vein-open rate
99     <<((la=KVOcommand) or (la=KVOalarm) or (la=TooMuchJuice))  and PB_DURATION()
100      and (Infusion_Flow_Rate@now=PCA_Properties::KVO_Rate)>>;
101  PumpingSuspended : complete state  --clinician pressed 'stop' button
102    <<((la=StopButton) or (la=SafetyStopPump)) and (Infusion_Flow_Rate@now=0)>>;
103  Crash : final state;    --abnormal termination
104  Done : final state    --normal termination
105    <<Infusion_Flow_Rate@now=0>>;
106 transitions
107 --wait for valid prescription
108   go : PowerOn-[ true ]->WaitForRx{};
109 --prescription validated
110   rxo : WaitForRx-[ on dispatch Begin_Infusion ]-> CheckPBR{};
111   pbr0 : CheckPBR-[ Patient_Bolus_Rate<=0 ]->Crash{}; --bad Patient_Bolus_Rate
112   pbrok : CheckPBR-[ Patient_Bolus_Rate>0 ]->RxApproved
113     {<<Rx_APPROVED() and (Patient_Bolus_Rate>0)>>  --likely will change from logic variable to predicate
       Rx_APPROVED()
114       pb_duration := VTBI/Patient_Bolus_Rate  --calculate patient bolus duration
115       --note division without knowing divsor is non-zero; should generate additional proof obligations for
       assignment using division
116     <<Rx_APPROVED() and PB_DURATION()>>};
117 --clinician press 'prime' button
118   rxpri : RxApproved-[ on dispatch Begin_Priming ]-> Priming
119     {
120     la :=StartPriming
121         <<Begin_Priming@now and Rx_APPROVED() and (la = StartPriming) and PB_DURATION()>>
122     ;
123     Infusion_Flow_Rate!(PCA_Properties::Prime_Rate)  --infuse at prime rate
124         <<(la = StartPriming) and Rx_APPROVED() and PB_DURATION() and
125           (Infusion_Flow_Rate@now=PCA_Properties::Prime_Rate)>>
126     };
127 --priming done, wait for start
128   prd: Priming-[ on dispatch End_Priming or timeout (Begin_Priming) PCA_Properties::Prime_Time sec]->
       WaitForStart
129     {
130     la:=EndPriming
131       <<HALT() and PB_DURATION()>>  --and Begin_Priming timed out
132     ;
133     Infusion_Flow_Rate!(0)   --stop priming flow
134       <<HALT() and (Infusion_Flow_Rate@now=0) and PB_DURATION()>>
135     };
136 --prime again
137   pri: WaitForStart-[ on dispatch Begin_Priming ]-> Priming
138     {
139     la:=StartPriming
140         <<Begin_Priming@now and PB_DURATION() and PRIME_RATE()>>
141     ;
142     Infusion_Flow_Rate!(PCA_Properties::Prime_Rate)  --infuse at prime rate
143         <<PRIME_RATE() and PB_DURATION() and
144           (Infusion_Flow_Rate@now=PCA_Properties::Prime_Rate)>>
145    };
146 --clinician press 'start' button after priming
147   sap: WaitForStart-[ on dispatch Begin_Infusion ]-> PumpBasalRate  --start after priming
148     {
149     la:=StartButton
150       <<(la=StartButton) and Begin_Infusion@now  and PB_DURATION()>>
151     ;
```

```
152     Infusion_Flow_Rate!(Basal_Rate)   --infuse at basal rate
153       <<(la=StartButton) and (Infusion_Flow_Rate@now=Basal_Rate@now) and PB_DURATION()>>
154     };
155 --Patient_Request_Bolus during basal rate infusion
156   pump_basal_rate:
157   PumpBasalRate-[ on dispatch Patient_Request_Not_Too_Soon]-> PumpPatientButtonVTBI
158     {
159     la := PatientButton
160       <<(la=PatientButton) and Patient_Request_Bolus@now and PB_DURATION()>>
161     ;
162     Infusion_Flow_Rate!(Patient_Bolus_Rate)   --infuse at patient button rate
163       <<(la=PatientButton) and PB_DURATION()
164         and (Infusion_Flow_Rate@now=Patient_Bolus_Rate)>>
165     };  --end of pump_basal_rate
166 --VTBI delivered
167   vtbi_delivered:
168   PumpPatientButtonVTBI -[ on dispatch timeout (Infusion_Flow_Rate) pb_duration ms ]-> PumpBasalRate
169     {
170     la:=ResumeBasal
171     ;
172     <<(la=ResumeBasal) and PB_DURATION()>>  --and timeout of patient button duration
173     Infusion_Flow_Rate!(Basal_Rate)   --infuse at basal rate
174       <<(la=ResumeBasal) and (Infusion_Flow_Rate@now=Basal_Rate@now) and PB_DURATION()>>
175     };  --end of vtbi_delivered
176 **};
177 end Rate_Controller.imp;
```

Listing D.1: `Rate_Controller` thread

Appendix E

Simplified PCA pump AADL models

This appendix contains simplified AADL/BLESS models. They were created based on AADL/BLESS models of PCA pump, created by Brian Larson.

```
property set BLESS_Properties is
  with AADL_Project;

  Supported_Operators : list of aadlstring applies to ( data );
  Supported_Relations : list of aadlstring applies to ( data );
  Radix : AADL_Project::Size_Units applies to ( data );
end BLESS_Properties;
```

Listing E.1: `BLESS_Properties` property set

```
property set BLESS is
  Assertion : aadlstring applies to ( all );
  Typed : aadlstring applies to ( all );
  Invariant : aadlstring applies to ( all );
end BLESS;
```

Listing E.2: `BLESS` property set

```
property set PCA_Properties is
  with PCA_Types;

  Drug_Library_Size : constant aadlinteger => 500;
  Fault_Log_Size : constant aadlinteger => 150;
  Event_Log_Size : constant aadlinteger => 1500;
  KVO_Rate_Constant : constant aadlinteger => 1;
  KVO_Rate : constant aadlinteger => PCA_Properties::KVO_Rate_Constant;
  Max_Rate : constant aadlinteger => 10;
end PCA_Properties;
```

Listing E.3: `PCA_Properties` property set

```
package BLESS_Types public
with Base_Types, BLESS_Properties, Data_Model, Memory_Properties, BLESS;

data Integer extends Base_Types::Integer
  properties  --operators and relation symbols defined for Integer
    BLESS::Typed => "integer";
    BLESS_Properties::Supported_Operators => ("+", "*", "-", "/", "mod", "rem", "**");
    BLESS_Properties::Supported_Relations => ("=", "!=", "<", "<=", ">=", ">");
end Integer;

data Natural extends Base_Types::Natural
  properties  --operators and relation symbols defined for Natural
    BLESS::Typed => "natural";
    BLESS_Properties::Supported_Operators => ("+", "*", "-", "/", "mod", "rem", "**");
    BLESS_Properties::Supported_Relations => ("=", "!=", "<", "<=", ">=", ">");
end Natural;

data Real extends Base_Types::Float
  properties  --operators and relation symbols defined for Float
    BLESS::Typed => "real";
    BLESS_Properties::Supported_Operators => ("+", "*", "-", "/", "**");
    BLESS_Properties::Supported_Relations => ("=", "!=", "<", "<=", ">=", ">");
end Real;

data String extends Base_Types::String
  properties  --operators and relation symbols defined for String
    BLESS::Typed => "string";
    BLESS_Properties::Supported_Operators => ("+", "-"); --just concatenation
    BLESS_Properties::Supported_Relations => ("=", "!=", "<", "<=", ">=", ">");
end String;

data Fixed_Point
  properties  --operators and relation symbols defined for fixed-point arithmetic
    BLESS::Typed => "fixed";
    BLESS_Properties::Supported_Operators => ("+", "*", "-", "/", "**");
    BLESS_Properties::Supported_Relations => ("=", "!=", "<", "<=", ">=", ">");
    Data_Model::Data_Representation => Integer;
end Fixed_Point;

data Time extends Base_Types::Integer_64  --in milliseconds
  properties  --operators and relation symbols defined for Time
    --don't have a way to say that Time may be multiplied or divided by scalar
    --but not another Time
    BLESS::Typed => "integer";
    BLESS_Properties::Supported_Operators => ("+", "*", "-", "/");
    BLESS_Properties::Supported_Relations => ("=", "!=", "<", "<=", ">=", ">");
end Time;

end BLESS_Types;
```

Listing E.4: `BLESS_Types` package

```
package ICE_Types
public
with Data_Model;
with Base_Types;
  data Milliliter
  properties
    Data_Model::Data_Representation => Integer;
    Data_Model::Base_Type => (classifier (Base_Types::Unsigned_16)); --two bytes for 0-1000 ml
    Data_Model::Integer_Range => 0 .. 1000;
    Data_Model::Measurement_Unit => "ml";
  end Milliliter;

  data Milliliter_Per_Hour
  properties
    Data_Model::Data_Representation => Integer;
    Data_Model::Base_Type => (classifier (Base_Types::Unsigned_16)); --two bytes for 0-1000 ml/hr
    Data_Model::Integer_Range => 0 .. 1000;
    Data_Model::Measurement_Unit => "ml_per_hr";
  end Milliliter_Per_Hour;

  data Microliter_Per_Hour
  properties
    Data_Model::Data_Representation => Integer;
    Data_Model::Base_Type => (classifier (Base_Types::Unsigned_16)); --two bytes for 0-1000 ul/hr
    Data_Model::Integer_Range => 0 .. 1000;
    Data_Model::Measurement_Unit => "ul_per_hr";
  end Microliter_Per_Hour;

  data Minute
  properties
    Data_Model::Data_Representation => Integer;
    Data_Model::Base_Type => (classifier (Base_Types::Unsigned_16)); --two bytes for 0-1000 minutes
    Data_Model::Integer_Range => 0 .. 1000;
    Data_Model::Measurement_Unit => "min";
  end Minute;

  data Alarm_Signal --according to IEC 60601-1-8/FDIS AAA.201.8 ALARM SIGNAL inactivation states
  properties
    Data_Model::Data_Representation => Enum;
    Data_Model::Enumerators => ("On", "Alarm_Off", "Alarm_Paused", "Audio_Off", "Audio_Paused");
  end Alarm_Signal;

  data Percent
  properties
    Data_Model::Data_Representation => Integer;
    Data_Model::Base_Type => (classifier (Base_Types::Unsigned_8)); --one byte for 0-100 percent
    Data_Model::Integer_Range => 0 .. 100;
  end Percent;

  data Minute_Count extends Base_Types::Integer
  end Minute_Count;

  data Second_Count extends Base_Types::Integer
  end Second_Count;

end ICE_Types;
```

Listing E.5: `ICE_Types` package

```
1  package PCA_Types
2  public
3    with Base_Types, Data_Model, PCA_Properties, ICE_Types, BLESS_Types, BLESS;
4
5  data Alarm_Type
6    properties
7      BLESS::Typed=>"enumeration (
8            No_Alarm,
9            Pump_Overheated,
10           Defective_Battery,
11           Low_Battery,
12               POST_Failure,
13           RAM_Failure,
14           ROM_failure,
15           CPU_Failure,
16           Thread_Monitor_Failure,
17           Air_In_Line,
18           Upstream_Occlusion,
19           Downstream_Occlusion,
20           Empty_Reservoir,
21           Basal_Overinfusion,
22           Bolus_Overinfusion,
23           Square_Bolus_Overinfusion)";
24     Data_Model::Data_Representation => Enum;
25     Data_Model::Enumerators => (
26       "No_Alarm",
27       "Pump_Overheated",
28       "Defective_Battery",
29       "Low_Battery",
30         "POST_Failure",
31         "RAM_Failure",
32         "ROM_failure",
33         "CPU_Failure",
34         "Thread_Monitor_Failure",
35         "Air_In_Line",
36         "Upstream_Occlusion",
37         "Downstream_Occlusion",
38         "Empty_Reservoir",
39       "Basal_Overinfusion",
40       "Bolus_Overinfusion",
41       "Square_Bolus_Overinfusion");
42 end Alarm_Type;
43
44 data Warning_Type
45   properties
46     BLESS::Typed=>
47       "enumeration (No_Warning,
48           Over_Max_Drug_Per_Hour,
49           Soft_Limit,
50           Low_Reservoir,
51           Priming_Failure,
52           Basal_Underinfusion,
53           Bolus_Underinfusion,
54           Square_Bolus_Underinfusion,
55           Input_Needed,
56           Long_Pause,
57           Drug_Not_In_Library,
58           Hard_Limit_Violated,
59           Voltage_OOR)";
60     Data_Model::Data_Representation => Enum;
61     Data_Model::Enumerators => (
62       "No_Warning",
63       "Over_Max_Drug_Per_Hour",
64         "Soft_Limit",
65         "Low_Reservoir",
```

```
66          "Priming_Failure",
67        "Basal_Underinfusion",
68        "Bolus_Underinfusion",
69        "Square_Bolus_Underinfusion",
70        "Input_Needed",
71        "Long_Pause",
72        "Drug_Not_In_Library",
73        "Hard_Limit_Violated",
74        "Voltage_OOR");
75 end Warning_Type;
76
77 data Status_Type
78   properties
79     BLESS::Typed=>"enumeration (Stopped,Bolus,Basal,KVO,Square_Bolus)";
80     Data_Model::Data_Representation => Enum;
81     Data_Model::Enumerators => ("Stopped","Bolus","Basal","KVO","Square_Bolus");
82 end Status_Type;
83
84 data Flow_Rate  --dose rate
85   properties
86     BLESS::Typed=>"integer";
87     Data_Model::Base_Type => (classifier(Base_Types::Integer_16));
88     Data_Model::Measurement_Unit => "ml/hr";
89 end Flow_Rate;
90
91 data Drug_Volume  --volume of VTBI
92   properties
93     BLESS::Typed=>"integer";
94     Data_Model::Base_Type => (classifier(Base_Types::Integer_16));
95     Data_Model::Measurement_Unit => "ml";
96 end Drug_Volume;
97
98 data Drug_Weight  --string representing what drug, conectration, and volume is in the reservoir
99   properties
100    BLESS::Typed=>"integer";
101    Data_Model::Base_Type => (classifier(Base_Types::Integer_16));
102    Data_Model::Measurement_Unit => "mg";
103 end Drug_Weight;
104
105 data Drug_Concentration  --string representing what drug, conectration, and volume is in the reservoir
106   properties
107    BLESS::Typed=>"integer";
108    Data_Model::Base_Type => (classifier(Base_Types::Integer));
109    Data_Model::Measurement_Unit => "mg/l";
110 end Drug_Concentration;
111
112 data Drug_Record  --holds pharmacy data for a drug that may be used with the pump
113   properties
114    BLESS::Typed =>
115    "record (
116      Amount : PCA_Types::Drug_Weight;               --The weight of the drug dissolved in the diluent (mg)
117      Concentration : PCA_Types::Drug_Concentration; --Drug concentration; as prescribed
118      Vtbi_Lower_Soft : PCA_Types::Drug_Volume;      --Lower soft limit of drug volume to be infused
119      Vtbi_Lower_Hard : PCA_Types::Drug_Volume;      --Lower hard limit of drug volume to be infused
120      Vtbi_Typical : PCA_Types::Drug_Volume;         --Typical drug volume to be infused
121      Vtbi_Upper_Soft : PCA_Types::Drug_Volume;      --Upper soft limit of drug volume to be infused
122      Vtbi_Upper_Hard : PCA_Types::Drug_Volume;      --Upper hard limit of drug volume to be infused
123      Basal_Rate_Lower_Soft : PCA_Types::Flow_Rate;  --Lower soft limit of basal drug dose rate
124      Basal_Rate_Lower_Hard : PCA_Types::Flow_Rate;  --Lower hard limit of basal drug dose rate
125      Basal_Rate_Typical : PCA_Types::Flow_Rate;     --Typical basal drug dose rate
126      Basal_Rate_Upper_Soft : PCA_Types::Flow_Rate;  --Upper soft limit of basal drug dose rate
127      Basal_Rate_Upper_Hard : PCA_Types::Flow_Rate;  --Upper hard limit of basal drug dose rate
128      Bolus_Typical : PCA_Types::Drug_Volume;        --Typical Value of Bolus Volume
129      Square_Bolus_rate_typical : PCA_Types::Flow_Rate;        --Typical duration of clinician commanded
     bolus
```

```
130       )";
131     Data_Model::Data_Representation => Struct;
132     Data_Model::Element_Names =>
133       ( "Amount",                 --The weight of the drug dissolved in the diluent (mg)
134       "Concentration",          --Drug concentration; as prescribed
135       "Vtbi_Lower_Soft",        --Lower soft limit of drug volume to be infused
136       "Vtbi_Lower_Hard",        --Lower hard limit of drug volume to be infused
137       "Vtbi_Typical",           --Typical drug volume to be infused
138       "Vtbi_Upper_Soft",        --Upper soft limit of drug volume to be infused
139       "Vtbi_Upper_Hard",        --Upper hard limit of drug volume to be infused
140       "Basal_Rate_Lower_Soft",  --Lower soft limit of basal drug dose rate
141       "Basal_Rate_Lower_Hard",  --Lower hard limit of basal drug dose rate
142       "Basal_Rate_Typical",     --Typical basal drug dose rate
143       "Basal_Rate_Upper_Soft",  --Upper soft limit of basal drug dose rate
144       "Basal_Rate_Upper_Hard",  --Upper hard limit of basal drug dose rate
145       "Bolus_Typical",          --Typical Value of Bolus Volume
146       "Square_Bolus_Rate_Typical"      --Typical rate of clinician commanded bolus
147       );
148     Data_Model::Base_Type =>
149       ( classifier(Drug_Weight),         --amount
150         classifier(Drug_Concentration),  --concentration
151         classifier(Drug_Volume),     --vtbi_lower_soft
152         classifier(Drug_Volume),     --vtbi_lower_hard
153         classifier(Drug_Volume),     --vtbi_typical
154         classifier(Drug_Volume),     --vtbi_upper_soft
155         classifier(Drug_Volume),     --vtbi_upper_hard
156         classifier(Flow_Rate),           --basal_rate_lower_soft
157         classifier(Flow_Rate),           --basal_rate_lower_hard
158         classifier(Flow_Rate),           --basal_rate_typical
159         classifier(Flow_Rate),           --basal_rate_upper_soft
160         classifier(Flow_Rate),           --basal_rate_upper_hard
161         classifier(Drug_Volume),     --bolus_typical
162         classifier(Flow_Rate)    --ssquare_bolus_rate_typical
163       );
164 end Drug_Record;
165
166
167 data Drug_Library  --holds drug records for all drugs approved by the hospital pharmacy
168   properties
169     BLESS::Typed => "array [PCA_Properties::Drug_Library_Size] of PCA_Types::Drug_Record";
170     Data_Model::Data_Representation => Array;
171     Data_Model::Base_Type => (classifier(Drug_Record));
172     Data_Model::Dimension => (PCA_Properties::Drug_Library_Size);
173 end Drug_Library;
174
175 data Prescription
176   properties
177     BLESS::Typed =>
178     "record (
179     Concentration : Drug_Concentration;
180     Initial_Volume : Drug_Volume;
181     Basal_Flow_Rate : Flow_Rate;
182     Vtbi : Drug_Volume;
183     Max_Drug_Per_Hour : Drug_Volume;
184     Minimum_Time_Between_Bolus : ICE_Types::Minute;
185     )";
186     Data_Model::Data_Representation => Struct;
187     Data_Model::Element_Names =>
188       ( "Concentration",
189     "Initial_Volume",
190     "Basal_Flow_Rate",
191     "Vtbi",
192     "Max_Drug_Per_Hour",
193     "Minimum_Time_Between_Bolus"
194       );
```

```
195     Data_Model::Base_Type =>
196       ( classifier(Drug_Concentration),   --concentration
197         classifier(Drug_Volume),          --initial volume
198         classifier(Flow_Rate),            --basal flow rate
199         classifier(Drug_Volume),          --VTBI
200         classifier(Drug_Volume),          --maximum drug allowed per hour
201         classifier(ICE_Types::Minute)    --min time between bolus doses
202       );
203 end Prescription;
204
205 data Fault_Record  --record of fault for log
206   properties
207     BLESS::Typed => "record (Alarm:Alarm_Type; Warning:Warning_Type; Occurrence_Time:BLESS_Types::Time;)";
208     Data_Model::Data_Representation => Struct;
209     Data_Model::Element_Names => ("Alarm","Warning","Occurrence_Time");
210     Data_Model::Base_Type => ( classifier(Alarm_Type),classifier(Warning_Type),classifier(BLESS_Types::Time))
       ;
211 end Fault_Record;
212
213 data Fault_Log  --holds records of faults
214   properties
215     BLESS::Typed => "array [PCA_Properties::Fault_Log_Size] of PCA_Types::Fault_Record";
216     Data_Model::Data_Representation => Array;
217     Data_Model::Base_Type => (classifier(Fault_Record));
218     Data_Model::Dimension => (PCA_Properties::Fault_Log_Size);
219 end Fault_Log;
220
221 data Event_Record  --record of event for log
222   properties
223     BLESS::Typed => "record ( Time : BLESS_Types::Time)";
224     Data_Model::Data_Representation => Struct;
225     Data_Model::Element_Names => ( "Time" );
226     Data_Model::Base_Type => (classifier(BLESS_Types::Time));
227 end Event_Record;
228
229 data Event_Log  --holds records of events
230   properties
231     BLESS::Typed => "array [PCA_Properties::Event_Log_Size] of PCA_Types::Event_Record";
232     Data_Model::Data_Representation => Array;
233     Data_Model::Base_Type => (classifier(Event_Record));
234     Data_Model::Dimension => (PCA_Properties::Event_Log_Size);
235 end Event_Log;
236
237 data Infusion_Type   --used for over- and under-infusion alarms
238   properties
239     BLESS::Typed=>"enumeration(Bolus_Infusion, Square_Infusion, Basal_Infusion, KVO_Infusion)";
240     Data_Model::Data_Representation => Enum;
241     Data_Model::Enumerators => ("Bolus_Infusion","Square_Infusion","Basal_Infusion","KVO_Infusion");
242 end Infusion_Type;
243
244 data Pump_Fault_Type
245   properties
246     BLESS::Typed=>"enumeration(Prime_Failure, Pump_Hot, Bubble, Upstream_Occlusion_Fault,
       Downstream_Occlusion_Fault, Overinfusion, Underinfusion)";
247     Data_Model::Data_Representation => Enum;
248     Data_Model::Enumerators => ("Prime_Failure","Pump_Hot","Bubble","Upstream_Occlusion_Fault","
       Downstream_Occlusion_Fault","Overinfusion","Underinfusion" );
249 end Pump_Fault_Type;
250
251 end PCA_Types;
```

Listing E.6: `PCA_Types` package

```
1  package PCA_Operation
2    public
3    with PCA_Properties, Base_Types, BLESS, BLESS_Types, ICE_Types, PCA_Types;
4
5  system operation
6    features
7      Start_Button_Pressed: in event port;
8      Stop_Button_Pressed: in event port;
9      Patient_Request_Bolus: in event port;
10     Clinician_Request_Bolus: in event port;
11     Bolus_Duration: in event data port ICE_Types::Minute;
12     Infusion_Flow_Rate: out data port PCA_Types::Flow_Rate;
13     System_Status: out data port PCA_Types::Status_Type;
14     Rx: in event data port PCA_Types::Prescription;
15 end operation;
16
17 system implementation operation.imp
18   subcomponents
19     operation_process: process operation_process.imp;
20   connections
21     start: port Start_Button_Pressed -> operation_process.Start_Button_Pressed;
22     stop: port Stop_Button_Pressed -> operation_process.Stop_Button_Pressed;
23     pbp: port Patient_Request_Bolus -> operation_process.Patient_Request_Bolus;
24     crb: port Clinician_Request_Bolus -> operation_process.Clinician_Request_Bolus;
25     bd: port Bolus_Duration -> operation_process.Bolus_Duration;
26     pfr: port operation_process.Infusion_Flow_Rate -> Infusion_Flow_Rate;
27     stat: port operation_process.System_Status -> System_Status;
28     rxo: port Rx->operation_process.Rx;
29 end operation.imp;
30
31 process operation_process
32   features
33     Start_Button_Pressed: in event port;
34     Stop_Button_Pressed: in event port;
35     Patient_Request_Bolus: in event port;
36     Clinician_Request_Bolus: in event port;
37     Bolus_Duration: in event data port ICE_Types::Minute;
38     Infusion_Flow_Rate: out data port PCA_Types::Flow_Rate;
39     System_Status: out data port PCA_Types::Status_Type;
40     Rx: in event data port PCA_Types::Prescription;
41 end operation_process;
42
43 process implementation operation_process.imp
44   subcomponents
45     Max_Drug_Per_Hour_Watcher : thread Max_Drug_Per_Hour_Watcher.imp;
46     Rate_Controller : thread Rate_Controller.imp;
47     Patient_Bolus_Checker : thread Patient_Bolus_Checker.imp;
48   connections
49     start: port Start_Button_Pressed -> Rate_Controller.Start_Button_Pressed;
50     stop: port Stop_Button_Pressed -> Rate_Controller.Stop_Button_Pressed;
51     pb: port Patient_Request_Bolus -> Patient_Bolus_Checker.Patient_Request_Bolus;
52     crb: port Clinician_Request_Bolus -> Rate_Controller.Clinician_Request_Bolus;
53     bd: port Bolus_Duration -> Rate_Controller.Bolus_Duration;
54     pfr: port Rate_Controller.Infusion_Flow_Rate -> Infusion_Flow_Rate;
55     ss: port Rate_Controller.System_Status -> System_Status;
56     rxrc: port Rx->Rate_Controller.Rx;
57 end operation_process.imp;
58
59 thread Max_Drug_Per_Hour_Watcher
60 features
61     Infusion_Flow_Rate: in data port PCA_Types::Flow_Rate
62       {BLESS::Assertion => "<<:=PUMP_RATE()>>";};
63     Max_Drug_Per_Hour: in data port PCA_Types::Drug_Volume
64       {BLESS::Assertion => "<<:=MAX_DRUG_PER_HOUR>>";};
65 end Max_Drug_Per_Hour_Watcher;
```

```
66
67 thread implementation Max_Drug_Per_Hour_Watcher.imp
68 end Max_Drug_Per_Hour_Watcher.imp;
69
70 thread Rate_Controller
71 features
72   Start_Button_Pressed: in event port;
73   Stop_Button_Pressed: in event port;
74   Rx: in event data port PCA_Types::Prescription
75     {BLESS::Assertion => "<<:=Rx_APPROVED()>>";};
76     Clinician_Request_Bolus: in event port;
77     Bolus_Duration: in event data port ICE_Types::Minute;
78     Infusion_Flow_Rate: out data port PCA_Types::Flow_Rate
79       {BLESS::Assertion => "<<:=PUMP_RATE()>>";};
80     System_Status: out event data port PCA_Types::Status_Type;
81 end Rate_Controller;
82
83 thread implementation Rate_Controller.imp
84 annex BLESS
85 {**
86 assert
87 <<HALT : :(la=StopButton) >>                              --pump at 0 if stop button
88 <<KVO_RATE : :(la=TooMuchJuice)>>                          --pump at KVO rate when commanded, some alarms, or
       excedded hourly limit
89 <<PB_RATE : :la=PatientButton>>                           --patient button pressed, and allowed
90 <<CCB_RATE : :(la=StartSquareBolus) or (la=ResumeSquareBolus)>>        --clinician-commanded bolus start or
       resumption after patient bolus
91 <<BASAL_RATE : :(la=StartButton) or (la=ResumeBasal) or (la=SquareBolusDone)>>  --regular infusion
92 <<PUMP_RATE : :=
93   (HALT()) -> 0,                                        --no flow
94   (KVO_RATE()) -> PCA_Properties::KVO_Rate,             --KVO rate
95   (PB_RATE()) -> Patient_Bolus_Rate,            --maximum infusion upon patient request
96   (CCB_RATE()) -> Square_Bolus_Rate,                    --square bolus rate=VTBI/duration, from data port
97   (BASAL_RATE()) -> Basal_Rate                          --basal rate, from data port
98 >>
99 invariant <<true>>
100 variables
101   la :   --last action
102     enumeration (
103       StopButton,       --clinician pressed stop button
104       TooMuchJuice,         --exceeded max drug per hour, pump at KVO until prescription and patient are re-
       authenticated
105       PatientButton,      --patient requested drug
106       ResumeSquareBolus,    --infusion of VTBI finished, resume clinician-commanded bolus
107       ResumeBasal,        --infusion of VTBI finished, resume basal-rate
108       StartSquareBolus,     --begin clinician-commanded bolus
109       SquareBolusDone,      --infusion of VTBI finished
110       StartButton       --start pumping at basal rate
111     );
112 **};
113 end Rate_Controller.imp;
114
115 thread Patient_Bolus_Checker
116   features
117     Patient_Request_Bolus: in event port;
118 end Patient_Bolus_Checker;
119
120 thread implementation Patient_Bolus_Checker.imp
121 end Patient_Bolus_Checker.imp;
122
123 end PCA_Operation;
```

Listing E.7: `PCA_Operation` package

Appendix F

Simplified PCA pump - translated from simplified AADL models

This appendinx presents PCA pump prototype, which was created by direct translation from simplified AADL/BLESS models presented in appendix E.

```
1  package Base_Types
2  is
3      protected type Boolean_Store
4      is
5          pragma Priority (10);
6
7          function Get return Boolean;
8          --# global in Boolean_Store;
9
10         procedure Put(X : in Boolean);
11         --# global out Boolean_Store;
12         --# derives Boolean_Store from X;
13     private
14         TheStoredData : Boolean := False;
15     end Boolean_Store;
16
17     protected type Integer_Store
18     is
19         pragma Priority (10);
20
21         function Get return Integer;
22         --# global in Integer_Store;
23
24         procedure Put(X : in Integer);
25         --# global out Integer_Store;
26         --# derives Integer_Store from X;
27     private
28         TheStoredData : Integer := 0;
29     end Integer_Store;
```

```
30
31      protected type Natural_Store
32      is
33          pragma Priority (10);
34
35          function Get return Natural;
36          --# global in Natural_Store;
37
38          procedure Put(X : in Natural);
39          --# global out Natural_Store;
40          --# derives Natural_Store from X;
41      private
42          TheStoredData : Natural := 0;
43      end Natural_Store;
44
45      type Integer_8 is new Integer range -2**(1*8-1) .. 2**(1*8-1)-1;
46
47      protected type Integer_8_Store
48      is
49          pragma Priority (10);
50
51          function Get return Integer_8;
52          --# global in Integer_8_Store;
53
54          procedure Put(X : in Integer_8);
55          --# global out Integer_8_Store;
56          --# derives Integer_8_Store from X;
57      private
58          TheStoredData : Integer_8 := 0;
59      end Integer_8_Store;
60
61      type Integer_16 is new Integer range -2**(2*8-1) .. 2**(2*8-1)-1;
62
63      protected type Integer_16_Store
64      is
65          pragma Priority (10);
66
67          function Get return Integer_16;
68          --# global in Integer_16_Store;
69
70          procedure Put(X : in Integer_16);
71          --# global out Integer_16_Store;
72          --# derives Integer_16_Store from X;
73      private
74          TheStoredData : Integer_16 := 0;
75      end Integer_16_Store;
76
77      type Integer_32 is new Integer range -2**(4*8-1) .. 2**(4*8-1)-1;
78
79      protected type Integer_32_Store
80      is
81          pragma Priority (10);
82
83          function Get return Integer_32;
84          --# global in Integer_32_Store;
85
86          procedure Put(X : in Integer_32);
87          --# global out Integer_32_Store;
88          --# derives Integer_32_Store from X;
89      private
90          TheStoredData : Integer_32 := 0;
91      end Integer_32_Store;
92
93      type Integer_64 is range -2**(8*8-1) .. 2**(8*8-1)-1; -- with new Integer gnat compiler error: value not
         in range of type "Standard.Integer"
```

```
94
95      protected type Integer_64_Store
96      is
97          pragma Priority (10);
98
99          function Get return Integer_64;
100         --# global in Integer_64_Store;
101
102         procedure Put(X : in Integer_64);
103         --# global out Integer_64_Store;
104         --# derives Integer_64_Store from X;
105     private
106         TheStoredData : Integer_64 := 0;
107     end Integer_64_Store;
108
109     type Unsigned_8 is new Integer range 0 .. 2**(1*8)-1;
110
111     protected type Unsigned_8_Store
112     is
113         pragma Priority (10);
114
115         function Get return Unsigned_8;
116         --# global in Unsigned_8_Store;
117
118         procedure Put(X : in Unsigned_8);
119         --# global out Unsigned_8_Store;
120         --# derives Unsigned_8_Store from X;
121     private
122         TheStoredData : Unsigned_8 := 0;
123     end Unsigned_8_Store;
124
125     type Unsigned_16 is new Integer range 0 .. 2**(2*8)-1;
126
127     protected type Unsigned_16_Store
128     is
129         pragma Priority (10);
130
131         function Get return Unsigned_16;
132         --# global in Unsigned_16_Store;
133
134         procedure Put(X : in Unsigned_16);
135         --# global out Unsigned_16_Store;
136         --# derives Unsigned_16_Store from X;
137     private
138         TheStoredData : Unsigned_16 := 0;
139     end Unsigned_16_Store;
140
141     type Unsigned_32 is range 0 .. 2**(4*8)-1; -- with new Integer gnat compiler error: value not in range of
          type "Standard.Integer"
142
143     protected type Unsigned_32_Store
144     is
145         pragma Priority (10);
146
147         function Get return Unsigned_32;
148         --# global in Unsigned_32_Store;
149
150         procedure Put(X : in Unsigned_32);
151         --# global out Unsigned_32_Store;
152         --# derives Unsigned_32_Store from X;
153     private
154         TheStoredData : Unsigned_32 := 0;
155     end Unsigned_32_Store;
156
```

```
157     --type Unsigned_64 is range 0 .. 2**64-1; -- gnat compiler error: integer type definition bounds out of
        range
158
159 end Base_Types;
160
161 package body Base_Types
162 is
163     protected body Boolean_Store is
164         function Get return Boolean
165         --# global in TheStoredData;
166         is
167         begin
168             return TheStoredData;
169         end Get;
170
171         procedure Put(X : in Boolean)
172           --# global out TheStoredData;
173           --# derives TheStoredData from X;
174         is
175         begin
176             TheStoredData := X;
177         end Put;
178     end Boolean_Store;
179
180     protected body Integer_Store is
181         function Get return Integer
182         --# global in TheStoredData;
183         is
184         begin
185             return TheStoredData;
186         end Get;
187
188         procedure Put(X : in Integer)
189           --# global out TheStoredData;
190           --# derives TheStoredData from X;
191         is
192         begin
193             TheStoredData := X;
194         end Put;
195     end Integer_Store;
196
197     protected body Natural_Store is
198         function Get return Natural
199         --# global in TheStoredData;
200         is
201         begin
202             return TheStoredData;
203         end Get;
204
205         procedure Put(X : in Natural)
206           --# global out TheStoredData;
207           --# derives TheStoredData from X;
208         is
209         begin
210             TheStoredData := X;
211         end Put;
212     end Natural_Store;
213
214     protected body Integer_8_Store is
215         function Get return Integer_8
216         --# global in TheStoredData;
217         is
218         begin
219             return TheStoredData;
220         end Get;
```

```
221
222         procedure Put(X : in Integer_8)
223           --# global out TheStoredData;
224           --# derives TheStoredData from X;
225         is
226         begin
227             TheStoredData := X;
228         end Put;
229     end Integer_8_Store;
230
231     protected body Integer_16_Store is
232         function Get return Integer_16
233         --# global in TheStoredData;
234         is
235         begin
236             return TheStoredData;
237         end Get;
238
239         procedure Put(X : in Integer_16)
240           --# global out TheStoredData;
241           --# derives TheStoredData from X;
242         is
243         begin
244             TheStoredData := X;
245         end Put;
246     end Integer_16_Store;
247
248     protected body Integer_32_Store is
249         function Get return Integer_32
250         --# global in TheStoredData;
251         is
252         begin
253             return TheStoredData;
254         end Get;
255
256         procedure Put(X : in Integer_32)
257           --# global out TheStoredData;
258           --# derives TheStoredData from X;
259         is
260         begin
261             TheStoredData := X;
262         end Put;
263     end Integer_32_Store;
264
265     protected body Integer_64_Store is
266         function Get return Integer_64
267         --# global in TheStoredData;
268         is
269         begin
270             return TheStoredData;
271         end Get;
272
273         procedure Put(X : in Integer_64)
274           --# global out TheStoredData;
275           --# derives TheStoredData from X;
276         is
277         begin
278             TheStoredData := X;
279         end Put;
280     end Integer_64_Store;
281
282     protected body Unsigned_8_Store is
283         function Get return Unsigned_8
284         --# global in TheStoredData;
285         is
```

```
286         begin
287            return TheStoredData;
288         end Get;
289
290         procedure Put(X : in Unsigned_8)
291           --# global out TheStoredData;
292           --# derives TheStoredData from X;
293         is
294         begin
295            TheStoredData := X;
296         end Put;
297      end Unsigned_8_Store;
298
299      protected body Unsigned_16_Store is
300         function Get return Unsigned_16
301         --# global in TheStoredData;
302         is
303         begin
304            return TheStoredData;
305         end Get;
306
307         procedure Put(X : in Unsigned_16)
308           --# global out TheStoredData;
309           --# derives TheStoredData from X;
310         is
311         begin
312            TheStoredData := X;
313         end Put;
314      end Unsigned_16_Store;
315
316      protected body Unsigned_32_Store is
317         function Get return Unsigned_32
318         --# global in TheStoredData;
319         is
320         begin
321            return TheStoredData;
322         end Get;
323
324         procedure Put(X : in Unsigned_32)
325           --# global out TheStoredData;
326           --# derives TheStoredData from X;
327         is
328         begin
329            TheStoredData := X;
330         end Put;
331      end Unsigned_32_Store;
332
333   end Base_Types;
```

Listing F.1: `Base_Types` package

```
1  with Base_Types;
2  --# inherit Base_Types;
3  package Bless_Types
4  is
5      subtype Fixed_Point is Integer;
6      protected type Fixed_Point_Store
7      is
8          pragma Priority (10);
9          function Get return Fixed_Point;
10         --# global in Fixed_Point_Store;
11         procedure Put(X : in Fixed_Point);
12         --# global out Fixed_Point_Store;
13         --# derives Fixed_Point_Store from X;
14     private
15         TheStoredData : Fixed_Point := 0;
16     end Fixed_Point_Store;
17
18     subtype Time is Base_Types.Integer_64;
19     protected type Time_Store
20     is
21         pragma Priority (10);
22         function Get return Time;
23         --# global in Time_Store;
24         procedure Put(X : in Time);
25         --# global out Time_Store;
26         --# derives Time_Store from X;
27     private
28         TheStoredData : Time := 0;
29     end Time_Store;
30 end Bless_Types;
31
32 package body Bless_Types
33 is
34     protected body Fixed_Point_Store is
35         function Get return Fixed_Point
36         --# global in TheStoredData;
37         is begin
38             return TheStoredData;
39         end Get;
40         procedure Put(X : in Fixed_Point)
41           --# global out TheStoredData;
42           --# derives TheStoredData from X;
43         is begin
44             TheStoredData := X;
45         end Put;
46     end Fixed_Point_Store;
47
48     protected body Time_Store is
49         function Get return Time
50         --# global in TheStoredData;
51         is begin
52             return TheStoredData;
53         end Get;
54         procedure Put(X : in Time)
55           --# global out TheStoredData;
56           --# derives TheStoredData from X;
57         is begin
58             TheStoredData := X;
59         end Put;
60     end Time_Store;
61 end Bless_Types;
```

Listing F.2: `Bless_Types` package

```
with Base_Types;
--# inherit Base_Types;
package Ice_Types
is
    subtype Milliliter is Base_Types.Unsigned_16 range 0 .. 1000;

    protected type Milliliter_Store
    is
        pragma Priority (10);

        function Get return Milliliter;
        --# global in Milliliter_Store;

        procedure Put(X : in Milliliter);
        --# global out Milliliter_Store;
        --# derives Milliliter_Store from X;
    private
        TheStoredData : Milliliter := 0;
    end Milliliter_Store;


    subtype Milliliter_Per_Hour is Base_Types.Unsigned_16 range 0 .. 1000;

    protected type Milliliter_Per_Hour_Store
    is
        pragma Priority (10);

        function Get return Milliliter_Per_Hour;
        --# global in Milliliter_Per_Hour_Store;

        procedure Put(X : in Milliliter_Per_Hour);
        --# global out Milliliter_Per_Hour_Store;
        --# derives Milliliter_Per_Hour_Store from X;
    private
        TheStoredData : Milliliter_Per_Hour := 0;
    end Milliliter_Per_Hour_Store;


    subtype Microliter_Per_Hour is Base_Types.Unsigned_16 range 0 .. 1000;

    protected type Microliter_Per_Hour_Store
    is
        pragma Priority (10);

        function Get return Microliter_Per_Hour;
        --# global in Microliter_Per_Hour_Store;

        procedure Put(X : in Microliter_Per_Hour);
        --# global out Microliter_Per_Hour_Store;
        --# derives Microliter_Per_Hour_Store from X;
    private
        TheStoredData : Microliter_Per_Hour := 0;
    end Microliter_Per_Hour_Store;


    subtype Minute is Base_Types.Unsigned_16 range 0 .. 1000;

    protected type Minute_Store
    is
        pragma Priority (10);

        function Get return Minute;
        --# global in Minute_Store;

        procedure Put(X : in Minute);
```

```
66          --# global out Minute_Store;
67          --# derives Minute_Store from X;
68      private
69          TheStoredData : Minute := 0;
70      end Minute_Store;
71
72
73      type Alarm_Signal is (On, Alarm_Off, Alarm_Paused, Audio_Off, Audio_Paused);
74
75      protected type Alarm_Signal_Store
76      is
77          pragma Priority (10);
78
79          function Get return Alarm_Signal;
80          --# global in Alarm_Signal_Store;
81
82          procedure Put(X : in Alarm_Signal);
83          --# global out Alarm_Signal_Store;
84          --# derives Alarm_Signal_Store from X;
85      private
86          TheStoredData : Alarm_Signal := Alarm_Signal'First;
87      end Alarm_Signal_Store;
88
89
90      subtype Percent is Base_Types.Unsigned_8 range 0 .. 100;
91
92      protected type Percent_Store
93      is
94          pragma Priority (10);
95
96          function Get return Percent;
97          --# global in Percent_Store;
98
99          procedure Put(X : in Percent);
100         --# global out Percent_Store;
101         --# derives Percent_Store from X;
102     private
103         TheStoredData : Percent := 0;
104     end Percent_Store;
105
106
107     type Minute_Count is new Integer;
108
109     protected type Minute_Count_Store
110     is
111         pragma Priority (10);
112
113         function Get return Minute_Count;
114         --# global in Minute_Count_Store;
115
116         procedure Put(X : in Minute_Count);
117         --# global out Minute_Count_Store;
118         --# derives Minute_Count_Store from X;
119     private
120         TheStoredData : Minute_Count := 0;
121     end Minute_Count_Store;
122
123
124     type Second_Count is new Integer;
125
126     protected type Second_Count_Store
127     is
128         pragma Priority (10);
129
130         function Get return Second_Count;
```

```
131         --# global in Second_Count_Store;
132
133         procedure Put(X : in Second_Count);
134         --# global out Second_Count_Store;
135         --# derives Second_Count_Store from X;
136     private
137         TheStoredData : Second_Count := 0;
138     end Second_Count_Store;
139 end Ice_Types;
140
141 package body Ice_Types
142 is
143     protected body Milliliter_Store is
144         function Get return Milliliter
145         --# global in TheStoredData;
146         is
147         begin
148             return TheStoredData;
149         end Get;
150
151         procedure Put(X : in Milliliter)
152           --# global out TheStoredData;
153           --# derives TheStoredData from X;
154         is
155         begin
156             TheStoredData := X;
157         end Put;
158     end Milliliter_Store;
159
160     protected body Milliliter_Per_Hour_Store is
161         function Get return Milliliter_Per_Hour
162         --# global in TheStoredData;
163         is
164         begin
165             return TheStoredData;
166         end Get;
167
168         procedure Put(X : in Milliliter_Per_Hour)
169           --# global out TheStoredData;
170           --# derives TheStoredData from X;
171         is
172         begin
173             TheStoredData := X;
174         end Put;
175     end Milliliter_Per_Hour_Store;
176
177     protected body Microliter_Per_Hour_Store is
178         function Get return Microliter_Per_Hour
179         --# global in TheStoredData;
180         is
181         begin
182             return TheStoredData;
183         end Get;
184
185         procedure Put(X : in Microliter_Per_Hour)
186           --# global out TheStoredData;
187           --# derives TheStoredData from X;
188         is
189         begin
190             TheStoredData := X;
191         end Put;
192     end Microliter_Per_Hour_Store;
193
194     protected body Minute_Store is
195         function Get return Minute
```

```
196         --# global in TheStoredData;
197         is
198         begin
199            return TheStoredData;
200         end Get;
201
202         procedure Put(X : in Minute)
203           --# global out TheStoredData;
204           --# derives TheStoredData from X;
205         is
206         begin
207            TheStoredData := X;
208         end Put;
209      end Minute_Store;
210
211      protected body Alarm_Signal_Store is
212         function Get return Alarm_Signal
213         --# global in TheStoredData;
214         is
215         begin
216            return TheStoredData;
217         end Get;
218
219         procedure Put(X : in Alarm_Signal)
220           --# global out TheStoredData;
221           --# derives TheStoredData from X;
222         is
223         begin
224            TheStoredData := X;
225         end Put;
226      end Alarm_Signal_Store;
227
228      protected body Percent_Store is
229         function Get return Percent
230         --# global in TheStoredData;
231         is
232         begin
233            return TheStoredData;
234         end Get;
235
236         procedure Put(X : in Percent)
237           --# global out TheStoredData;
238           --# derives TheStoredData from X;
239         is
240         begin
241            TheStoredData := X;
242         end Put;
243      end Percent_Store;
244
245      protected body Minute_Count_Store is
246         function Get return Minute_Count
247         --# global in TheStoredData;
248         is
249         begin
250            return TheStoredData;
251         end Get;
252
253         procedure Put(X : in Minute_Count)
254           --# global out TheStoredData;
255           --# derives TheStoredData from X;
256         is
257         begin
258            TheStoredData := X;
259         end Put;
260      end Minute_Count_Store;
```

```
261
262     protected body Second_Count_Store is
263         function Get return Second_Count
264         --# global in TheStoredData;
265         is
266         begin
267             return TheStoredData;
268         end Get;
269
270         procedure Put(X : in Second_Count)
271           --# global out TheStoredData;
272           --# derives TheStoredData from X;
273         is
274         begin
275             TheStoredData := X;
276         end Put;
277     end Second_Count_Store;
278
279 end Ice_Types;
```

Listing F.3: `Ice_Types` package

```
 1 with Base_Types;
 2 with Bless_Types;
 3 with Ice_Types;
 4 with Pca_Properties;
 5 --# inherit Base_Types,
 6 --#         Bless_Types,
 7 --#         Ice_Types,
 8 --#         Pca_Properties;
 9 package Pca_Types
10 is
11     type Alarm_Type is (
12                         No_Alarm,
13                         Pump_Overheated,
14                         Defective_Battery,
15                         Low_Battery,
16                         POST_Failure,
17                         RAM_Failure,
18                         ROM_failure,
19                         CPU_Failure,
20                         Thread_Monitor_Failure,
21                         Air_In_Line,
22                         Upstream_Occlusion,
23                         Downstream_Occlusion,
24                         Empty_Reservoir,
25                         Basal_Overinfusion,
26                         Bolus_Overinfusion,
27                         Square_Bolus_Overinfusion
28                         );
29
30     protected type Alarm_Type_Store
31     is
32         pragma Priority (10);
33
34         function Get return Alarm_Type;
35         --# global in Alarm_Type_Store;
36
37         procedure Put(X : in Alarm_Type);
38         --# global out Alarm_Type_Store;
39         --# derives Alarm_Type_Store from X;
40     private
41         TheStoredData : Alarm_Type := Alarm_Type'First;
```

```
42      end Alarm_Type_Store;
43
44
45      type Warning_Type is (No_Warning,
46                            Over_Max_Drug_Per_Hour,
47                            Soft_Limit,
48                            Low_Reservoir,
49                            Priming_Failure,
50                            Basal_Underinfusion,
51                            Bolus_Underinfusion,
52                            Square_Bolus_Underinfusion,
53                            Input_Needed,
54                            Long_Pause,
55                            Drug_Not_In_Library,
56                            Hard_Limit_Violated,
57                            Voltage_OOR
58                            );
59
60      protected type Warning_Type_Store
61      is
62          pragma Priority (10);
63
64          function Get return Warning_Type;
65          --# global in Warning_Type_Store;
66
67          procedure Put(X : in Warning_Type);
68          --# global out Warning_Type_Store;
69          --# derives Warning_Type_Store from X;
70      private
71          TheStoredData : Warning_Type := Warning_Type'First;
72      end Warning_Type_Store;
73
74
75      type Status_Type is (Stopped, Bolus, Basal, KVO, Square_Bolus);
76
77      protected type Status_Type_Store
78      is
79          pragma Priority (10);
80
81          function Get return Status_Type;
82          --# global in Status_Type_Store;
83
84          procedure Put(X : in Status_Type);
85          --# global out Status_Type_Store;
86          --# derives Status_Type_Store from X;
87      private
88          TheStoredData : Status_Type := Status_Type'First;
89      end Status_Type_Store;
90
91
92      subtype Flow_Rate is Base_Types.Integer_16;
93
94      protected type Flow_Rate_Store
95      is
96          pragma Priority (10);
97
98          function Get return Flow_Rate;
99          --# global in Flow_Rate_Store;
100
101         procedure Put(X : in Flow_Rate);
102         --# global out Flow_Rate_Store;
103         --# derives Flow_Rate_Store from X;
104     private
105         TheStoredData : Flow_Rate := 0;
106     end Flow_Rate_Store;
```

```
107
108
109     subtype Drug_Volume is Base_Types.Integer_16;
110
111     protected type Drug_Volume_Store
112     is
113         pragma Priority (10);
114
115         function Get return Drug_Volume;
116         --# global in Drug_Volume_Store;
117
118         procedure Put(X : in Drug_Volume);
119         --# global out Drug_Volume_Store;
120         --# derives Drug_Volume_Store from X;
121     private
122         TheStoredData : Drug_Volume := 0;
123     end Drug_Volume_Store;
124
125
126     subtype Drug_Weight is Base_Types.Integer_16;
127
128     protected type Drug_Weight_Store
129     is
130         pragma Priority (10);
131
132         function Get return Drug_Weight;
133         --# global in Drug_Weight_Store;
134
135         procedure Put(X : in Drug_Weight);
136         --# global out Drug_Weight_Store;
137         --# derives Drug_Weight_Store from X;
138     private
139         TheStoredData : Drug_Weight := 0;
140     end Drug_Weight_Store;
141
142
143     type Drug_Concentration is new Integer;
144
145     protected type Drug_Concentration_Store
146     is
147         pragma Priority (10);
148
149         function Get return Drug_Concentration;
150         --# global in Drug_Concentration_Store;
151
152         procedure Put(X : in Drug_Concentration);
153         --# global out Drug_Concentration_Store;
154         --# derives Drug_Concentration_Store from X;
155     private
156         TheStoredData : Drug_Concentration := 0;
157     end Drug_Concentration_Store;
158
159
160     type Drug_Record is record
161         Amount : Drug_Weight;
162         Concentration : Drug_Concentration;
163         Vtbi_Lower_Soft : Drug_Volume;
164         Vtbi_Lower_Hard : Drug_Volume;
165         Vtbi_Typical : Drug_Volume;
166         Vtbi_Upper_Soft : Drug_Volume;
167         Vtbi_Upper_Hard : Drug_Volume;
168         Basal_Rate_Lower_Soft : Flow_Rate;
169         Basal_Rate_Lower_Hard : Flow_Rate;
170         Basal_Rate_Typical : Flow_Rate;
171         Basal_Rate_Upper_Soft : Flow_Rate;
```

```
172         Basal_Rate_Upper_Hard : Flow_Rate;
173         Bolus_Typical : Drug_Volume;
174         Bolus_Time_Typical : Ice_Types.Minute;
175     end record;
176
177     protected type Drug_Record_Store
178     is
179         pragma Priority (10);
180
181         function Get return Drug_Record;
182         --# global in Drug_Record_Store;
183
184         procedure Put(X : in Drug_Record);
185         --# global out Drug_Record_Store;
186         --# derives Drug_Record_Store from X;
187     private
188         TheStoredData : Drug_Record :=
189           Drug_Record'(Amount => Drug_Weight'First,
190                        Concentration => Drug_Concentration'First,
191                        Vtbi_Lower_Soft => Drug_Volume'First,
192                        Vtbi_Lower_Hard => Drug_Volume'First,
193                        Vtbi_Typical => Drug_Volume'First,
194                        Vtbi_Upper_Soft => Drug_Volume'First,
195                        Vtbi_Upper_Hard => Drug_Volume'First,
196                        Basal_Rate_Lower_Soft => Flow_Rate'First,
197                        Basal_Rate_Lower_Hard => Flow_Rate'First,
198                        Basal_Rate_Typical => Flow_Rate'First,
199                        Basal_Rate_Upper_Soft => Flow_Rate'First,
200                        Basal_Rate_Upper_Hard => Flow_Rate'First,
201                        Bolus_Typical => Drug_Volume'First,
202                        Bolus_Time_Typical => Ice_Types.Minute'First
203                        );
204     end Drug_Record_Store;
205
206
207     subtype Drug_Library_Index is Integer range 1 .. Pca_Properties.Drug_Library_Size;
208     type Drug_Library is array (Drug_Library_Index) of Drug_Record;
209
210     protected type Drug_Library_Store
211     is
212         pragma Priority (10);
213
214         function Get(Ind : in Integer) return Drug_Record;
215         --# global in Drug_Library_Store;
216
217         procedure Put(Ind : in Integer; Val : in Drug_Record);
218         --# global in out Drug_Library_Store;
219         --# derives Drug_Library_Store from Drug_Library_Store, Ind, Val;
220     private
221       TheStoredData : Drug_Library := Drug_Library'(others =>
222                             Drug_Record'(Amount => Drug_Weight'First,
223                                          Concentration => Drug_Concentration'First,
224                                          Vtbi_Lower_Soft => Drug_Volume'First,
225                                          Vtbi_Lower_Hard => Drug_Volume'First,
226                                          Vtbi_Typical => Drug_Volume'First,
227                                          Vtbi_Upper_Soft => Drug_Volume'First,
228                                          Vtbi_Upper_Hard => Drug_Volume'First,
229                                          Basal_Rate_Lower_Soft => Flow_Rate'First,
230                                          Basal_Rate_Lower_Hard => Flow_Rate'First,
231                                          Basal_Rate_Typical => Flow_Rate'First,
232                                          Basal_Rate_Upper_Soft => Flow_Rate'First,
233                                          Basal_Rate_Upper_Hard => Flow_Rate'First,
234                                          Bolus_Typical => Drug_Volume'First,
235                                          Bolus_Time_Typical => Ice_Types.Minute'First
236                                         ));
```

```
237     end Drug_Library_Store;
238
239     type Prescription is record
240         Concentration : Drug_Concentration;
241         Initial_Volume : Drug_Volume;
242         Basal_Flow_Rate : Flow_Rate;
243         Vtbi : Drug_Volume;
244         Max_Drug_Per_Hour : Drug_Volume;
245         Minimum_Time_Between_Bolus : Ice_Types.Minute;
246     end record;
247
248     protected type Prescription_Store
249     is
250         pragma Priority (10);
251
252         function Get return Prescription;
253         --# global in Prescription_Store;
254
255         procedure Put(Prescription_In : in Prescription);
256         --# global out Prescription_Store;
257         --# derives Prescription_Store from Prescription_In;
258
259     private
260         TheStoredData : Prescription :=
261           Prescription'(Concentration => 0,
262                         Initial_Volume => 0,
263                         Basal_Flow_Rate => 0,
264                         Vtbi => 0,
265                         Max_Drug_Per_Hour => 0,
266                         Minimum_Time_Between_Bolus => 0
267                        );
268
269     end Prescription_Store;
270
271     type Fault_Record is record
272         Alarm : Alarm_Type;
273         Warning : Warning_Type;
274         Time : Bless_Types.Time;
275     end record;
276
277     protected type Fault_Record_Store
278     is
279         pragma Priority (10);
280
281         function Get return Fault_Record;
282         --# global in Fault_Record_Store;
283
284         procedure Put(X : in Fault_Record);
285         --# global out Fault_Record_Store;
286         --# derives Fault_Record_Store from X;
287     private
288         TheStoredData : Fault_Record := Fault_Record'(Alarm => Alarm_Type'First,
289                                                       Warning => Warning_Type'First,
290                                                       Time => Bless_Types.Time'First
291                                                      );
292     end Fault_Record_Store;
293
294
295     subtype Fault_Log_Index is Integer range 1 .. Pca_Properties.Fault_Log_Size;
296     type Fault_Log is array (Fault_Log_Index) of Fault_Record;
297
298     protected type Fault_Log_Store
299     is
300         pragma Priority (10);
301
```

```
302         function Get(Ind : in Integer) return Fault_Record;
303         --# global in Fault_Log_Store;
304
305         procedure Put(Ind : in Integer; Val : in Fault_Record);
306         --# global in out Fault_Log_Store;
307         --# derives Fault_Log_Store from Fault_Log_Store, Ind, Val;
308     private
309         TheStoredData : Fault_Log := Fault_Log'(others =>
310                                                   Fault_Record'(Alarm => Alarm_Type'First,
311                                                                 Warning => Warning_Type'First,
312                                                                 Time => Bless_Types.Time'First
313                                                                ));
314     end Fault_Log_Store;
315
316
317     type Event_Record is record
318         Time : Bless_Types.Time;
319     end record;
320
321     protected type Event_Record_Store
322     is
323         pragma Priority (10);
324
325         function Get return Event_Record;
326         --# global in Event_Record_Store;
327
328         procedure Put(X : in Event_Record);
329         --# global out Event_Record_Store;
330         --# derives Event_Record_Store from X;
331     private
332         TheStoredData : Event_Record := Event_Record'(Time => Bless_Types.Time'First);
333     end Event_Record_Store;
334
335
336     subtype Event_Log_Index is Integer range 1 .. Pca_Properties.Event_Log_Size;
337     type Event_Log is array (Event_Log_Index) of Event_Record;
338
339     protected type Event_Log_Store
340     is
341         pragma Priority (10);
342
343         function Get(Ind : in Integer) return Event_Record;
344         --# global in Event_Log_Store;
345
346         procedure Put(Ind : in Integer; Val : in Event_Record);
347         --# global in out Event_Log_Store;
348         --# derives Event_Log_Store from Event_Log_Store, Ind, Val;
349     private
350         TheStoredData : Event_Log := Event_Log'(others => Event_Record'(Time => Bless_Types.Time'First));
351     end Event_Log_Store;
352
353
354     type Infusion_Type is (Bolus_Infusion, Square_Infusion, Basal_Infusion, KVO_Infusion);
355
356     protected type Infusion_Type_Store
357     is
358         pragma Priority (10);
359
360         function Get return Infusion_Type;
361         --# global in Infusion_Type_Store;
362
363         procedure Put(X : in Infusion_Type);
364         --# global out Infusion_Type_Store;
365         --# derives Infusion_Type_Store from X;
366     private
```

```
367         TheStoredData : Infusion_Type := Infusion_Type'First;
368     end Infusion_Type_Store;
369
370
371     type Pump_Fault_Type is (Prime_Failure, Pump_Hot, Bubble, Upstream_Occlusion_Fault,
        Downstream_Occlusion_Fault, Overinfusion, Underinfusion);
372
373     protected type Pump_Fault_Type_Store
374     is
375         pragma Priority (10);
376
377         function Get return Pump_Fault_Type;
378         --# global in Pump_Fault_Type_Store;
379
380         procedure Put(X : in Pump_Fault_Type);
381         --# global out Pump_Fault_Type_Store;
382         --# derives Pump_Fault_Type_Store from X;
383     private
384         TheStoredData : Pump_Fault_Type := Pump_Fault_Type'First;
385     end Pump_Fault_Type_Store;
386
387 end Pca_Types;
388
389 package body Pca_Types
390 is
391     protected body Alarm_Type_Store is
392         function Get return Alarm_Type
393         --# global in TheStoredData;
394         is
395         begin
396             return TheStoredData;
397         end Get;
398
399         procedure Put(X : in Alarm_Type)
400           --# global out TheStoredData;
401           --# derives TheStoredData from X;
402         is
403         begin
404             TheStoredData := X;
405         end Put;
406     end Alarm_Type_Store;
407
408     protected body Warning_Type_Store is
409         function Get return Warning_Type
410         --# global in TheStoredData;
411         is
412         begin
413             return TheStoredData;
414         end Get;
415
416         procedure Put(X : in Warning_Type)
417           --# global out TheStoredData;
418           --# derives TheStoredData from X;
419         is
420         begin
421             TheStoredData := X;
422         end Put;
423     end Warning_Type_Store;
424
425     protected body Status_Type_Store is
426         function Get return Status_Type
427         --# global in TheStoredData;
428         is
429         begin
430             return TheStoredData;
```

```
431         end Get;
432
433         procedure Put(X : in Status_Type)
434           --# global out TheStoredData;
435           --# derives TheStoredData from X;
436         is
437         begin
438             TheStoredData := X;
439         end Put;
440     end Status_Type_Store;
441
442     protected body Flow_Rate_Store is
443         function Get return Flow_Rate
444         --# global in TheStoredData;
445         is
446         begin
447             return TheStoredData;
448         end Get;
449
450         procedure Put(X : in Flow_Rate)
451           --# global out TheStoredData;
452           --# derives TheStoredData from X;
453         is
454         begin
455             TheStoredData := X;
456         end Put;
457     end Flow_Rate_Store;
458
459     protected body Drug_Volume_Store is
460         function Get return Drug_Volume
461         --# global in TheStoredData;
462         is
463         begin
464             return TheStoredData;
465         end Get;
466
467         procedure Put(X : in Drug_Volume)
468           --# global out TheStoredData;
469           --# derives TheStoredData from X;
470         is
471         begin
472             TheStoredData := X;
473         end Put;
474     end Drug_Volume_Store;
475
476     protected body Drug_Weight_Store is
477         function Get return Drug_Weight
478         --# global in TheStoredData;
479         is
480         begin
481             return TheStoredData;
482         end Get;
483
484         procedure Put(X : in Drug_Weight)
485           --# global out TheStoredData;
486           --# derives TheStoredData from X;
487         is
488         begin
489             TheStoredData := X;
490         end Put;
491     end Drug_Weight_Store;
492
493     protected body Drug_Concentration_Store is
494         function Get return Drug_Concentration
495         --# global in TheStoredData;
```

```
496         is
497         begin
498            return TheStoredData;
499         end Get;
500
501         procedure Put(X : in Drug_Concentration)
502           --# global out TheStoredData;
503           --# derives TheStoredData from X;
504         is
505         begin
506            TheStoredData := X;
507         end Put;
508     end Drug_Concentration_Store;
509
510     protected body Drug_Record_Store is
511         function Get return Drug_Record
512         --# global in TheStoredData;
513         is
514         begin
515            return TheStoredData;
516         end Get;
517
518         procedure Put(X : in Drug_Record)
519           --# global out TheStoredData;
520           --# derives TheStoredData from X;
521         is
522         begin
523            TheStoredData := X;
524         end Put;
525     end Drug_Record_Store;
526
527     protected body Drug_Library_Store is
528         function Get(Ind : in Integer) return Drug_Record
529         --# global in TheStoredData;
530         is
531         begin
532            return TheStoredData(Ind);
533         end Get;
534
535         procedure Put(Ind : in Integer; Val : in Drug_Record)
536           --# global in out TheStoredData;
537           --# derives TheStoredData from TheStoredData, Ind, Val;
538         is
539         begin
540            TheStoredData(Ind) := Val;
541         end Put;
542     end Drug_Library_Store;
543
544     protected body Prescription_Store
545     is
546         function Get return Prescription
547         --# global in TheStoredData;
548         is
549         begin
550            return TheStoredData;
551         end Get;
552
553         procedure Put(Prescription_In : in Prescription)
554         --# global out TheStoredData;
555         --# derives TheStoredData from Prescription_In;
556         is
557         begin
558            TheStoredData := Prescription_In;
559         end Put;
560     end Prescription_Store;
```

```
561
562     protected body Fault_Record_Store is
563         function Get return Fault_Record
564         --# global in TheStoredData;
565         is
566         begin
567             return TheStoredData;
568         end Get;
569
570         procedure Put(X : in Fault_Record)
571           --# global out TheStoredData;
572           --# derives TheStoredData from X;
573         is
574         begin
575             TheStoredData := X;
576         end Put;
577     end Fault_Record_Store;
578
579     protected body Fault_Log_Store is
580         function Get(Ind : in Integer) return Fault_Record
581         --# global in TheStoredData;
582         is
583         begin
584             return TheStoredData(Ind);
585         end Get;
586
587         procedure Put(Ind : in Integer; Val : in Fault_Record)
588           --# global in out TheStoredData;
589           --# derives TheStoredData from TheStoredData, Ind, Val;
590         is
591         begin
592             TheStoredData(Ind) := Val;
593         end Put;
594     end Fault_Log_Store;
595
596     protected body Event_Record_Store is
597         function Get return Event_Record
598         --# global in TheStoredData;
599         is
600         begin
601             return TheStoredData;
602         end Get;
603
604         procedure Put(X : in Event_Record)
605           --# global out TheStoredData;
606           --# derives TheStoredData from X;
607         is
608         begin
609             TheStoredData := X;
610         end Put;
611     end Event_Record_Store;
612
613     protected body Event_Log_Store is
614         function Get(Ind : in Integer) return Event_Record
615         --# global in TheStoredData;
616         is
617         begin
618             return TheStoredData(Ind);
619         end Get;
620
621         procedure Put(Ind : in Integer; Val : in Event_Record)
622           --# global in out TheStoredData;
623           --# derives TheStoredData from TheStoredData, Ind, Val;
624         is
625         begin
```

```
626             TheStoredData(Ind) := Val;
627         end Put;
628     end Event_Log_Store;
629
630     protected body Infusion_Type_Store is
631         function Get return Infusion_Type
632         --# global in TheStoredData;
633         is
634         begin
635             return TheStoredData;
636         end Get;
637
638         procedure Put(X : in Infusion_Type)
639           --# global out TheStoredData;
640           --# derives TheStoredData from X;
641         is
642         begin
643             TheStoredData := X;
644         end Put;
645     end Infusion_Type_Store;
646
647     protected body Pump_Fault_Type_Store is
648         function Get return Pump_Fault_Type
649         --# global in TheStoredData;
650         is
651         begin
652             return TheStoredData;
653         end Get;
654
655         procedure Put(X : in Pump_Fault_Type)
656           --# global out TheStoredData;
657           --# derives TheStoredData from X;
658         is
659         begin
660             TheStoredData := X;
661         end Put;
662     end Pump_Fault_Type_Store;
663
664 end Pca_Types;
```

Listing F.4: `Pca_Types` package

```
1 package Pca_Properties
2 is
3     Drug_Library_Size : constant Integer := 500;
4     Fault_Log_Size : constant Integer := 150;
5     Event_Log_Size : constant Integer := 1500;
6     KVO_Rate_Constant : constant Integer := 1;
7     KVO_Rate : constant Integer := KVO_Rate_Constant;
8     Max_Rate : constant Integer := 10;
9 end Pca_Properties;
```

Listing F.5: `Pca_Properties` package

```
with Pca_Properties,
    Base_Types,
    Bless_Types,
    Ice_Types,
     Pca_Types;
--# inherit Pca_Properties,
--#         Base_Types,
--#         Bless_Types,
--#         Ice_Types,
--#         Pca_Types;
package Pca_Operation
--# own protected Infusion_Flow_Rate : PCA_Types.Flow_Rate_Store (Priority=>10);
--#     protected System_Status : Pca_Types.Status_Type_Store (Priority=>10);
--#     task mdphw : Max_Drug_Per_Hour_Watcher;
--#     task rc : Rate_Controller;
--#     task pbc : Patient_Bolus_Checker;
is
    procedure Put_Start_Button_Pressed;

    procedure Put_Stop_Button_Pressed;

    procedure Put_Patient_Request_Bolus;

    procedure Put_Clinician_Request_Bolus;

    procedure Put_Bolus_Duration (Bolus_Duration_In : Ice_Types.Minute);

    procedure Get_Infusion_Flow_Rate (Infusion_Flow_Rate_Out : out Pca_Types.Flow_Rate);
    --# global in Infusion_Flow_Rate;
    --# derives Infusion_Flow_Rate_Out from Infusion_Flow_Rate;

    procedure Get_System_Status (System_Status_Out : out Pca_Types.Status_Type);
    --# global in System_Status;
    --# derives System_Status_Out from System_Status;

    procedure Put_Rx (Rx_In : Pca_Types.Prescription);


    task type Max_Drug_Per_Hour_Watcher
    --# global in Infusion_Flow_Rate;
    is
        pragma Priority(10);
    end Max_Drug_Per_Hour_Watcher;

    task type Rate_Controller
    --# global out Infusion_Flow_Rate;
    --#        out System_Status;
    is
        pragma Priority(10);
    end Rate_Controller;

    task type Patient_Bolus_Checker
    is
        pragma Priority(10);
    end Patient_Bolus_Checker;

end Pca_Operation;

package body Pca_Operation
is
    type la_type is (
                     StopButton,
                     TooMuchJuice,
                     PatientButton,
                     ResumeSquareBolus,
```

```
                    ResumeBasal,
                    StartSquareBolus,
                    SquareBolusDone,
                    StartButton);

    Infusion_Flow_Rate : PCA_Types.Flow_Rate_Store;
    System_Status : Pca_Types.Status_Type_Store;

    mdphw : Max_Drug_Per_Hour_Watcher;
    rc : Rate_Controller;
    pbc : Patient_Bolus_Checker;

    procedure Put_Start_Button_Pressed
    is
    begin
        -- TODO: implement event handler
        null;
    end Put_Start_Button_Pressed;

    procedure Put_Stop_Button_Pressed
    is
    begin
        -- TODO: implement event handler
        null;
    end Put_Stop_Button_Pressed;

    procedure Put_Patient_Request_Bolus
    is
    begin
        -- TODO: implement event handler
        null;
    end Put_Patient_Request_Bolus;

    procedure Put_Clinician_Request_Bolus
    is
    begin
        -- TODO: implement event handler
        null;
    end Put_Clinician_Request_Bolus;

    procedure Put_Bolus_Duration (Bolus_Duration_In : ICE_Types.Minute)
    is
    begin
        -- TODO: implement data event handler
    end Put_Bolus_Duration;

    procedure Get_Infusion_Flow_Rate (Infusion_Flow_Rate_Out : out Pca_Types.Flow_Rate)
    is
    begin
        Infusion_Flow_Rate_Out := Infusion_Flow_Rate.Get;
    end Get_Infusion_Flow_Rate;

    procedure Get_System_Status (System_Status_Out : out Pca_Types.Status_Type)
    is
    begin
        System_Status_Out := System_Status.Get;
    end Get_System_Status;

    procedure Put_Rx (Rx_In : Pca_Types.Prescription)
    is
    begin
        -- TODO: implement data event handler
    end Put_Rx;


```

```
131     task body Max_Drug_Per_Hour_Watcher
132     is
133     begin
134         loop
135             --# assert PUMP_RATE;
136             null;
137         end loop;
138     end Max_Drug_Per_Hour_Watcher;
139
140     task body Rate_Controller
141     is
142         la : la_type;
143     begin
144         loop
145             --# assert true;
146             --# assert Rx_APPROVED;
147             --# assert PUMP_RATE;
148             --# assert (la=StopButton) -> HALT;
149             --# assert (la=TooMuchJuice) -> KVO_RATE;
150             --# assert (la=PatientButton) -> PB_RATE;
151             --# assert ((la=StartSquareBolus) or (la=ResumeSquareBolus)) -> CCB_RATE;
152             --# assert ((la=StartButton) or (la=ResumeBasal) or (la=SquareBolusDone)) -> BASAL_RATE;
153             --# assert (PUMP_RATE = 0) -> HALT;
154             --# assert (PUMP_RATE = Pca_Properties.KVO_Rate) -> KVO_RATE;
155             --# assert (PUMP_RATE = Patient_Bolus_Rate) -> PB_RATE;
156             --# assert (PUMP_RATE = Square_Bolus_Rate) -> CCB_RRATE;
157             --# assert (PUMP_RATE = Basal_Rate) -> BASAL_RATE;
158             null;
159         end loop;
160     end Rate_Controller;
161
162     task body Patient_Bolus_Checker
163     is
164     begin
165         loop
166             --# assert true;
167             null;
168         end loop;
169     end Patient_Bolus_Checker;
170
171 end Pca_Operation;
```

Listing F.6: `Pca_Operation` package

Appendix G

AUnit tests for PCA pump dose monitor module

This appendix presents AUnit tests for isolated, sequential module for PCA pump dose monitoring.

```
1  with AUnit.Test_Fixtures;
2
3  package Pca_Pump.Test_Data is
4
5     type Test is new AUnit.Test_Fixtures.Test_Fixture
6     with null record;
7
8     procedure Set_Up (Gnattest_T : in out Test);
9     procedure Tear_Down (Gnattest_T : in out Test);
10
11 end Pca_Pump.Test_Data;
12
13 package body Pca_Pump.Test_Data is
14
15    procedure Set_Up (Gnattest_T : in out Test) is
16       pragma Unreferenced (Gnattest_T);
17    begin
18       null;
19    end Set_Up;
20
21    procedure Tear_Down (Gnattest_T : in out Test) is
22       pragma Unreferenced (Gnattest_T);
23    begin
24       null;
25    end Tear_Down;
26
27 end Pca_Pump.Test_Data;
```

Listing G.1: Package `Pca_Pump.Test_Data`

```
with Gnattest_Generated;

package Pca_Pump.Test_Data.Tests is

   type Test is new GNATtest_Generated.GNATtest_Standard.Pca_Pump.Test_Data.Test
   with null record;

   procedure Test_Sum_Zero (Gnattest_T : in out Test);

   procedure Test_Sum_100 (Gnattest_T : in out Test);

   procedure Test_Read_Dosed_Zero (Gnattest_T : in out Test);

   procedure Test_Increase_Dosed_By_1 (Gnattest_T : in out Test);

   procedure Test_Move_Dosed_First_Element_Zero (Gnattest_T : in out Test);

   procedure Test_Move_Dosed_First_Element_Not_Zero (Gnattest_T : in out Test);

end Pca_Pump.Test_Data.Tests;

with AUnit.Assertions; use AUnit.Assertions;
with Pca_Pump;

package body Pca_Pump.Test_Data.Tests is

   procedure Test_Sum_Zero (Gnattest_T : in out Test) is
      pragma Unreferenced (Gnattest_T);
      Arr : Pca_Pump.Doses_Array := Pca_Pump.Doses_Array'(others => 0);
      Result : Pca_Pump.Drug_Volume := 0;
   begin
      -- Arrange

      -- Act
      Result := Pca_Pump.Sum(Arr);

      -- Assert
      AUnit.Assertions.Assert
        (Result = 0,
         "Sum function result is incorrect.");
   end Test_Sum_Zero;

   procedure Test_Sum_100 (Gnattest_T : in out Test) is
      pragma Unreferenced (Gnattest_T);
      Arr : Pca_Pump.Doses_Array := Pca_Pump.Doses_Array'(others => 0);
      Result : Pca_Pump.Drug_Volume := 0;
   begin
      -- Arrange
      Arr(Pca_Pump.Doses_Array_Index'First) := 51;
      Arr(Pca_Pump.Doses_Array_Index'Last) := 49;

      -- Act
      Result := Pca_Pump.Sum(Arr);

      -- Assert
      AUnit.Assertions.Assert
        (Result = 100,
         "Sum function result is incorrect:" & Pca_Pump.Drug_Volume'Image(Result) & " /= 100");
   end Test_Sum_100;


   procedure Test_Read_Dosed_Zero (Gnattest_T : in out Test) is
      pragma Unreferenced (Gnattest_T);
      Result : Pca_Pump.Drug_Volume;
      Expected  : Pca_Pump.Drug_Volume;
```

```
66    begin
67       -- Arrange
68       Expected := 0;
69
70       -- Act
71       Result := Pca_Pump.Read_Dosed;
72
73       -- Assert
74       AUnit.Assertions.Assert
75         (Expected = Result,
76          "Readed dose incorrect: " &  Pca_Pump.Drug_Volume'Image(Expected) & " /= " &  Pca_Pump.Drug_Volume'
      Image(Result));
77
78    end Test_Read_Dosed_Zero;
79
80    procedure Test_Increase_Dosed_By_1 (Gnattest_T : in out Test) is
81       pragma Unreferenced (Gnattest_T);
82       Pre_Sum : Pca_Pump.Drug_Volume := 0;
83       Post_Sum : Pca_Pump.Drug_Volume := 0;
84    begin
85       -- Arrange
86       Pre_Sum := Pca_Pump.Read_Dosed;
87
88       -- Act
89       Pca_Pump.Increase_Dosed;
90       Post_Sum := Pca_Pump.Read_Dosed;
91
92       -- Assert
93       AUnit.Assertions.Assert
94         (Post_Sum = Pre_Sum + 1,
95          "Total dose not increased properly: " &  Pca_Pump.Drug_Volume'Image(Post_Sum) & " /= " &  Pca_Pump.
      Drug_Volume'Image(Pre_Sum+1));
96    end Test_Increase_Dosed_By_1;
97
98    procedure Test_Move_Dosed_First_Element_Zero (Gnattest_T : in out Test) is
99       pragma Unreferenced (Gnattest_T);
100      Pre_Sum : Pca_Pump.Drug_Volume := 0;
101      Post_Sum : Pca_Pump.Drug_Volume := 0;
102   begin
103      -- Arrange
104      Pre_Sum := Pca_Pump.Read_Dosed;
105
106      -- Act
107      Pca_Pump.Move_Dosed;
108      Post_Sum := Pca_Pump.Read_Dosed;
109
110      -- Assert
111      AUnit.Assertions.Assert
112        (Post_Sum = Pre_Sum,
113         "Total dose changed: " &  Pca_Pump.Drug_Volume'Image(Pre_Sum) & " /= " &  Pca_Pump.Drug_Volume'Image
      (Post_Sum));
114   end Test_Move_Dosed_First_Element_Zero;
115
116   procedure Test_Move_Dosed_First_Element_Not_Zero (Gnattest_T : in out Test) is
117      pragma Unreferenced (Gnattest_T);
118      Pre_Sum : Pca_Pump.Drug_Volume := 0;
119      Post_Sum : Pca_Pump.Drug_Volume := 0;
120   begin
121      -- Arrange
122      Pca_Pump.Increase_Dosed;
123      for I in Pca_Pump.Doses_Array_Index range 1 .. Pca_Pump.Doses_Array_Index'Last-1 loop
124         Pca_Pump.Move_Dosed;
125      end loop;
126      Pre_Sum := Pca_Pump.Read_Dosed;
127
```

```
128       -- Act
129       Pca_Pump.Move_Dosed;
130       Post_Sum := Pca_Pump.Read_Dosed;
131
132       -- Assert
133       AUnit.Assertions.Assert
134         (Post_Sum < Pre_Sum,
135          "Total dose changed: " &  Pca_Pump.Drug_Volume'Image(Pre_Sum) & " should be greater than " &
      Pca_Pump.Drug_Volume'Image(Post_Sum));
136    end Test_Move_Dosed_First_Element_Not_Zero;
137
138 end Pca_Pump.Test_Data.Tests;
```

Listing G.2: Package `Pca_Pump.Test_Data.Tests`