Diplomarbeit

SPARK for Concurrent Programming

Michael Haller
<haller@fmi.uni-passau.de>

29. März 2006

Aufgabensteller und Betreuer:
Prof. Christian Lengauer, Ph. D.
Lehrstuhl für Programmierung
Universität Passau

Abstract

SPARK is a powerful tool for proving the partial correctness of sequential Ada programs. RavenSPARK extends SPARK for multi-tasking in real-time applications. We show that RavenSPARK has limitations regarding proof capabilities. We present an alternative approach to parallel abstract data types in SPARK. Our new approach is linked more closely to the Ada 95 standard and is also applicable to other than real-time programs. We demonstrate the workability of our approach by applying it to the proof of a concurrent communication system.

Acknowledgments

I am grateful to Prof. Christian Lengauer, Ph. D. for his critical interest and liberal attitude in the supervision of this thesis. He also helped me to improve my usage of the English language. I would like to thank Rod Chapman from Praxis High Integrity Systems for his support on the matter of set types in SPARK. I also thank Armin Größlinger for the maintenance of the SPARK installation at the University of Passau.

Contents

List of Figures

Chapter 1

Introduction

The language Ada has been used traditionally for high integrity software, a field in which the absence of errors is crucial. But Ada itself provides no means to assist the programmer in proving the correctness of a program written in Ada. One approach to the verification of Ada programs—and presently the only one which has been used in practice—is SPARK, which is firstly a language based on a subset of Ada with proof annotations and secondly a set of tools for the semi-automatic validation of correctness.

In the past three years an interesting extension to SPARK has been brought to market, namely RavenSPARK, featuring support for parallel programs based on the model of processes and monitors. For parallel programs, a proof of correctness is especially difficult, for the very reason that concurrency causes non-determinism and additional sources of error. The main academic work on proving the correctness of parallel processes and monitors has been done in the 1970s. This marks the state of the art by which RavenSPARK is measured in this thesis.

We evaluate RavenSPARK by re-implementing a parallel program which has been verified manually in Hoare logic in 1977. This program is a small multi-tasking program named COCO. Originally it has been implemented in the language Concurrent Pascal, a Pascal-like language with support for processes and monitors. A re-implementation in Ada is straightforward because Ada is partially based on Pascal and has a task model similar to that of Concurrent Pascal.

We show that RavenSPARK's applicability to COCO proves unsatisfactory. To overcome the deficiencies, we propose a new approach to multi-tasking in SPARK. We refer to SPARK Version 7.2 of Praxis High Integrity Systems [Pra06]. Newer versions might solve some of the issues we address in this thesis.

Readers of this thesis are not required to have any previous knowledge of SPARK. This thesis is also accessible to readers searching an insight into SPARK. A modest background in verification is assumed.

Chapter 2

Specification of COCO

In this chapter we introduce the Coco system by specifying its requirements. We reiterate the original specification of Coco [Len77]. The specification is still relevant for up-to-date applications.

2.1 Hardware Configuration

Coco is the sketch of an operating system for a network switching device. The switch is the central component in a star-wired network topology. It operates at link layer, i. e. there are only direct connections between the switch and the nodes. The links have full-duplex capability. The nature of the nodes linked to the switch is unspecified. The switching device has a single processor and separate input and output interfaces for each port.

2.2 Data Format and Data Handling

The transport unit in the network is the packet. The packet format is only sketched. Packets consist of a header and a variable-sized but bounded-length body. The header contains at least two pieces of information: the packet destination address and the packet priority. Small priority values signify a high priority, zero is the highest priority. The addressing scheme of the network is static.

The switch operates in store-and-forward mode. Packets are stored before they are sent out. The switch is able to modify stored packets as long as the maximum packet size is not exceeded. It could check for transmission errors and drop erroneous packets. It could re-assign priorities. The switch itself cannot create new packets.

To avoid unnecessary and time-consuming copying of the packet data, each incoming packet is assigned to a memory frame which it will occupy until it leaves the system.

The switch has to process the packets in the order of their priorities. The priority field in the packet header is set by the sending node or modified during the data handling by the switch. Packets with the same priority are handled in first-in first-out order. The data handling and output is always done for the most urgent packet.

2.3 System Behavior

We describe the system behavior on the basis of a packet traveling through the system. Figure 1 shows possible paths.

Once a packet enters the system, an empty memory frame is reserved for it and filled with the packet data. If no empty memory frame is available, the reception of the packet is delayed. The filled memory frame is entered into a priority queue. From this queue, the most urgent packet is selected for data handling. After the packet has been processed, it is put into another priority queue associated with the destination output port. From there it leaves the system. When the output of the packet data has been completed, the memory frame is freed.

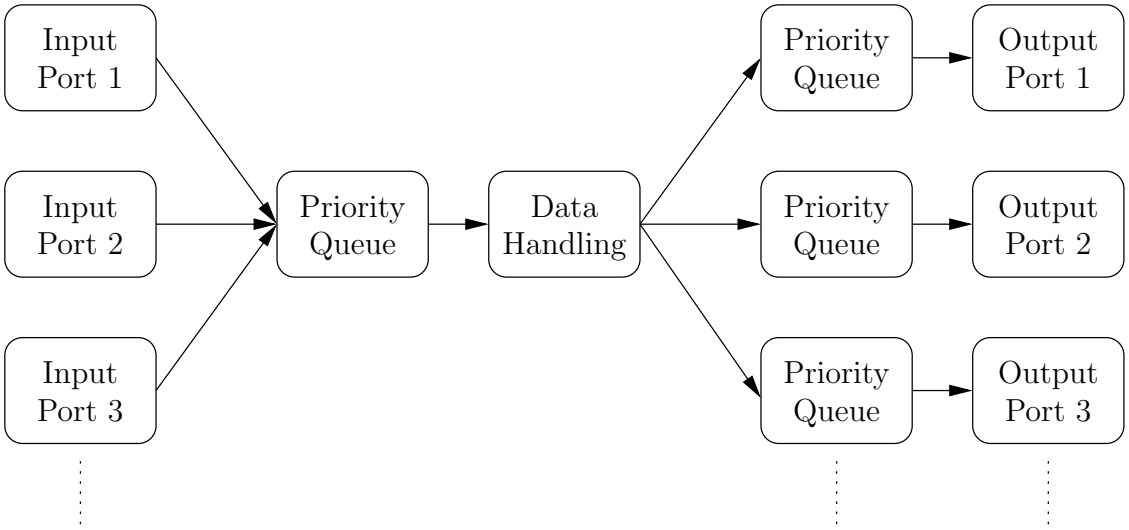

Figure 1: Packet processing

Chapter 3

Overview of SPARK

In this chapter we give an overview of the SPARK language and its tools as well as of RavenSPARK. We have been using SPARK Version 7.2 of Praxis High Integrity Systems [Pra06]. We only address features required for the understanding of this thesis and refer to the manuals distributed with the SPARK tools for reference.

We elaborate especially on abstract data types and objects because the next chapter builds on them. The understanding of the SPARK tools' interactions is important for the proof of COCO in Chapter 6.

3.1 SPARK Annotations

The SPARK language is a restricted subset of Ada with annotations. The annotations are embedded in especially marked Ada comments (two dashes followed by a diamond). Thus, each valid SPARK program is a valid Ada program which can be translated by every Ada compiler. Barnes [Bar03] describes the language subset and the annotations in detail.

Only procedure, function and package declarations and statements can be annotated. In general, the annotations are put immediately after a declaration or—if a defining body follows—before the keyword **is**. Alternatively bodies can be hidden from the analysis by the SPARK tools with the **hide** annotation.

3.1.1 Statements

In general, no statements other than loop statements are annotated. After the loop head, the **assert** annotation indicates a loop invariant.

3.1.2 Procedures and Functions

Each procedure must be annotated at least with the derives annotation which specifies the data flow. A global annotation must be used if a procedure accesses non-local variables. The kind of access (read or write) is indicated by the modes in and out in the same manner as in procedure parameter declarations. The pre and post annotations specify conditions which must hold before and after a procedure call, respectively. The following example illustrates their use:

```
Sum : Integer;

procedure Inc (X : in Integer)
--# global in out Sum;
--# derives Sum from Sum, X;
--# pre X > 0;
--# post Sum = Sum~ + X;
is
begin
   Sum := Sum + X;
end Inc;
```

This procedure increments the global variable Sum (read and write access) by the value of X. The new value of Sum depends on its old value and the value of X. We chose the pre-condition arbitrarily. The post-condition states that the value of Sum is the previous value of Sum (expressed by an appended tilde) plus the value of X.

Functions are pure, i. e. they have no side-effects. The only valid annotations are return and global (with mode in because global variables can only be accessed read-only). The result of the function is specified by a return annotation rather than a post-condition.

The proof annotations pre, post, return and assert are in the language of first-order logic. The syntax is similar to Ada's with a few extensions of which we only use the universal quantifier. Its syntax is (for all X in T => P) which means: for all values of the variable X in the range of type T the predicate P holds.

3.1.3 Packages

Packages are the largest building blocks in Ada. Packages can have child packages and, thus, form a package hierarchy. Packages can import other packages with the Ada with clause. The imported packages *and* the parent packages must be listed in the SPARK inherit annotation.

Variables declared at package level must itself appear in an own annotation. The own annotation appears twice: in the package specification it either defines names

of abstract variables or lists concrete variables, in the package body it associates concrete variables with the abstract variables. When variables are initialized at package level, they must be listed in an initializes annotation.

The state of a package consists of the configuration of its variables. These concrete variables can be associated with one or more abstract variables in the own annotations. These abstract variables, thus, also represent the state of a package. Packages with a state are *abstract data objects* (in SPARK terminology also called abstract state machines).

Procedures and functions can be annotated in the package body and in the package specification. The annotations in the package specification usually reference only formal parameters and abstract variables from the own annotation. The pre-conditions in the body must follow from the pre-conditions in the specification, and the post-conditions in the specification must follow from the post-conditions in the body. This is called *refinement*. For the sake of completeness it must be pointed out that it is possible to write packages with no abstract variables, and consequently with no refinement and no annotations in the body.

The function annotation declares a proof function. It is like an ordinary function declaration in Ada but can only be referenced in other annotations. The function annotation contains just a function signature. A semantics is associated by proof rules which we describe in Section 3.3.

Proof functions need not always be declared explicitly. For each ordinary function declared in the package, a corresponding proof function is declared implicitly. The proof function's signature is adopted from the concrete function declaration. One parameter is added for each variable in the global annotation. The imported variables in the global annotation are treated as further additional parameters.

The type annotation declares a proof type. Proof types can only be referenced in other annotations. For example, the type of an abstract variable in the own annotation might be such a proof type.

Figure 2 illustrates the use of some possible annotations in a package. The left side of the figure is the package specification, the right side the package body. The only variable declared at package level is the variable Sum. It is initialized to zero in the package initialization part. The abstract variable State denotes the state of the package. Its type is the proof type State_Type (proof types bear always the attribute abstract). Procedure Inc is the same as in the previous example. In the package specification, all annotations reference the abstract variable State because the global variable Sum is not visible. For this reason we require a proof function to formulate the post-condition. The proof function Add is just a signature; its semantics is defined in Section 3.3. The own annotation in the package body gives the abstract variable State and the proof type State_Type a semantics.

```
package Calc
--# own State : State_Type;
--# initializes State;
is
   --# type State_Type is abstract;

   --# function Add (A, B : in State_Type;
   --#   J :  in  Integer)  return Boolean;

   procedure Inc (X : in  Integer);
   --# global in out State;
   --# derives State from State, X;
   --# pre X >= 0;
   --# post Add (State, State~, X);

end Calc;
```

```
package body Calc
--# own State is Sum;
is
   Sum : Integer;

   procedure Inc (X : in  Integer)
   --# global in out Sum;
   --# derives Sum from Sum, X;
   --# pre X >= 0;
   --# post Sum = Sum~ + X;
   is
   begin
      Sum := Sum + X;
   end Inc;

begin
   Sum := 0;
end Calc;
```

Figure 2: Example of a package

3.1.4 Classes

In object-oriented programming, classes are *abstract data types*. Ada realizes classes in the following way. Class attributes are implemented in a record aggregate with restricted visibility. An object is instantiated when a variable of this record type is declared. Class methods are ordinary procedures and functions which have a parameter of the class type. The explicit type parameter in the object methods is a particular characteristic of Ada. We do not consider inheritance, as Coco does not require it.

Figure 3 shows an exemplary class definition. It requires a proof function to formulate the post-condition because the class attribute Sum is not visible and, therefore, cannot be referenced directly. The class methods in the package body are not annotated in Spark. It would be conceivable to specify pre- and post-conditions in the body which directly reference the class attributes. Here Spark does not take advantage of the abstraction offered by Ada.

In Figure 3 we name the abstract data type after the abstract data object of Figure 2. Both provide the method Inc which in both cases has the same semantics. Both provide the same attribute Sum. We want to show how fundamentally differently abstract data types and abstract data objects are implemented in Ada. This becomes

important for our discussion in Section 4.3.1. There we argue which of them has the closest resemblance to a parallel abstract data type.

```
package Container is

   type Calc is private;

   --# function Add (A, B : in Calc;
   --#   J : in Integer) return Boolean;

   procedure Inc
      (C : in out Calc; X : in Integer);
   --# derives C from C, X;
   --# pre X >= 0;
   --# post Add (C, C~, X);

private
   type Calc is record
      Sum : Integer;
   end record;
end Container;
```

```
package body Container is

   procedure Inc
      (C : in out Calc; X : in Integer) is
   begin
      C.Sum := C.Sum + X;
   end Inc;

end Container;
```

Figure 3: Example of a class

3.2 RavenSPARK and the Ravenscar Profile

Pure SPARK is sequential. Multi-tasking has only been added recently to SPARK by RavenSPARK. It uses Ada task types and protected types. Task objects (i. e. objects of type task) are processes. Protected types are abstract data types which provide mutually exclusive operations for tasks to access their private data.

Figure 4 shows an exemplary protected type. It provides the same procedure Inc and has the same private variable Sum as the examples in Figures 2 and 3. The pragmas (compiler directives) are RavenSPARK-specific: Pragma Ravenscar activates the Ravenscar Profile which we describe below. Pragma Priority specifies a priority for the real-time scheduler.

In packages, abstract variables can be declared with an own annotation. These abstract variables denote the state of the package. Protected types have a state, too, but no own annotation. Instead, RavenSPARK implicitly declares a state variable which has the name of the protected type.

RavenSPARK disallows pre and post annotations in a protected type declaration. We elaborate on this in Section 4.1.

```
pragma Ravenscar;
package Container is

   protected type Calc is
      pragma Priority (System.Default_Priority);

      procedure Inc (X : in Integer);
      --# global in out Calc;
      --# derives Calc from Calc, X;

   private
      Sum : Integer := 0;
   end Calc;

end Container;
```

```
package body Container is

   protected body Calc is

      procedure Inc (X : in Integer)
      --# global in out Sum;
      --# derives Sum from Sum, X;
      --# pre X >= 0;
      --# post Sum = Sum~ + X;
      is
      begin
         Sum := Sum + X;
      end Inc;

   end Calc;

end Container;
```

Figure 4: Example of a protected type

Protected objects (i. e. objects of a protected type) are very similar to monitors with the added advantage that the low-level wait and signal scheduling operations are substituted by entry barriers. We describe the general Ada synchronization mechanism first [Ada95, Sect. 9.5.2].

Entries are procedures with a barrier condition. When a task calls an entry, the run-time system evaluates the barrier. If it is satisfied the task is allowed to execute, otherwise the task execution is blocked and the task is put into a waiting queue which is associated implicitly with the entry. Each time a procedure or entry call is finished, i. e. a changed state of the protected object becomes visible, the run-time system re-evaluates the barriers of all non-empty entry queues. The queues are managed in first-in first-out order, the evaluation order of the barriers is non-deterministic. When a task is re-activated, the barrier condition is guaranteed to be true.

Figure 5 shows the syntax of an entry. The entry Get fetches some element of the buffer array Data. The barrier is the expression Count $>$ 0. Count is a private integer variable of the protected type. When a task calls this entry, it will be blocked until Count $>$ 0 evaluates to true.

```
entry Get (X : out Element) when Count > 0 is
begin
   X := Data (Count);
   Count := Count − 1;
end Get;
```

Figure 5: Example of an entry procedure

RavenSpark enforces the activation of the Ravenscar Profile.[1] The Ravenscar Profile has been incorporated into the Real-Time Annex of the Ada standard. We base our description on the guide to the Ravenscar Profile [BDV03].

The Ravenscar Profile defines a subset of the Ada task model. It is activated with the pragma Ravenscar which, in turn, activates a number of other pragmas to configure the Ada compiler and the Ada run-time system.

The goals of the Ravenscar Profile are a deterministic selection of tasks, hard deadlines and a simple schedulability analysis. This is accomplished by several restrictions. The Ravenscar Profile allows only one entry per protected type. The entry queue size is limited to one. Every entry may only be called by one distinct task. The entry barrier is restricted to a static Boolean expression. We discuss the implications of these restrictions in Section 4.1.

3.3 SPARK Tools

The Spark distribution contains a number of tools. We describe three of them: the Spark Examiner, the Spade Automatic Simplifier and the Spade Proof Checker.

The Spark Examiner performs a static analysis of the annotated Ada source code. It checks that the source code conforms to the language specification, that there are no uninitialized or unused variables and that there are no lost variable updates. It also checks that the use of variables matches their interdependence annotations and that there is no unreachable code. After this analysis, the Examiner outputs files with verification conditions, proof rules and FDL type declarations. We describe these in turn below.

The Examiner generates the verification conditions from the proof annotations. The Examiner also inserts automatically conditions which are directly derived from the Ada type declarations, to check for range overflows. The verification conditions are

[1] The Ravenscar Profile is the result of the 8th International Real-Time Ada Workshop that was held in Ravenscar, North Yorkshire, UK.

conjectures consisting of hypotheses $H_1, \ldots, H_n$ and conclusions $C_1, \ldots, C_m$. It has to be proved that $(H_1 \wedge \ldots \wedge H_n \Rightarrow C_1 \wedge \ldots \wedge C_m)$.

The Examiner translates all constants, variables, functions and type identifiers into FDL type declarations. FDL (abbr. for Functional Description Language)[2] is a Pascal-based language. The SPARK tools use only the declarative part of FDL. The type system of FDL is weaker than that of Ada. This is compensated for by the generated verification conditions.

FDL identifiers are set in lowercase, all dots and single quotation marks are replaced by two underlines. For our proof function Add of the example in Figure 2, the FDL declarations are as follows:

```
type state_type = record
       sum: integer
     end;

function add(state_type, state_type, integer): boolean;
```

The proof type State_Type is translated into an FDL record and contains the variables listed in the own annotation of the package body, in this case only the variable Sum. The signature of the function annotation is translated into an FDL function declaration.

As their names suggest, the SPADE Automatic Simplifier and the SPADE Proof Checker depend on SPADE's language FDL, which explains why the Examiner translates the Ada declarations into FDL.

The Simplifier is an automated theorem prover with a Prolog system in the background. The Simplifier takes the verification conditions and the FDL declarations as input and simplifies the verification conditions according to proof rules as much as possible. The proof rules are built-in, generated by the Examiner or written manually.[3] Each proof rule has one of the following, self-explanatory formats:

```
X may_be_replaced_by Y.

X may_be_replaced_by Y if [ C1, ..., Cn ].

X & Y are_interchangeable.

X & Y are_interchangeable if [ C1, ..., Cn ].
```

[2] FDL is the language of SPADE (abbr. for Southampton Program Analysis Development Environment). The SPARK Examiner is the successor of the SPADE Verification Condition Generator, the acronym SPARK stands for SPADE Ada Kernel.

[3] With the upcoming SPARK Version 7.3 the Simplifier should also be able to use manually written proof rules. As we are still using Version 7.2, we help ourselves by appending our manual proof rules to the automatically generated ones before we start the Simplifier.

X **may_be_deduced**.

X **may_be_deduced_from** [C_1, ..., C_n].

X, Y, C_1, ..., C_n are Prolog variables. The syntax of the verification conditions and the proof rules has an affinity to both FDL and Prolog. Atoms starting with a capital letter are Prolog variables. Every clause must be followed by a period. A single underscore means "any variable". All FDL identifiers are accessible as Prolog predicates.

The built-in (non-trivial) predicates used in the proof rules of this thesis are:

Predicate	**Description**
element(A, [J])	get element J of array A
update(A, [J], X)	update element J of array A with value X
fld_F(R)	get field F of record R
upf_F(R, X)	update field F of record R with value X
mk__R(F_1 := X_1, ..., F_n := X_n)	record aggregate R with fields F_1, ..., F_n
for_all(VAR : TYPE, P)	universal quantifier for predicate P
for_some(VAR : TYPE, P)	existential quantifier for predicate P

The proof rules are arbitrarily grouped into rule families. Each rule family has a header with type declarations for each predicate used in the rules. The type system of the proof rules is even weaker than that of FDL, consisting only of four types (integer, real, enumeration and any). Each rule must be preceded with the rule family name and a distinct number in parentheses.

We complete the example of Figure 2 with the definition of the proof function Add. It has to be defined as a proof rule:

rule_family calc_rules:
add(A, B, J) **requires** [A: any, B: any, J: i].

calc_rules(1): add(A, B, J) **may_be_replaced_by** fld_sum(A) = fld_sum(B) + J.

The rule family name is calc_rules. The type i means integer. The rule calc_rules(1) can be referenced in any proof.

The simplified verification conditions (if not simplified to true) can then be proved manually with the Checker. The Checker is implemented as a front-end to a Prolog system. It provides a command line interface to the user. The Checker commands are explained during their usage in the proof sessions of Chapter 6.

The interactions between the SPARK tools are summarized in Figure 6.

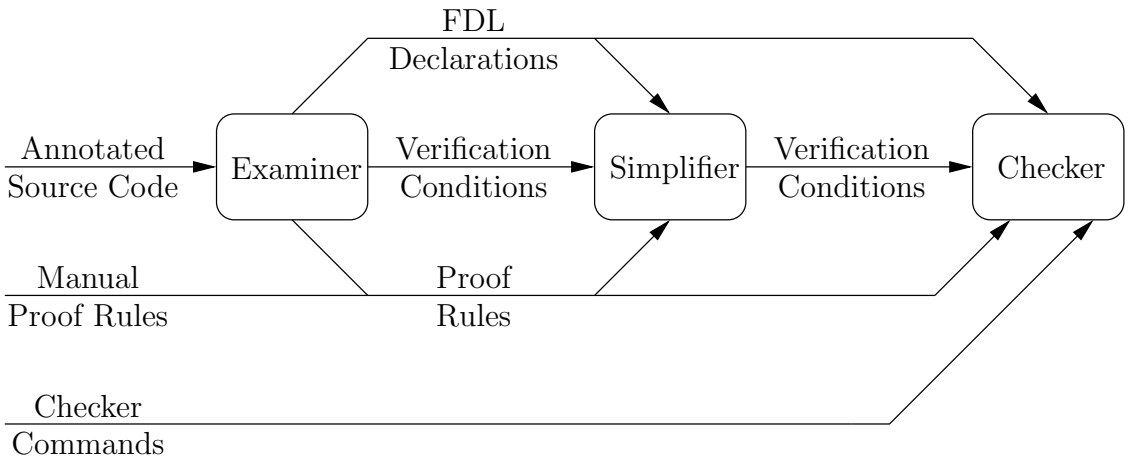

Figure 6: Work flow of Examiner, Simplifier and Checker

Chapter 4

Alternative Version of SPARK

Before we start into the design and implementation of an alternative approach to multi-tasking in SPARK, let us address the deficiencies we observed.

4.1 Deficiencies of RavenSPARK

We focus our criticism on RavenSPARK and, in particular, on protected types. The original design of CORO requires monitors. In Ada, protected objects are similar to monitors. We show that RavenSPARK's protected objects have only little resemblance to monitors.

The Ravenscar Profile imposes certain restrictions on the Ada run-time system, any violation leads to a run-time error. RavenSPARK makes sure that, with the help of language restrictions and static analysis, several potential run-time errors cannot occur [Spa05]. Moreover, RavenSPARK does support verification for sequential code but it has, as we show, virtually no support for the verification of protected types. Even if we changed the overall design of CORO to meet the specification of RavenSPARK, nothing would be gained because we could not verify the correctness of our protected types.

4.1.1 Entry Barriers

RavenSPARK forces entry barriers to be Boolean elements of the protected type [Spa05, Sect. 3.3.3.8]. In particular, functions are not allowed to be evaluated in entry barriers. The guide to the Ravenscar Profile justifies this with the avoidance of side-effects [BDV03, Sect. 4.1.1]. But functions in SPARK have no side-effects.

The use of a single variable as entry barrier introduces the necessity of additional Boolean variables for book keeping. Entry barriers may only use these variables

instead of dynamically evaluated functions. The dynamic evaluation must be made static. These variables have to be updated in every procedure and entry call to reflect the current state of the monitor.

4.1.2 Initialization

The initialization of protected objects is handled by the run-time system. Each private variable is assigned one evaluated static expression. RavenSPARK disallows functions to be evaluated here, too. One of the implications is that the initialization of an n-sized array with n distinct values for a reasonable large value of n becomes tedious.

A further issue is that the Examiner does not generate any verification conditions for the initialization part. One cannot prove that an invariant holds after the initialization.

4.1.3 Task Model

The Ada 95 Rationale states that the control-oriented Ada 83 rendezvous model for task synchronization proved to be unsatisfactory [BB+95, Sect. II.9]. The designers of Ada 95 preferred an object-oriented approach and introduced protected types. RavenSPARK incorporates tasks and protected types but, instead of relying on the old Ada 83 or the new Ada 95 task model, RavenSPARK is based on the Ravenscar Profile. The Ravenscar Profile restricts protected types by disallowing multiple entries. It restricts the run-time system by limiting the size of entry queues to one and, thus, eliminates all implicit waiting queues.

In RavenSPARK, protected objects cannot be components of other data types. This implies that one cannot construct an explicit waiting queue. RavenSPARK pre-defines a so-called suspension object with operations Suspend_Until_True and Set_True which can only be called at task level—a necessary restriction because otherwise there would be a monitor hierarchy: if Suspend_Until_True were called during a protected type operation, a deadlock would occur.

On the one hand the Ravenscar Profile forbids the use of the Ada 83 task synchronization, on the other hand it practically re-introduces a primitive wait and signal mechanism by suspension objects and stripped down protected types, placing the burden of scheduling again on tasks.

We do not question the usefulness of the Ravenscar Profile for real-time applications. But it is an annex to the Ada language standard and obviously not intended for general purpose.

4.1.4 Verification

RavenSpark deliberately forbids any variables of a protected type in proof annotations [Spa05, 3.6.2]. This implies the absence of any proof annotations in a protected type declaration.

Interestingly enough, sequential abstract data types can only have proof annotations in their declarative part but not in their body (cf. Sect. 3.1.4). This is in contrast to parallel abstract data types which must not have any proof annotations in their declarative part.

In principle, it is possible to use proof annotations in a protected type body but their usefulness is limited. Due to an unresolved problem in RavenSpark, no proof functions can be declared within a protected type declaration.[4] This implies that no proof functions can be used in the proof annotations and, therefore, the expressive power of the annotations is limited.

We speculate that this is strongly related to the fact that the state variable of the protected type is declared implicitly by RavenSpark and bears the protected type's name. A proof function declaration would require the state variable as formal parameter. The name of this variable is the type name. But what are this variable's type and the name of this type? This question remains unanswered, and consequently no proof function can be declared.

4.2 Proposed Solution

As discussed in the previous section, protected types in RavenSpark have several deficiencies. To make our criticism constructive, we suggest some modifications to RavenSpark. The result is a new approach to multi-tasking in Spark. We want Spark to support protected types that are linked more closely to Ada 95. To achieve this, we take RavenSpark as a foundation but do not rely on the Ravenscar Profile. Without the Ravenscar Profile, the name RavenSpark does no longer apply. We call our approach "PassauSpark".

This section is structured as a list of changes relative to RavenSpark. Again: we focus only on protected types. First we describe the context-free syntax, then the altered semantics.

[4] We reported this issue to Praxis High Integrity Systems and it was allocated tracking number SEPR 1807. As of the date of this printing, it remains unresolved.

4.2.1 Syntax

The basis of our alternative protected type syntax is the Ada 95 standard [Ada95, Sect. 9.4] because even a modified protected type must still be valid Ada.

For interested implementers we specify the context-free syntax of PassauSpark's protected type in Figure 7. The grammar used here is in the same simple variant of Backus-Naur Form as used in the Ada 95 standard. All non-terminals that are not specified here are defined in the grammar of the Spark language [Bar03].

```
protected_type_declaration ::=
   protected type defining_identifier
   [known_discriminant_part]
   own_variable_clause
[ invariant_clause ]
   is
      { protected_declarative_item }
[ private
      { protected_element_declaration }
      | hidden_part ]
   end defining_identifier;

invariant_clause ::=
   --# invariant predicate;

protected_declarative_item ::=
      proof_function_declaration
   | proof_type_declaration
   | subprogram_declaration
   | entry_declaration

protected_element_declaration ::=
   { component_declaration }

hidden_part ::=
   --# hide defining_identifier;

entry_declaration ::=
   entry_specification;
   procedure_annotation

entry_specification ::=
   entry defining_identifier [formal_part]

protected_body ::=
   protected body defining_identifier
[ invariant_clause ]
   is
    { protected_operation_item }
   end defining_identifier;

protected_operation_item ::=
      subprogram_body
   | entry_body

entry_body ::=
   entry_specification when condition
   procedure_annotation
   is subprogram_implementation
```

Figure 7: BNF grammar of a protected type in PassauSpark

4.2.2 Initialization

In RavenSPARK, the protected object initialization is limited to static expressions for which no verification conditions are generated.

We extract the initialization of the private variables into a special procedure, named Initialize, which must not have a pre-condition other than true. This enables a more expressive initialization. As the initialization is an ordinary procedure, the Examiner will automatically generate appropriate verification conditions for it.

The newly introduced initialization procedure requires a change in the main program. The initialization procedures of all protected objects must be executed by the main task before any other task is allowed to start. We propose the following prototype for the main program:

```
-- include all packages with protected types and task types here
procedure Main is
begin
   -- call protected objects' initialization procedures here
   declare
      -- declare all task objects here ⇒ tasks start executing here
   begin
      null; -- null statement as in RavenSPARK
   end;
end Main;
```

4.2.3 Proof Annotations

In RavenSPARK, the state variable of a protected type is declared implicitly. We suggest that, similarly to a package specification, an own and a type annotation must be used to declare an abstract state variable. With this modification, it is possible to declare proof functions for protected types. One of their obligatory parameters is the state variable.

In contrast to RavenSPARK, we allow pre, post and return annotations in the procedure and function annotations of the protected type declaration.

The invariant of a protected object describes the consistency of the private variables and must be true before and after calls of sub-programs, in addition to the pre- and post-conditions. In RavenSPARK, the invariant must be included in every pre and post annotation. We make the invariant explicit: a protected type invariant may be defined with the invariant annotation. Invariant is a reserved but currently unused keyword in SPARK. Both the protected type declaration and the body may have different invariant annotations.

Figure 8 shows the example of Figure 4 in PassauSpark. We chose the invariant Sum >= 0 arbitrarily. In the protected type declaration, this invariant is specified as a proof function.

```
package Container is

   protected type Calc
   --# own State : State_Type;
   --# invariant Sum_Is_Positive (State);
   is
      --# type State_Type is abstract;

      --# function Add (A, B : in State_Type;
      --#   J :  in  Integer)  return Boolean;

      --# function Sum_Is_Positive
      --#   (S :  in  State_Type) return Boolean;

      procedure Initialize ;
      --# global out State;
      --# derives State from ;

      procedure Inc (X : in  Integer);
      --# global in out State;
      --# derives State from State, X;
      --# pre X >= 0;
      --# post Add (State, State~, X);

   private
      Sum : Integer;
   end Calc;

end Container;
```

```
package body Container is

   protected body Calc
   --# invariant Sum >= 0;
   is

      procedure Initialize
      --# global out Sum;
      --# derives Sum from ;
      is
      begin
         Sum := 0;
      end Initialize;

      procedure Inc (X : in Integer)
      --# global in out Sum;
      --# derives Sum from Sum, X;
      --# pre X >= 0;
      --# post Sum = Sum~ + X;
      is
      begin
         Sum := Sum + X;
      end Inc;

   end Calc;

end Container;
```

Figure 8: Example of a protected type in PassauSpark

4.2.4 Synchronization

So far we have viewed protected types as abstract data types. Now we look at synchronization.

Without the Ravenscar Profile the default Ada run-time system is used. This imposes changes on the handling of entries. The waiting queue sizes become virtually

unlimited. A protected type may have multiple entries and every task may call every entry.

A monitor is a collection of private data and public operations shared by several tasks. There are two levels of scheduling. *Short-term scheduling* is the mutual exclusion of all executions of monitor operations. It frees us from proving the absence of interferences introduced by concurrency. Short-term scheduling is handled by the run-time system, so there are no proof obligations. *Medium-term scheduling* is the synchronization of tasks depending on the state of the monitor, specified by the programmer. It is normally implemented by wait and signal operations, calls of which are written into the monitor operations by the programmer.

Hoare [Hoa74] defines the following partial correctness axioms for monitor operations:

$\{\text{true}\}$	initialization	$\{J\}$
$\{J \wedge P\}$	each procedure	$\{J \wedge R\}$
$\{J\}$	q.wait	$\{J \wedge B\}$
$\{J \wedge B\}$	q.signal	$\{J\}$

J is the monitor invariant. For each procedure, P is a pre-condition and R a post-condition. B is a Boolean expression that describes the queue q.

Ada implements automatic signaling. The Ada run-time system administers an implicit waiting queue for each entry. It tests repeatedly the entry barrier B and reactivates waiting tasks. Howard [How76] proposed additional predicates to prevent tasks to be left waiting by recording the queue length in history variables, among other signaling schemes for automatic signaling.

Ada also implements automatic waiting. The run-time system automatically queues calling tasks when the entry barrier B evaluates to false. Consequently there are no explicit wait and signal operations in Ada.

Eventually, there is no other proof obligation than the invariant holding before and after monitor operations because the run-time system completely takes care of medium-term scheduling. This implies that the verification conditions are the same as those generated for a sequential abstract data type with only one difference: B is guaranteed to hold when a task finally executes the entry body.

4.3 Pre-Processor

Before we can use PassauSpark for the implementation of Coco, we must implement it first. Unfortunately the source code of the Spark tools is not public. As

a temporary solution we developed a pre-processor to the SPARK tools. This pre-processor must transform the input sources from PassauSPARK to pure SPARK such that the Examiner accepts it as legal SPARK *and* generates the correct verification conditions. A clean solution would be to modify the SPARK tools directly but, without the source being public, this can only be done by the right-holders of SPARK.

The output of the pre-processor is a valid Ada program with valid SPARK annotations but it should not be compiled and must not be executed. Its sole purpose is to leverage the Examiner to generate the right verification conditions.

As shown in the previous section, the proof obligations for protected types are strongly related to other abstract data types. Our strategy is to convert all protected types into sequential types, so that the Examiner will generate the appropriate verification conditions. Entries are to be converted into procedures and we must force the Examiner to incorporate the entry barrier as a hypothesis into the verification conditions of the entry.

In the following sub-sections we firstly choose the right sequential type representation and secondly deal with entries and entry barriers.

We do not intend to prove any properties of tasks in COCO, only of protected types. For this reason we do not handle tasks in the pre-processor.

4.3.1 Selection of the Sequential Type

At first glance it might be tempting to transform a protected type (a parallel abstract data type) into a class (a sequential abstract data type). But there are some obstacles.

There is a fundamental syntactic difference between protected types and classes: while the object parameter of protected object operations is prepended to the operation call (i. e. it is implicit to the operation), object methods require an explicit object parameter. This leads to a more complex transformation, especially for the generation of appropriate global and derives annotations.

RavenSPARK takes Ada's division of a protected type into declaration and body into account and requires annotations (at least global and derives annotations) in both parts. Classes, on the other hand, have no annotations in the body.

A single protected object has more resemblance with a package (an abstract data object). Operations of packages have no object parameter because every package represents only one single object. SPARK also honors the partition into package specification and body.

As the pre-processor does not handle tasks, it does not need to handle protected objects. This enables us to assume one single protected object per protected type. This again allows us to choose between abstract data types and objects. For the reasons just given, we conclude that, in this situation, the package is more suitable than the class.

Our pre-processor generates, for every protected type, one child package with the name of the protected type. This approach is possible because in RavenSpark protected types are always declared at the package level.

4.3.2 Protected Type Conversion

Every protected type is transformed into a child package. We always insert the pragma Elaborate_Body into the parent package specification because the package body may become empty, which would otherwise trigger an error.

The inherit annotation of the child package must list its parent package name together with the inherited packages from the parent.

The private variables of the protected type declaration are transferred into the new package body. Spark requires the initialization of the private variables but this is handled already by the procedure Initialize. So we pretend to initialize these variables by outputting an initializes annotation and hiding the package initialization with a hide annotation. An own annotation in the package body is added, which lists the private variables.

The invariant specified in the invariant annotation is prepended to all post annotations and to all pre annotations except for the initialization procedure.

Figure 9 shows the protected type example from Figure 8 after the pre-processing. It can be further processed by the Examiner.

4.3.3 Entry Conversion

Entries are transformed to procedures. The entry barrier is enforced by the run-time system. We must make sure that the Examiner knows that the entry barrier is true before the first statement. This is accomplished by a local procedure which does nothing, but which is declared axiomatically to satisfy the entry barrier on exit. This procedure must be called before the first statement. Its body, i. e. the null statement, has no effect and must of course be hidden from Spark. The post annotation of this procedure specifies the entry barrier.

```
package Container is
   pragma Elaborate_Body (Container);
end Container;
```

```
package body Container is
end Container;
```

```
--# inherit Container;
package Container.Calc
--# own State : State_Type;
--# initializes State;
is
   --# type State_Type is abstract;

   --# function Add (A, B : in State_Type;
   --#   J :  in  Integer)  return Boolean;

   --# function Sum_Is_Positive
   --#   (S :  in  State_Type) return Boolean;

   procedure Initialize ;
   --# global out State;
   --# derives State from ;
   --# post Sum_Is_Positive (State);

   procedure Inc (X : in  Integer);
   --# global in out State;
   --# derives State from State, X;
   --# pre Sum_Is_Positive (State) and X >= 0;
   --# post Sum_Is_Positive (State)
   --#      and Add (State, State~,  X);

end Container.Calc;
```

```
package body Container.Calc
--# own State is Sum;
is
   Sum : Integer;

   procedure Initialize
   --# global out Sum;
   --# derives Sum from ;
   --# post Sum >= 0;
   is
   begin
      Sum := 0;
   end Initialize  ;

   procedure Inc (X : in  Integer)
   --# global in out Sum;
   --# derives Sum from Sum, X;
   --# pre Sum >= 0 and X >= 0;
   --# post Sum >= 0
   --#      and Sum = Sum~ + X;
   is
   begin
      Sum := Sum + X;
   end Inc;

begin
   --# hide Container.Calc;
   null;
end Container.Calc;
```

Figure 9: Example of a protected type after pre-processing

For all non-local variables in the entry barrier, SPARK requires the pre-processor to generate appropriate global and derives annotations. The post annotation of the procedure must also specify that the values of the non-local variables do not change. Otherwise the Examiner would lose track of the relations between the variables. Actually, this is a contradiction because on the one hand the post annotation specifies that this procedure will potentially modify variables to fulfill the entry barrier, and on the other hand it specifies that the variables are not modified. But it works insofar that the Examiner generates the right hypothesis for the verification conditions of

the entry procedure.

Figure 10 shows the example of Figure 5 pre-processed. The local procedure is named Barrier and is called before the first statement. The post-condition of procedure Barrier is the entry barrier Count $>$ 0. The information in the global and the derives annotations of procedure Barrier must be repeated in the entry's annotations. We only hint on the pre and post annotations of the entry procedure. They are handled as for the other procedures, i. e. the invariant is prepended.

```
procedure Get (X : out Element)
--# global in out Count;
--# derives Count from Count & X from Data, Count;
--# pre ...;
--# post ...;
is

   procedure Barrier
   --# global in out Count;
   --# derives Count from Count;
   --# post Count = Count~ and Count > 0;
   is
   --# hide Barrier;
   begin
      null;
   end Barrier;

begin
   Barrier;
   X := Data (Count);
   Count := Count - 1;
end Get;
```

Figure 10: Example of an entry procedure after pre-processing

4.4 Implementation of the Pre-Processor

We build the pre-processor on top of the ASIS (abbr. for Ada Semantic Interface Specification) library. ASIS is an interface to the Ada compiler that enables the pre-processor to access the abstract syntax tree directly from the compiler. The compiler is used at run time of the pre-processor to parse the source code of an Ada program. The ASIS library provides a skeleton tree traversal procedure with user-supplied operations to visit the nodes of the syntax tree in pre- and post-order.

To ease implementation and due to its temporary nature, we limit the pre-processor to the language features used by Coco. We do not implement any error checking. At least ASIS does check for source code errors and the Examiner will detect errors in the Spark annotations.

For character string handling, we use unbounded-length strings. These have the advantage of an automatic memory management which further eases the implementation.

In this section we describe the general workings of the pre-processor. The complete source code of the pre-processor is available online (cf. Appendix A.5).

4.4.1 Software Requirements

We use the following software to compile and run the pre-processor:

- The GNU Ada Compiler (GNAT) with ASIS-for-GNAT GPL 2005 Edition [ACT05]. The ASIS version always requires the same compiler version. Aside from ASIS, we use the GNAT hash table implementation and the GNAT utility gnatchop.
- The scanner generator aflex 1.4a-10 in a modified version for GNAT [ALT01].

4.4.2 Input and Output

The pre-processor expects the input file names as command line arguments in the same order as the Examiner. Different ASIS operations have to be executed for package specifications and bodies, the distinction is based on the file name suffix.

The pre-processor does not generate individual output files but prints all its output on console. This output must then be redirected to the program *gnatchop* which is part of the GNU Ada Compiler. It splits its input and creates individual and appropriately named source code files.

4.4.3 Tree Traversal

The pre-processor is based on the tree traversal procedure of ASIS. The tree nodes are called *elements* in ASIS terminology. For each element the action to perform is chosen by making a case distinction on the element kind.

The tree for the package specification example of Figure 2 looks like this:

```
> A_Declaration
>    A_Package_Declaration
—> A_Defining_Name
—>    A_Defining_Identifier . . . Calc
—> A_Declaration
—>    A_Procedure_Declaration
—>       An_Ordinary_Trait
——> A_Defining_Name
——>    A_Defining_Identifier . . . Inc
——> A_Declaration
——>    A_Parameter_Specification
——>       An_Ordinary_Trait
——>       An_In_Mode
———> A_Defining_Name
———>    A_Defining_Identifier . . . X
———> An_Expression
———>    An_Identifier ... Integer
```

This tree has been generated by the *display_source* program included in the ASIS distribution. When the tree is traversed the pre-processor has to save some context information. For example, when the element kind A_Defining_Name for the identifier Inc is met, then the pre-processor must know that it is a procedure name—the first A_Defining_Name after the A_Procedure_Declaration element is always the procedure name.

Context information is saved about the current package kind (specification head, specification or body), protected type kind (declaration head, declaration, body or none), declaration kind (type declaration or other), expression kind (infixed function call or other) and procedure kind (procedure head, procedure head with parameters, end of procedure head, end of procedure head with parameters or none). We use the term "head" for the declarative part before the token is.

We handle entries as special procedures and save information about the entry kind (entry head, end of entry head, entry body). Entry heads can be handled like procedures, but the barrier procedure must be inserted in their bodies. Not only must the barrier be saved verbatim, but also appropriate global, derives and post annotations must be generated.

Detecting the end of a procedure head is a bit complex because the procedure head ends if there are no further parameters but only if the procedure has parameters

at all. This explains the relatively large number of different values for the context information of procedures.

4.4.4 Source Code Retrieval

We save the line and column position of the already processed source code in the context information.

As soon as the parser recognizes an element, a source code span (i. e. the line and column numbers of the start and the end positions of the source code range) is associated with it. This does in general not include comments, closing brackets, semi-colons or tokens like is or begin.

ASIS provides an operation to retrieve the span of an element. It also provides an operation to retrieve the source code lines (including white space and newlines) from a given span. This span can also be enlarged. The drawback of this approach is that there is always an element required to retrieve any source code lines *and* the span of this element has to be completely in the requested (and possibly larger) span. This is inflexible, for example, it is hard to retrieve comments because comments are treated as white space and not as elements by ASIS. There are no other operations to retrieve source code lines. In particular, there is no operation to retrieve the source code lines of an arbitrary span.

We wrote a much more flexible procedure to retrieve the source code lines. This procedure retrieves source code lines from ASIS and trims them. It can retrieve lines up to and including the element position. It can retrieve the empty and commented lines up to the first non-comment line. It can retrieve empty and commented lines up to the position of the element and move the positional pointer over this element (the skip element). It can skip the next token or it can retrieve anything up to a specified token. The tokens mentioned are character strings and are different from the element; they are mainly used to skip tokens like is or begin.

Comments always precede an element (in the span from the last printed element to the current element), although we would need them succeed an element because SPARK annotations naturally succeed declarations in Ada. We solve this by calling the above procedure twice: in the first run it searches for comments and handles them, in the second run it handles the element itself.

Infixed function calls must be detected and skipped because the source code position is based on the infix operator. We would overrun the left-hand side of the function if we retrieved the source code for the operator element. When the right-hand side is processed, the source code of the operator is output as well because we always start our output from the last printed position.

All type declarations in the package specification are saved into a symbol hash table. Additionally all private variables of protected types are prefixed with the protected type name and saved in the symbol hash table. Maybe a hash table of hash tables would be a cleaner solution instead of the prefixing, but this is not possible with the GNAT hash table. The GNAT hash table is an abstract data object implemented as a package, there is no hash table type to create objects of.

When the private part of a protected type is processed, the output is saved in a temporary buffer. The package body requires an own annotation with the private variables and can only be output after the private part has been processed. The same applies to the package specification, which cannot be output before the abstract variable name of the state has been read. The temporary buffer is flushed as soon as possible.

In every package there could be more than one protected type. For each protected type an own output buffer is required. We use a hash table with the protected type name as hash key to access the right buffer. When an identifier is to be output, it is looked up in the symbol hash table and prefixed with the package name if necessary.

4.4.5 Handling of Annotations

Comments are neither elements in the syntax tree nor is their content parsed by the compiler. We tokenize them with a scanner generated by *aflex*. The end of a multi-line annotation is detected when an annotation starts with one of the SPARK keywords.

We do not parse the annotations but only tokenize them. It is sufficient to look up each token in the symbol hash table, if it is an identifier that needs to be prefixed by the package name. Due to the strict naming scheme of SPARK, which forbids the overloading of identifiers, this is a safe approach.

The invariant annotation is not output but saved in the context information. The inherit annotation is also saved. The name of the abstract state variable is extracted from the own annotation. The invariant is prepended to the post annotations and to the pre annotations if the procedure name is not Initialize.

4.5 Set Types in SPARK

The data types that are allowed in proof annotations exclude sets. We require set types in the proof annotations of COCO. The Examiner does not support any set

types, but the Simplifier and the Checker do provide set types as a legacy. We want to revive them for PassauSPARK.

First we declare a proof type for sets:

```
--# type Set_Type is abstract;
```

for which the Examiner generates the following FDL representation:

```
type set_type = pending;
```

At this point the type Set_Type is abstract. After the Examiner has been run, the generated FDL files must be modified and the declaration of set_type must be replaced by a concrete definition. For COCO we need only one type of sets, namely integer sets. The FDL replacement is:

```
type set_type = set of integer;
```

The pre-processor cannot perform this replacement because it is run before the Examiner. This post-processing to the pre-processor can be performed by some simpler tools, e. g. the stream editor *sed*.[5]

The set predicates provided by the Simplifier and the Checker are:

Predicate	Description
(set [ ])	empty set
(set [E])	singleton set containing only E
A \/ B	set union of set A and set B
A /\ B	set intersect of set A and set B
A \ B	set difference of set A and set B
E in A	E is element of set A
E not_in A	E is not element of set A

The Examiner forbids the occurrence of these predicates in any proof annotations. For this reason we have to declare some wrapper proof functions. We only declare wrappers for the predicates that are required for COCO:

```
--# function Set_Member (S : in Set_Type; E : in Integer) return Boolean;
--# function Set_Delete (S : in Set_Type; E : in Integer)  return Set_Type;
--# function Set_Insert (S : in Set_Type; E :  in  Integer)  return Set_Type;
--# function Set_Element (E : in Integer) return Set_Type;
--# function Empty_Set return Set_Type;
```

[5]Exemplary usage for GNU sed: sed -i "s/set_type = pending/set_type = set of integer/" file.fdl

The function Set_Member returns True if an element E is a member of the set S. The function Set_Delete deletes the element E from the set S and returns the resulting set. The function Set_Insert inserts the the element E into the set S and returns the resulting set. The function Set_Element returns a set which only contains the element E. The function Empty_Set returns the empty set.

In the implementation of Coco, we replace the general Integer type of E by an integer sub-range. Thus, the range overflow check of the Examiner will be stronger.

The semantics of the proof functions is defined by proof rules. The proof functions are just wrappers for the set predicates of the Simplifier and the Checker:

```
rule_family setfun:
        set_insert(S,E) requires [ S:any, E:i ] &
        set_delete(S,E) requires [ S:any, E:i ] &
        set_member(S,E) requires [ S:any, E:i ] &
        set_element(E) requires [ E:i ] &
        empty_set requires [ ].

setfun(1): set_insert(S,E) may_be_replaced_by S \/ (set [E]).
setfun(2): set_delete(S,E) may_be_replaced_by S \ (set [E]).
setfun(3): set_member(S,E) may_be_replaced_by E in S.
setfun(4): set_element(E) may_be_replaced_by (set [E]).
setfun(5): empty_set may_be_replaced_by (set []).
```

A remark is due about the for_all and for_some predicates. Their syntax is:

```
for_all(VAR : TYPE, P)
for_some(VAR : TYPE, P)
```

The Checker's user manual [Spa04b] does not specify the valid data types for TYPE. Apparently the set type is not a valid type. But, because our set types are sets of integers, we are able to use the following workarounds:

```
for_all(VAR : integer, VAR in SET -> P)
for_some(VAR : integer, VAR in SET -> P)
```

Variable VAR is an integer variable and, if it is a member of the set SET, the predicate P must hold by an implication.

Chapter 5

Design and Implementation of COCO

This chapter discusses the design and the implementation of Coco, for which we use PassauSpark. We explain our design decisions and show the corresponding source code declarations. These declarations are also used as FDL identifiers in the proof of Coco in the next chapter. The complete source code of Coco is shown in Appendix A.1.

Our design is much simpler than the original Coco design [Len77], mainly because we do not implement error handling, terminal handling and the administrator console. These are not required by the Coco specification. We point out where our design differs from the original design.

The implementation constants are adjustable in package Conf. Only when the number of tasks is changed, must this be adjusted in the main procedure as well.

5.1 Packet Format

The packet header contains an address field with the receiver address, the priority value and the length of the data portion.

We implement Coco as an eight-port switch. The number of ports is adjustable. The ports and their addresses are numbered from one to eight. The packet priority is a value between zero (highest) and 63 (lowest). The data consists of an array of bytes. The data size is variable and is given by the data length value. It must not exceed a maximum of 1500 bytes.

A packet can be directly mapped on a fixed-sized memory frame. As there are no pointers or memory references in Spark, we index the memory frames from one to the maximum frame count of 65536. We implement the memory frame format as follows:

```
subtype Port_Adr is Positive range 1 .. Conf.Num_Ports;

subtype Packet_Priority is Natural range 0 .. Conf.Min_Prio;

subtype Data_Length is Natural range 0 .. Conf.Max_Length;

subtype Data_Index is Positive range 1 .. Conf.Max_Length;

type Packet_Data is array (Data_Index) of Byte;

type Packet is record
   Dest : Port_Adr;
   Prio : Packet_Priority;
   Length : Data_Length;
   Data : Packet_Data;
end record;
```

5.2 Memory Access Control

The access to the shared buffer must be handled in a critical region. The protected type Buffer_Guard manages the insertion and deletion of buffer elements. It provides the following operations:

```
entry Request (Element_Index : out Buffer_Index);

procedure Release (Element_Index : in Buffer_Index);
```

where:

```
subtype Buffer_Index is Positive range 1 .. Conf.Buffer_Size;
```

With the entry Request, a free memory frame can be reserved. The entry returns the index of a memory frame which is ready to be filled. If there are currently no empty frames, this request will be suspended until another frame becomes empty.

With the procedure Release, a used memory frame can be freed again.

The private data of the Buffer_Guard consists of the array of free memory frames Free_List, which is managed as a stack, and the object Stack, which contains the stack pointer and its operations.

```
Free_List : Buffer_Index_Array;
Stack : Stacks.Stack;
```

where:

```
type Buffer_Index_Array is array (Buffer_Index) of Buffer_Index;
```

The Stack class in package Stacks provides the following procedures to manipulate the stack pointer:

```
procedure Push (S : in out Stack; P : out Integer);

procedure Pop (S : in out Stack; P : out Integer);
```

These procedures push respectively pop an element to and from the stack. Both return the pointer position P. The object data consists of the current pointer position and the maximum stack size:

```
subtype Stack_Range is Positive range 1 .. Positive'Last − 1;

type Stack is record
   Pointer : Positive;
   Max : Stack_Range;
end record;
```

The stack pointer is initially 1 (i. e. the stack is empty) and points always to the successor element of the top element. Consequently, the stack is full when the pointer has the value Max + 1.

The maximum stack size is initialized with the following procedure. After the initialization the stack is full.

```
procedure Initialize (S : out Stack; Max : in Stack_Range);
```

The state of the stack can be queried with the following functions:

```
function Empty (S : in Stack) return Boolean;

function Full (S : in Stack) return Boolean;
```

5.3 Ordering

The ordering structure is implemented as a priority queue. The priority queue is accessed simultaneously by different tasks, so we implement it as a protected type.

The priority ordering induces a problem: low priority packets could be delayed forever in favor of more urgent packets. To prevent this starvation of packets within the system, packets must age. When a packet is entered into a priority queue, it is assigned a fixed system priority. The system priority is obtained by adding a dynamic priority in form of a time-stamp to the packet priority. The dynamic priority value is made available and periodically incremented by the system. The packet priority can only take on a limited number of discrete values. Packets with a long residence time are eventually forced to leave the system because their priority will supersede the priority of any other packet residing in the system.

This prevention of starvation is especially efficient as the aging of packets can be implemented with the modification of a single value—the dynamic priority—without the need to change all packets in the system. Nevertheless, we do not prove the absence of starvation in the Coco system, as this would require to include all tasks. Our proof—just like the original proof of Coco—focuses on protected types.

The protected type Priority_List provides the following operations:

```
procedure Enter (Element_Index : in Buffer_Index; Priority : in System_Priority);

entry Remove (Element_Index : out Buffer_Index);
```

With the procedure Enter a memory frame index can be inserted into the priority queue. The parameter Priority contains the system priority, which is the sum of the packet priority and the dynamic priority.

With the entry Remove the index of the memory frame with the highest priority can be requested. This index is then removed from the priority queue. If the queue is empty, this request will be suspended until there is an index available.

The private data of Priority_List consists of one variable:

```
List : Priority_Queue;
```

The priority queue is implemented by an array which is accessed like a linked list:

```
Ext : constant := 0;

subtype System_Priority is Long_Long_Integer range 0 .. Long_Long_Integer'Last;

subtype List_Index is Natural range Ext .. Conf.Buffer_Size;

type Priority_Element is record
   Pri : System_Priority;
   Suc : List_Index;
end record;

type Priority_Queue is array (List_Index) of Priority_Element;
```

The elements of the linked list are the system priority values. The linked list is extended by one element Ext. Figure 11 illustrates this. The first element of the array is List (Ext). The upper line of Figure 11 shows the array elements.

The first element of the list is List (Ext).Suc. The successor of the first list element is List (List (Ext).Suc).Suc and so on. The successor of the last list element is Ext.

With exception of Ext, all the indices of the list are valid indices of memory frames.

If the list is empty then List (Ext) = Ext. Because List (Ext) has the highest possible priority value, the list is ordered, starting from the first list element.

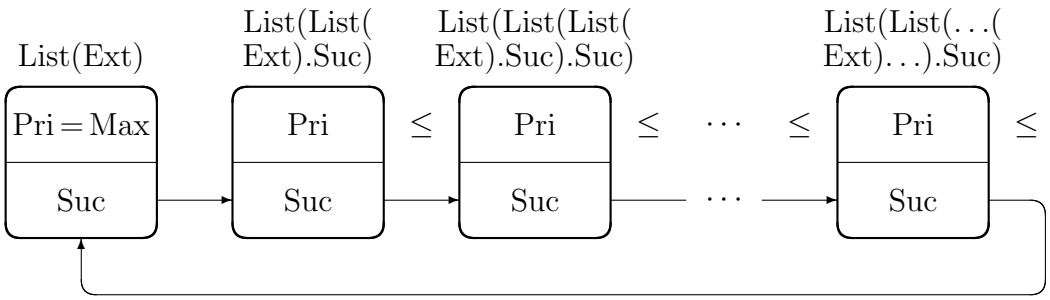

Figure 11: Priority queue

To enable an efficient removal operation the element with the highest priority is the head of the list. To enable an efficient insertion operation only relevant parts of the list are traversed. This avoids unnecessary scanning.

As an aside, in the original Coco system the ordering structure was implemented as a single monitor with several sub-lists. This design was imposed by Concurrent Pascal which only provides monitor objects, not monitor types. We choose to implement it as one list and multiple protected objects (of type Priority_List). This enables more concurrency.

5.4 Transfer Tasks

We implement one data handling task and several input and output tasks. One input and one output task speak to one port.

It would be unwise to decompose the data handling task further into several tasks. Although the packets could be handled independently, i e. in parallel, the decomposed tasks would only run in pseudo-parallel mode. They would compete for the same single processor and introduce overhead with no gain.

The data handling task is of type Data_Process. We only implement the basic structure of this task without any real data handling. The Data_Process removes one index from the input Priority_List, handles the packet, and puts the memory index into the output Priority_List of the packet's destination port.

The tasks of type Input_Process are parametrized with a port address. Each task listens to its port for incoming packets. When a packet arrives, the input task reserves a memory frame at the Buffer_Guard. It then copies the incoming packet data into the memory frame. When the packet has been received completely, the index of this memory frame is inserted into the input Priority_List.

The tasks of type Output_Process are also parametrized with a port address. Each task takes a memory frame index from the output Priority_List associated with its port and writes the packet to the network. When the packet has been sent, the memory frame is freed at the Buffer_Guard.

The real input/output handling of the Input_Process and Output_Process is not implemented. This is beyond the scope of SPARK and would be hidden in a hide annotation anyway.

In contrast to the original COCO design, we did not displace the task logic into a separate class. Instead, each task calls the operations of the Buffer_Guard and the Priority_List directly.

The implementation differs from the specification in one point: the run-time system chooses the output task non-deterministically and not always the output task with the most urgent packet in the queue. The same problem arose in Concurrent Pascal.

5.5 Memory Access

The packets are stored in memory frames. We implement the buffer with the memory frames as shared array:

```
type Packet_Array is array (Buffer_Index) of Packet;

Shared_Buffer : Packet_Array;
```

A shared variable is potentially dangerous because the concurrent access may lead to lost updates. In our case, there is concurrent access to the array but no concurrent access to the same array element. Before the input and the output tasks can insert or remove elements, they have to register the array index at the Buffer_Guard. When the data handling task accesses the array, it gets the array index from the input priority list. The input task inserts the index into this priority list after it has finished its access to the array element. So the access to an array element is in every case protected by a critical region.

It would be interesting to prove that the access to the shared buffer is really safe but we can specify proof annotations for tasks neither in RavenSPARK nor in PassauSPARK.

The definitions of the packet type and the packet array type are not in the private part of the package specification because their internal structure is not private but shared.

There are no special operations provided to access the memory frames. The array can be accessed directly.

In the original Coco design, the memory buffer was implemented as a monitor because Concurrent Pascal does not allow any shared data types other than monitors. But there is no need to implement it as a monitor. We gain more concurrency and more performance if we implement it as a shared array.

5.6 Real-Time Control

The dynamic priority, which is used for the aging of packets, is implemented by one protected type and a timer task.

The protected type Dyn_Priority contains the dynamic priority value for the transfer tasks and provides the following operations:

```
procedure Increment;

function Count return System_Priority;
```

Procedure Increment increments the dynamic priority. Only the timer task may call this operation.

The function Count provides the current dynamic priority value. The initial value is zero.

The task of type Timer_Process calls procedure Increment regularly. Ada provides no unbounded integer type, so we use Long_Long_Integer, the largest available integer type, for the definition of System_Priority. With a frequency of 100 Hz and 2^{63} different priority values Coco is able to run for a long period of time.

5.7 System Structure

An access graph shows the access rights of tasks and protected objects [BH75]. Its nodes are abstract data objects, an arrow signifies that the object at its end has the right to access the object at its tip. The access graph of the Coco system is shown in Figure 12. It shows the access relations between the objects (P = protected object, T = task object, S = shared object) described in this chapter. The captions denote the type names. The line styles of the arrows have no function other than to make the figure more comprehensible.

Obviously the system structure is very flat. In particular, there are no monitor hierarchies. In a monitor hierarchy, there are nested calls to monitor operations which would almost unavoidably lead to deadlocks [Kee79]. The access graph of the original Coco system is more complex; it has 19 nodes and 6 levels. It also does not contain a monitor hierarchy.

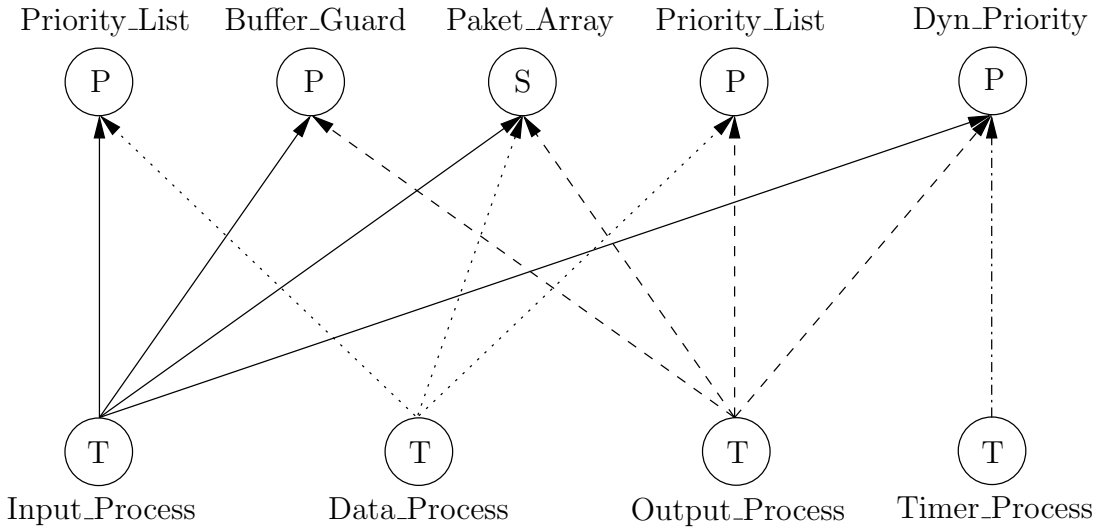

Figure 12: System structure

Chapter 6

Verification of COCO

We ran several systematic tests with packets generated at random points in time and with randomly generated port addresses. Consequently, we have good faith in the correctness of Coco. Nevertheless, we want to prove certain properties of Coco with the means of formal verification, which is based on logic and not on faith. Spark supports us in this matter.

A proof in Spark proceeds by the following scheme:

1. Declare proof functions.
2. Specify Spark annotations.

   The original Coco system has been verified manually in Hoare logic [Len77]. We adhere to the pre- and post-conditions of the original proof, even though they differ in detail: the stronger type system of Ada and the automatically inserted range checks of the Examiner make some assertions in the original proof unnecessary.

   When we define the proof annotations, we also show the source code of the procedures for clarification. In general, we omit the data flow dependency annotations **global** and **derives** in this chapter because they do not lead to further insight. The proof rules (including rule family headers) are shown in Appendix A.1.
3. Infer properties of proof functions.

   The properties of proof functions are determined with proof rules. The formal parameters of the proof functions are defined in Step 1. The proof rule family headers are then of minor interest, we omit the headers in this chapter. They are shown a second time in Appendix A.2 for reference.

   All proof rules are axiomatic, i. e. Spark does not impose a proof obligation for them.

4. Run the Examiner and the Simplifier.

5. Prove verification conditions with the Checker.

   Verification conditions are generated for each procedure. There is one proof session for every verification condition not proved by the Simplifier in Step 4.

Readers not interested in the proof sessions may skip the according subsections. They are not required for the understanding of the rest.

6.1 Stack Class

The stack is implemented as a class. There are no annotations in a class body, so we only need to handle the declaration part. Similarly to the example in Figure 3, we cannot directly access the class components in the annotations. The point is that we need to reference the value of the stack pointer and the maximum stack size in the proof of the Buffer_Guard and, therefore, we need to export these values in the stack annotations. We define the following proof functions:

```
--# function Ptr (S : in Stack) return Integer;

--# function Max_Ptr (S : in Stack) return Integer;
```

These proof functions enable us to reference the class components Pointer and Max. We named the functions differently from the component names because the naming convention of SPARK requires it. Classes in Ada are records. As described in the table of Section 3.3, the stack record components Pointer and Max can be addressed with the predicates fld_pointer and fld_max. The corresponding proof rules are straightforward:

```
stacks(1): ptr(S) may_be_replaced_by fld_pointer(S).

stacks(2): max_ptr(S) may_be_replaced_by fld_max(S).
```

The uppercase S is a Prolog variable for the stack record.

We define another proof function which is used like an invariant of the class:

```
--# function Ptr_In_Range (S : in Stack) return Boolean;
```

This proof function Ptr_In_Range returns true if the stack pointer is in the range between the first value and the maximum value plus one:

```
stacks(3): ptr_in_range(S) may_be_replaced_by
           fld_pointer(S) >= 1 and fld_pointer(S) <= fld_max(S) + 1.
```

With these proof functions we can define the pre- and post-conditions of the stack operations as follows:

```
procedure Initialize (S : out Stack; Max : in Stack_Range);
--# post Ptr_In_Range (S) and Max_Ptr (S) = Max and Ptr (S) = Max + 1;

procedure Push (S : in out Stack; P : out Integer);
--# pre Ptr_In_Range (S) and not Full (S);
--# post Ptr_In_Range (S) and Max_Ptr (S) = Max_Ptr (S~) and
--#      Ptr (S) = Ptr (S~) + 1 and P = Ptr (S~);

procedure Pop (S : in out Stack; P : out Integer);
--# pre Ptr_In_Range (S) and not Empty (S);
--# post Ptr_In_Range (S) and Max_Ptr (S) = Max_Ptr (S~) and
--#      Ptr (S) = Ptr (S~) - 1 and P = Ptr (S);
```

During the initialization the maximum value is assigned. The pointer has the maximum value plus one; in other words, the stack is full.

Before the Push operation the stack pointer has to be in range and the stack must not be full. After the Push operation the stack pointer still has to be in range, the maximum stack value must not have changed and the stack pointer must be incremented. The returned value of P has the value of the new stack pointer.

The pre- and post-conditions of the procedure Pop are similar to the procedure Push. Only the stack pointer is decremented and the returned value of P is the value of the stack pointer before the decrement.

The proof functions Empty and Full are implicitly defined by their concrete functions:

```
function Empty (S : in Stack) return Boolean;
--# return Empty (S);

function Full (S : in Stack) return Boolean;
--# return Full (S);
```

These implicitly defined proof functions have the same semantics as the following concrete functions:

```
stacks(4): empty(S) may_be_replaced_by fld_pointer(S) = 1.

stacks(5): full(S) may_be_replaced_by fld_pointer(S) = fld_max(S) + 1.
```

When the annotated source code of the stack class is processed by the Examiner and the resulting verification conditions and the proof rules of the rule family stacks are run through the Simplifier, all verification conditions are simplified to true. We do not have to invoke the Checker.

6.2 Protected Type Buffer_Guard

6.2.1 Annotations in the Body

The protected type Buffer_Guard manages a list of free frames. The indices of the frames are stored in an array. The occupied array elements are managed by a stack. We want to abstract from the array and the stack and define the properties of the Buffer_Guard operations with the means of a set of free frames instead. Let F be this set of free frames:

$$F := \{j \mid \exists\, i : 1 \leq i < \text{Stack.Pointer} : \text{Free\_List}(i) = j\}$$

The stack pointer points always to the successor element of the top of the stack, so a less-than-equal sign is required. Unfortunately it is not possible to declare a set like this directly in SPARK. But we can declare it as proof function. The parameters of the set are the stack pointer and the array Free_List. The return type is a set of integers:

```
--# function F (Free_List : in Buffer_Index_Array;
--#             Pointer:  in  Stacks.Stack_Range) return Set_Type;
```

With this proof function we are able to define the pre- and post-conditions of the Buffer_Guard operations:

```
entry Request (Element_Index : out Buffer_Index) when not Stacks.Empty (Stack)
--# post Set_Member (F (Free_List, Stacks.Ptr (Stack~)), Element_Index)
--#      and F (Free_List,  Stacks.Ptr (Stack))
--#          = Set_Delete (F (Free_List,  Stacks.Ptr (Stack~)),  Element_Index);
is
   Top :  Buffer_Index;
begin
   Stacks.Pop (Stack, Top);
   Element_Index := Free_List (Top);
end Request;

procedure Release (Element_Index : in Buffer_Index)
--# pre not Set_Member (F (Free_List, Stacks.Ptr (Stack)), Element_Index)
--#     and not Stacks.Full (Stack);
--# post F (Free_List, Stacks.Ptr (Stack))
--#      = Set_Insert  (F (Free_List~,  Stacks.Ptr (Stack~)),  Element_Index);
is
   Top :  Buffer_Index;
begin
   Stacks.Push (Stack, Top);
   Free_List  (Top) :=  Element_Index;
end Release;
```

The entry Request removes one element from the set of free frames. There is no pre annotation because the entry barrier not Stacks.Empty (Stack) will be satisfied when the entry body is executed. The post-condition states that the element was a member of the set of free frames and is no more in the set.

The procedure Release adds a frame to the set of free frames. The frame must not be in the set already. The pre-condition not Stacks.Full (Stack) is the equivalent to the barrier of the entry Request.

The initialization procedure assigns to each element of array Free_List its index value, i. e. the array of free frames corresponds to the entire array of the shared buffer. There is no pre-condition. The post-condition could be $F = \{1, \ldots, \text{Stack.Pointer} - 1\}$. It would be tedious to write this static list of elements down in the SPARK annotations. For the procedure Initialize we forgo F and make a direct statement about array Free_List:

```
procedure Initialize
--# post (for all J in Buffer_Index => (Free_List (J) = J));
is
begin
   for I in Buffer_Index
   --# assert (for all J in Buffer_Index range 1 .. I - 1 => (Free_List (J) = J));
   loop
      Free_List (I) := I;
   end loop;
   Stacks.Initialize (Stack, Conf.Buffer_Size);
end Initialize;
```

The universal quantifier expresses that each element of array Free_List is assigned its index value. The loop invariant is trivial to find: for each element of the array counted from 1 up to I−1, its index value must have been assigned.

We have to define the proof function F with the help of proof rules. We cannot do this directly because the expressive power of the rule language is not sufficient. But we can infer properties of F:

```
buffer_guard(1): f(F,P) \/ (set [E]) may_be_replaced_by f(update(F,[P],E),P+1)
                 if [ E not_in f(F,P) ].

buffer_guard(2): f(F,P) \ (set [element(F,[P−1])]) may_be_replaced_by f(F,P−1)
                 if [ element(F,[P−1]) in f(F,P) ].
```

Of course we only infer properties we really need for the proof. In the proof rules, the FDL identifier f is used for the proof function name F. F and P are the parameters for the Free_List and the stack pointer. These two rules define the insertion and the removal of one element. When an element E is inserted into the set, provided that it is not already in the set, the array is updated and the stack pointer is incremented. The removal works conversely.

Further on we define a rule for set membership:

```
buffer_guard(3): element(F,[J]) in f(F,P) may_be_deduced_from [ J>=1, J<P ].
```

An element of the array Free_List is not necessarily an element of the set F. It is an element of F if and only if its index is in the range between one and the stack pointer minus one.

The invariant of the protected type Buffer_Guard is dictated by the internal stack object:

```
--# invariant Stacks.Ptr_In_Range (Stack) and Stacks.Max_Ptr (Stack) = Conf.Buffer_Size;
```

The rule stacks(3) cannot be applied to resolve Ptr_In_Range, because this time it is prefixed by the package name Stacks. We need a new rule with the same semantics but with a package prefix:

```
stacks(6): stacks__ptr_in_range(S) may_be_replaced_by
           stacks__ptr(S) >= 1 and stacks__ptr(S) <= stacks__max_ptr(S) + 1.
```

Rules for the identifiers stacks__ptr and stacks__max_ptr are not required. But we do need rules for the functions Stacks.Empty and Stacks.Full, which are used in the barrier of entry Release and the pre annotation of procedure Request. They are prefixed by the package name but otherwise have the same semantics as the rules stacks(4) and stacks(5):

```
stacks(7): stacks__empty(S) may_be_replaced_by stacks__ptr(S) = 1.

stacks(8): stacks__full(S) may_be_replaced_by stacks__ptr(S) = stacks__max_ptr(S) + 1.
```

6.2.2 Annotations in the Declaration

This is the complete annotated source code of the declaration part; the explanations follow below:

```
protected type Buffer_Guard
--# own State : State_Type;
--# invariant Buffer_Guard_Invariant (State);
is
   --# type State_Type is abstract;

   --# type Set_Type is abstract;
   --# function Set_Member (S : in Set_Type; E : in Buffer_Index) return Boolean;
   --# function Set_Delete (S : in Set_Type; E : in Buffer_Index) return Set_Type;
   --# function Set_Insert (S : in Set_Type; E : in Buffer_Index) return Set_Type;
   --# function Empty_Set return Set_Type;
```

```
   --# function F (Free_List : in Buffer_Index_Array;
   --#               Pointer:  in  Stacks.Stack_Range) return Set_Type;

   --# function Buffer_Guard_Invariant (S : in State_Type) return Boolean;
   --# function Can_Be_Released (S : in State_Type; E : in Buffer_Index) return Boolean;

   procedure Initialize ;
   --# global in out State;
   --# derives State from State;

   entry Request (Element_Index : out Buffer_Index);
   --# global in out State;
   --# derives Element_Index, State from State;

   procedure Release (Element_Index : in Buffer_Index);
   --# global in out State;
   --# derives State from Element_Index, State;
   --# pre Can_Be_Released (State, Element_Index);

private
   Free_List  :  Buffer_Index_Array;
   Stack :  Stacks.Stack;
end Buffer_Guard;
```

The state variable is simply named State and has the proof type State_Type.

The invariant has the same semantics as the invariant in the protected type body. It must be parametrized by the state variable and not by the—at this point invisible—Free_List and Stack. The pre-processor transforms the protected type into a package. As shown in Section 3.3, the state_type for packages is an FDL record. The record components can be accessed by the prefix fld_. The invariant can then be defined as follows:

```
buffer_guard(4): buffer_guard_invariant(S) may_be_replaced_by
                 stacks__ptr_in_range(fld_stack(S))
                 and stacks__max_ptr(fld_stack(S)) = conf__buffer_size.
```

The declaration part must also contain the declarations of all the used proof functions. This includes the proof function F as well as the set proof functions of Section 4.5.

One note about the global and derives annotations of the procedure Initialize: they falsely imply that the initial state, i. e. the initial values of the variables Free_List or Stack are used. The initial values are not used because all array elements are overwritten in procedure Initialize, but the Examiner is not able to recognize this.

As we do not make any assumptions about the properties of tasks—and the tasks are the only objects calling the protected type operations—we limit the pre- and

post-conditions in the declaration part to the bare necessities. This means: no post-conditions except the invariant. The pre-conditions are dictated by the refinement of the body annotations.

Procedure Initialize has by definition no pre-condition and entry Request does not need one because of its barrier condition. Only procedure Release requires a proof function which has the same semantics as the pre-condition of procedure Release in the body:

```
buffer_guard(5): can_be_released(S,E) may_be_replaced_by
                 E not_in f(fld_free_list(S),stacks__ptr(fld_stack(S)))
                 and not stacks__full(fld_stack(S)).
```

6.2.3 Proof Sessions for Entry Request

The Simplifier is not able to prove all verification conditions. We have to revert to the Checker.

In every proof session we first list the hypotheses and conclusions of the verification condition. We omit superfluous hypotheses. On the one hand this leads to holes in the numbering of hypotheses, on the other hand it increases clarity considerably.

The Checker produces a lot of output. We reduce the amount of information by skipping superfluous lines. The displayed output lines of the Checker have not been modified, even the line breaks are verbatim.

The complete output of the Checker and the input command logs are available online (cf. Appendix A.5).

We mark the Checker input and output typographically by a vertical bar on the left-hand side. This way, the Checker protocol and our comments are easy to distinguish.

The command prompt of the Checker is |: (a vertical bar followed by a colon). We highlight the command keywords with bold face.

Proof Session from Start to Finish

We start with the proof session that spans the source code range from the pre-condition to the post-condition of entry Request. The verification condition consists of the following hypotheses and conclusions:

```
CHECK|: list.
H4: stacks__ptr(stack__2) = stacks__ptr(stack) − 1
H5: stacks__ptr(stack__2) >= 1
−−>
  C1: element(free_list, [stacks__ptr(stack__2)]) in f(free_list,
        stacks__ptr(stack))
  C2: f(free_list, stacks__ptr(stack__2)) = f(free_list, stacks__ptr(stack))
        \ (set [element(free_list, [stacks__ptr(stack__2)])])
```

We can infer C1 from buffer_guard(3), provided that the necessary conditions to apply buffer_guard(3) are satisfied, namely

1. stacks__ptr(stack__2) >= 1

2. stacks__ptr(stack__2) < stacks__ptr(stack)

They are indeed satisfied by the hypotheses H5 and H4.

```
CHECK|: infer c#1 using buffer_guard(3) from [4,5].
*** New H12: element(free_list, [stacks__ptr(stack__2)]) in f(free_list,
        stacks__ptr(stack))
*** PROVED C1: element(free_list, [stacks__ptr(stack__2)]) in f(free_list,
        stacks__ptr(stack))
```

Hypotheses and conclusions on which the command is to be applied are referenced by an h or c, followed by a diamond and their number.

The hypotheses which are conditions for the application of a proof rule can be referenced by their numbers in squared brackets. This is not always required, especially if the Checker is able to find the hypotheses by itself. We always state in the explanations which hypotheses are used during the application of a proof rule.

The proved conclusion C1 is added as new hypothesis H12. Next we want to resolve C2 using buffer_guard(2). The necessary condition for the application of buffer_guard(2) is:

```
element(free_list,[stacks__ptr(stack)−1]) in f(free_list,stacks__ptr(stack))
```

H12 is already similar to this condition. We prepare H12 and C2 by replacing the predicate stacks__ptr(stack__2) by stacks__ptr(stack) − 1.

In contrast to command infer, command replace does not get a list of used hypothesis as parameter. The replacement of H12 is justified by the information from hypothesis H4.

```
CHECK|: replace h#12 & c#2: stacks__ptr(stack__2) by stacks__ptr(stack) − 1 using eq(1).
*** New H12: element(free_list, [stacks__ptr(stack) − 1]) in f(free_list,
        stacks__ptr(stack))
>>> New goal C2: f(free_list, stacks__ptr(stack) − 1) = f(free_list,
        stacks__ptr(stack)) \ (set [element(free_list, [stacks__ptr(stack)
        − 1])])
```

The new goal C2 replaces the existing conclusion C2 and the new hypothesis H12 the existing hypothesis H12. The built-in rule eq(1) allows equality substitutions. We always show the definition of the built-in rules we use. All built-in rule definitions are copied from the Checker Rules Manual [Spa04a]. Rule eq(1) is defined with a short-cut to the Prolog system:

```
eq(1): X may_be_replaced_by Y if [ X=Y, goal(X\=Y) ].
```

After the modification of H12 and C2 we can apply buffer_guard(2) to C2.

```
CHECK|: replace c#2: f(F,P) \ (set [element(F,[P−1])]) by f(F,P−1) using buffer_guard(2).
NEW EXPRESSION: f(free_list, stacks__ptr(stack) − 1) = f(free_list,
        stacks__ptr(stack) − 1)
```

The NEW EXPRESSION replaces the existing conclusion C2. The resulting equation in C2 is trivially true. We let the inference engine of the Checker solve it.

```
CHECK|: done.
*** PROVED C2: f(free_list, stacks__ptr(stack) − 1) = f(free_list,
        stacks__ptr(stack) − 1)
*** VC PROVED −− Well done!
```

There are no conclusions left. Therefore this verification condition (abbr. VC) is proved.

Proof Session for Refinement Integrity

Regarding the pre-condition of entry Request, the invariant in the body part must follow from the invariant in the declaration part. The resulting verification condition is listed below.

```
CHECK|: list.
H3: stacks__max_ptr(fld_stack(state)) = conf__buffer_size
−−>
  C1: stacks__max_ptr(fld_stack(state)) = 65536
```

Obviously the Simplifier has a problem to recognize that conf__buffer_size = 65536. The rule which resolves this is automatically generated by the Examiner.

```
request_rules(3): conf__buffer_size may_be_replaced_by 65536.
```

We apply this rule by hand.

```
CHECK|: replace h#3: conf__buffer_size by 65536 using request_rules(3).
NEW EXPRESSION: stacks__max_ptr(fld_stack(state)) = 65536
```

The existing hypothesis H3 is replaced by this new expression. H3 is now equal to C1. The inference engine of the Checker is able to prove C1.

```
CHECK|: done.
*** PROVED C1: stacks__max_ptr(fld_stack(state)) = 65536
*** VC PROVED -- Well done!
```

There is a second verification condition; this time the hypothesis and the conclusion are exchanged. It stems from the refinement of the post-condition: the invariant in the declaration part must follow from the invariant in the body part. We prove it in the same way but do not show the proof here.

6.2.4 Proof Sessions for Procedure Release

The following proof session has some resemblance to the proof session of entry Request. It makes no difference whether a hypothesis comes from an entry barrier or from a pre annotation.

Proof Session from Start to Finish:

The following verification condition must be proved:

```
CHECK|: list.
H2: not element_index in f(free_list, stacks__ptr(stack))
H8: stacks__ptr(stack__1) = stacks__ptr(stack) + 1
-->
  C1: f(update(free_list, [stacks__ptr(stack)], element_index), stacks__ptr(
        stack__1)) = f(free_list, stacks__ptr(stack)) \/ (set [element_index]
        )
```

We prove C1 with proof rule buffer_guard(1). The following condition must be satisfied before this rule can be applied:

```
element_index not_in f(free_list, stacks__ptr(stack))
```

H2 is already very similar to this condition. We prepare H2 by replacing the not ... in by not_in. This is justified by the built-in rule sets(2) which is defined as follows:

sets(2): (X not_in A) & (not (X in A)) **are_interchangeable**.

```
CHECK|: replace h#2: not (X in A) by X not_in A using sets(2).
NEW EXPRESSION: element_index not_in f(free_list, stacks__ptr(stack))
```

The modified hypothesis H2 allows us to apply rule buffer_guard(1).

```
CHECK|: replace c#1: f(F,P) \/ (set [E]) by f(update(F,[P],E),P+1) using buffer_guard(1).
NEW EXPRESSION: f(update(free_list, [stacks__ptr(stack)], element_index),
        stacks__ptr(stack__1)) = f(update(free_list, [stacks__ptr(stack)],
        element_index), stacks__ptr(stack) + 1)
```

We make both sides of the equation equal by replacing stacks__ptr(stack__1) using the information in hypothesis H8.

```
CHECK|: replace c#1: stacks__ptr(stack__1) by stacks__ptr(stack) + 1 using eq(1).
NEW EXPRESSION: f(update(free_list, [stacks__ptr(stack)], element_index),
        stacks__ptr(stack) + 1) = f(update(free_list, [stacks__ptr(stack)],
        element_index), stacks__ptr(stack) + 1)
```

The resulting equation is trivially true.

```
CHECK|: done.
*** PROVED C1: f(update(free_list, [stacks__ptr(stack)], element_index),
        stacks__ptr(stack) + 1) = f(update(free_list, [stacks__ptr(stack)],
        element_index), stacks__ptr(stack) + 1)
*** VC PROVED -- Well done!
```

Proof Session for Refinement Integrity

```
CHECK|: list.
H5: stacks__max_ptr(fld_stack(state)) = conf__buffer_size
H7: element_index not_in f(fld_free_list(state), stacks__ptr(fld_stack(state
        )))
-->
  C1: stacks__max_ptr(fld_stack(state)) = 65536
  C2: not element_index in f(fld_free_list(state), stacks__ptr(fld_stack(
        state)))
```

C1 is easy to prove by replacing conf__buffer_size with 65536. The rule which allows this is automatically generated by the Examiner:

release_rules(3): conf__buffer_size **may_be_replaced_by** 65536.

```
CHECK|: replace h#5: conf__buffer_size by 65536 using release_rules(3).
NEW EXPRESSION: stacks__max_ptr(fld_stack(state)) = 65536
CHECK|: done.
*** PROVED C1: stacks__max_ptr(fld_stack(state)) = 65536
```

C2 can be proved by adjusting it to hypothesis H7. We replace the not ... in by not_in.

```
CHECK|: replace c#2: not (X in A) by X not_in A using sets(2).
NEW EXPRESSION: element_index not_in f(fld_free_list(state), stacks__ptr(
        fld_stack(state)))
```

Conclusion C2 and hypothesis H7 are now identical.

```
CHECK|: done.
*** PROVED C2: element_index not_in f(fld_free_list(state), stacks__ptr(
        fld_stack(state)))
*** VC PROVED -- Well done!
```

There is one other verification condition which is equally simple to prove as C1. We do not show the proof here.

6.2.5 Proof Sessions for Procedure Initialize

Proof Session from Start to the Loop Invariant

The following verification condition spans the source code from the pre-condition to the start of the loop. For the proof, we do not need any hypotheses; the conclusion is always true.

```
CHECK|: list.
  C1: for_all(j_ : integer, 1 <= j_ and j_ <= 0 -> element(free_list,
        [j_]) = j_)
```

Inside the quantifier, the left side of the implication is always false, the implication, therefore, true. The universal quantifier can be eliminated with command unwrap.

```
CHECK|: unwrap c#1.
>>> New goal C1: 1 <= int_j__1 and int_j__1 <= 0 -> element(free_list,
        [int_j__1]) = int_j__1
```

We have difficulties to infer that the left side of the implication is false. We cannot make a definitive statement here, but it seems that, if the Checker is able to simplify the result of an inference to false, then it aborts with the error message “FAIL”. But there is another way to prove C1. First we infer a hypothesis.

```
CHECK|: infer 0 < int_j__1 or not 0 < int_j__1 using inference(1).
*** New H2: 0 < int_j__1 or not 0 < int_j__1
```

The rule inference(1) is defined as follows:

```
inference(1): X may_be_deduced_from [ X ].
```

Then we start a proof by cases on hypothesis H2. There are two cases: 0 < int_j__1 and its negation.

```
CHECK|: prove c#1 by cases on h#2.
CASE 1: 0 < int_j__1
        *** New H3: 0 < int_j__1
        >>> New goal C1: 1 <= int_j__1 and int_j__1 <= 0 -> element(free_list,
                [int_j__1]) = int_j__1
```

On command prove the Checker enters a new proof frame. The case is added as a new hypothesis. When the proof frame is exited, this hypothesis and all other hypotheses which were inferred within that proof frame are deleted. In the new proof frame there is only one conclusion to prove. In this case, the conclusion is identical to the conclusion of the outer proof frame. We mark the new proof frame typographically with an indentation.

The hypothesis 0 < int_j__1 is sufficient to prove the implication.

```
        CHECK|: done.
        *** PROVED C1: 1 <= int_j__1 and int_j__1 <= 0 -> element(free_list, [
                int_j__1]) = int_j__1 FOR CASE 1
```

The proof frame is exited, the second case is added as a new hypothesis and the conclusion has to be proved again.

```
CASE 2: not 0 < int_j__1
        *** New H3: not 0 < int_j__1
        >>> New goal C1: 1 <= int_j__1 and int_j__1 <= 0 -> element(free_list,
                [int_j__1]) = int_j__1
```

This single hypothesis is also sufficient to prove the implication.

```
        CHECK|: done.
        *** PROVED C1: 1 <= int_j__1 and int_j__1 <= 0 -> element(free_list, [
                int_j__1]) = int_j__1 FOR CASE 2
*** VC PROVED -- Well done!
```

The verification condition has been proved. Without the case distinction, the inference engine would not be able to prove the implication.

Proof Session for the Loop Invariant

This verification condition handles the transition from one loop step to the next.

```
CHECK|: list.
H1: for_all(j_ : integer, 1 <= j_ and j_ <= loop__1__i − 1 −> element(
        free_list, [j_]) = j_)
−−>
  C1: for_all(j_ : integer, 1 <= j_ and j_ <= loop__1__i −> element(update(
        free_list, [loop__1__i], loop__1__i), [j_]) = j_)
```

First we get rid of the universal quantifier in conclusion C1.

```
CHECK|: unwrap c#1.
>>> New goal C1: 1 <= int_j__1 and int_j__1 <= loop__1__i −> element(update(
        free_list, [loop__1__i], loop__1__i), [int_j__1]) = int_j__1
```

We start a proof by implication. The left-hand side of C1 is assumed to be true and added as hypotheses H5 and H6. Similarly to a proof by cases, a new proof frame is entered.

```
CHECK|: prove c#1 by implication.
      *** New H5: 1 <= int_j__1
      *** New H6: int_j__1 <= loop__1__i
      >>> New goal C1: element(update(free_list, [loop__1__i], loop__1__i),
            [int_j__1]) = int_j__1
```

We make a case distinction on loop__1__i because, if loop__1__i were the same as int_j__1, the array update would vanish. To this effect, we infer a new hypothesis.

```
      CHECK|: infer loop__1__i = int_j__1 or not loop__1__i = int_j__1 using inference(1).
      CHECK|: simplify.
      *** New H7: loop__1__i = int_j__1 or loop__1__i <> int_j__1
```

We let the Checker normalize the negation to an inequality sign. This extra step is necessary because the Checker cannot infer H7 directly if it contains an inequality sign. With H7 we start the prove by cases on C1.

```
      CHECK|: prove c#1 by cases on h#7.
      CASE 1: loop__1__i = int_j__1
            *** New H8: loop__1__i = int_j__1
            >>> New goal C1: loop__1__i = int_j__1
```

The inference engine of the Checker automatically simplifies conclusion C1, but it does not simplify it enough. Hypothesis H8 is identical to C1. When we manually start the inference engine of the Checker, it is able to recognize this equality.

```
CHECK|: done.
*** PROVED C1: loop__1__i = int_j__1 FOR CASE 1
```

The next case is more challenging.

```
CASE 2: loop__1__i <> int_j__1
*** New H8: loop__1__i <> int_j__1
>>> New goal C1: element(free_list, [int_j__1]) = int_j__1
```

We revert to hypothesis H1 by eliminating its quantifier.

```
CHECK|: unwrap h#1.
*** New H9: 1 <= int_J__1 and int_J__1 <= loop__1__i − 1 −> element(
        free_list, [int_J__1]) = int_J__1
```

This time the quantifier of a hypothesis is eliminated. There is a capital letter in atom int_J__1. This is not a Prolog variable but a so-called *universal variable*. The Checker requires us to instantiate it by another atom before we do anything else. We instantiate int_J__1 by int_j__1 because this is the predicate used in C1.

```
CHECK|: instantiate int_J__1 with int_j__1.
*** New H9: 1 <= int_j__1 and int_j__1 <= loop__1__i − 1 −> element(
        free_list, [int_j__1]) = int_j__1
```

Our strategy is to show that the left-hand side of this implication is true. Hypothesis H5 states that 1 <= int_j__1. We just need to infer that int_j__1 <= loop__1__i − 1.

```
CHECK|: infer loop__1__i >= int_j__1 using inequals(6).
*** New H10: loop__1__i >= int_j__1
CHECK|: infer loop__1__i > int_j__1 using strengthen(1) from [8,10].
*** New H11: loop__1__i > int_j__1
CHECK|: infer int_j__1 + 1 <= loop__1__i using inequals(102) from [11].
*** New H12: int_j__1 + 1 <= loop__1__i
CHECK|: infer int_j__1 + 1 − 1 <= loop__1__i − 1 using inequals(74) from [12].
*** New H13: int_j__1 + 1 − 1 <= loop__1__i − 1
CHECK|: infer int_j__1 <= loop__1__i − 1 using inequals(3) from [13].
*** New H14: int_j__1 <= loop__1__i − 1
```

Maybe there is an easier way to infer H14 from H8—we just did not find one. For completeness, these are the built-in rules we used:

```
strengthen(1): I>J may_be_deduced_from [ I>=J, I<>J ]

inequals(3): I−N<=J may_be_deduced_from [ I<=J, N>=0 ].

inequals(6): J>=I−N may_be_deduced_from [ I<=J, N>=0 ].

inequals(74): I−N<=J−N may_be_deduced_from [ I<=J ].

inequals(102): Y+1<=X may_be_deduced_from [ X>Y ].
```

With H5 and H14, the left-hand side of H9's implication becomes true and we can finally deduce C1.

```
            CHECK|: deduce c#1 from [5,9,14].
            *** PROVED C1: element(free_list, [int_j__1]) = int_j__1 FOR CASE 2
*** VC PROVED −− Well done!
```

Proof Session for Refinement Integrity

Similarly to the refinement integrity proofs for entry Request and procedure Release, we have to replace the predicate conf__buffer_size by 65536. We do not show the proof here.

6.3 Protected Type Priority_List

6.3.1 Annotations in the Body

The protected type Priority_List manages a list of memory frame indices, which are stored in an array. This array is logically a linked list. In analogy to the Buffer_Guard, we abstract from the array and define the properties of the Priority_List operations by means of a set. Let L be this set of indices:

$$L := \{\mathsf{Ext}\} \cup \{j \mid \exists\ i : i \in L : \mathsf{List}(i).\mathsf{Suc} = j\}$$

In contrast to the original Coco proof, we include the list extension Ext in the set. This way, we save several cases in the proof.

In analogy to the Buffer_Guard, we declare a proof function for set L. The only parameter is the array List. The return type is a set of integers:

```
−−# function L (List : in Priority_Queue) return Set_Type;
```

Again we cannot define proof function L directly as proof rule. But we can infer properties of set L. The first property is that the list extension is always a set member.

```
priority_list(1): buf__ext in I(L) may_be_deduced.
```

The name of the list extension is Ext. After the pre-processor has transferred the protected type into a child package, Ext is declared in the (parent) package Buf. This leads to the FDL identifier buf__ext. Another property of set L is that the value of the successor field of an element is also a set member. This stems from the linked list nature of the array.

```
priority_list(2): fld_suc(element(L,[I])) in I(L) may_be_deduced_from [ I in I(L) ].
```

The next proof rule builds on the array property. Every element in the array has a unique index. Every element is either in set L or not. In particular, no element in the set has the same array index as an element outside the set. We use the quantifier for sets as described in Section 4.5.

```
priority_list(3): for_all(x : integer, x in I(L) -> I <> x) may_be_deduced_from [ I not_in I(L) ].
```

In analogy to the rules buffer_guard(1) and buffer_guard(2), we define rules for insertion into and removal from the set.

```
priority_list(4): I(L) \/ (set [E]) & I(update(L,[I],upf_suc(element(L,[I]),E)))
                  are_interchangeable
                  if [ I in I(L), E not_in I(L), fld_suc(element(L,[E])) = fld_suc(element(L,[I])) ].

priority_list(5): I(L) \ (set [E]) & I(update(L,[I],upf_suc(element(L,[I]),fld_suc(element(L,[E])))))
                  are_interchangeable
                  if [ I in I(L), E in I(L), fld_suc(element(L,[I])) = E ].
```

When an element E is inserted into the set, the successor of some other element I must be E, provided that I is in the set, E is not already in the set, and the successor of E points to the previous successor of I. The removal works the opposite way.

We require a special rule for the list extension element. If the successor of the list extension is updated to be the list extension itself, then the set consists only of a single element, namely the list extension.

```
priority_list(6): I(update(_, [buf__ext], upf_suc(_, buf__ext))) & (set [buf__ext])
                  are_interchangeable.
```

When an array element is modified that is not in the set, then this has no effect on the set.

```
priority_list(7): I(update(L,[I],_)) & I(L) are_interchangeable if [ I not_in I(L) ].
```

The set is best suited to control the number of elements but it says nothing about the ordering of these elements. To accomplish the latter, we define the proof function Sorted which returns true if those elements of the array List, which are also members of the set L, are sorted in ascending order:

```
--# function Sorted (List : in Priority_Queue) return Boolean;
```

In the following descriptions, we mix the terms array and list because it depends on the view which applies best to the array variable List.

Again we specify proof rules for the insertion and the removal of elements to and from the sorted list.

```
priority_list(8): sorted(update(L,[I],upf_suc(element(L,[I]),E))) & sorted(L)
                  are_interchangeable
                  if [ I in I(L), E not_in I(L),
                       fld_suc(element(L,[E])) = fld_suc(element(L,[I])),
                       (I = buf__ext or fld_pri(element(L, [I])) <= fld_pri(element(L,[E]))),
                       fld_pri(element(L,[E])) <= fld_pri(element(L,[fld_suc(element(L,[I]))]))
                  ].

priority_list(9): sorted(update(L,[I],upf_suc(element(L,[I]),fld_suc(element(L,[E])))))
                  may_be_replaced_by sorted(L)
                  if [ I in I(L), E in I(L), fld_suc(element(L,[I])) = E ].
```

When an element E is inserted by modifying an existing element I, the same conditions must be satisfied as for rule priority_list(4). Additionally the priority of I must be lower than the priority of E except when I is the list extension. The priority of E must be lower than the priority of its successor which is the former successor of I.

When an element E is removed, only the conditions from priority_list(5) must be satisfied. A list remains sorted when a single element is removed.

A list that consists of only one element is always sorted. This single element can only be the list extension:

```
priority_list(10): sorted(update(_,[buf__ext],upf_suc(_,buf__ext))) may_be_deduced.
```

In analogy to priority_list(7), the update of an array element that is not in set L has no effect.

```
priority_list(11): sorted(update(L,[I],_)) & sorted(L) are_interchangeable if [ I not_in I(L) ].
```

The last proof rule of the proof function Sorted specifies the ordering property directly. For every element in the set (except for the list extension), its successor has a higher priority value.

```
priority_list(12): for_all(x : integer, x in I(L) and x <> buf__ext ->
                     fld_pri(element(L,[x])) <= fld_pri(element(L,[fld_suc(element(L,[x]))])))
                   may_be_deduced_from [ sorted(L) ].
```

In correspondence with the proof function Sorted, we define a proof function which returns the element with the minimum priority value, i. e. the highest priority:

```
--# function Min (List : in Priority_Queue) return System_Priority;
```

The element with the highest priority is the successor of the list extension, provided the list is sorted.

```
priority_list(13): min(L) may_be_replaced_by
                    fld_pri(element(L,[fld_suc(element(L,[buf__ext]))])) if [ sorted(L) ].
```

With the proof functions L, Sorted and Min, we can finally specify the proof annotations. The invariant of the protected type is:

```
--# invariant Sorted (List) and List (Ext).Pri = System_Priority'Last;
```

The list is always sorted and the priority of the list extension element Ext, which is always in the list, does not change.

We define the following pre- and post-conditions for the Priority_List operations:

```
procedure Initialize
--# post L (List) = Set_Element (Ext);
is
begin
   List (Ext). Pri := System_Priority'Last;
   List (Ext). Suc := Ext;
end Initialize ;

procedure Enter (Element_Index : in Buffer_Index; Priority : in System_Priority)
--# pre not Set_Member (L (List), Element_Index);
--# post L (List) = Set_Insert (L (List~), Element_Index);
is
   I : List_Index;
begin
   I := Ext;
   while (List (List (I). Suc).Pri < Priority)
   --# assert Set_Member (L (List), I) and Loop_Condition (List, List (I). Suc)
   --#        and List~ = List;
   loop
      I := List (I). Suc;
   end loop;
   List (Element_Index) := Priority_Element'( Pri => Priority, Suc => List (I). Suc);
   List (I). Suc := Element_Index;
end Enter;
```

```
entry Remove (Element_Index : out Buffer_Index) when List (Ext).Suc /= Ext
--# post L (List) = Set_Delete (L (List~),  Element_Index)
--#      and List (Element_Index).Pri = Min (List~)  and Min (List)  >= Min (List~);
is
begin
   Element_Index := List (Ext). Suc;
   List (Ext). Suc := List (List (Ext). Suc).Suc;
end Remove;
```

The initialization procedure initializes the list extension element. The set consists only of this single element.

Procedure Enter inserts an element into the list. The element must not be in the list already. This pre-condition has been erroneously omitted in the original proof of Coco. It is an essential condition because the linked list could contain inner loops otherwise. Without this pre-condition, we were not able to finish the proof successfully with the Checker.

Entry Remove removes an element from the list. There is no pre annotation because the entry barrier condition will be satisfied when the entry body is executed. The barrier states that the list must contain other elements than just Ext. The returned element Element_Index is no longer a member of the list and had the lowest priority of the list. Consequently the minimum priority value of the new list is higher than the minimum priority value of the unmodified list.

The loop invariant of procedure Enter specifies that element I is always a member of the list and that the list is not modified during the loop step. Otherwise the Examiner misses this piece of information. We declare the proof function Loop_Condition as follows:

```
--# function Loop_Condition (List : in Priority_Queue; Index : in List_Index) return Boolean;
```

Its semantics is defined by the loop condition List (List (I).Suc).Pri $<$ Priority. The loop traverses the list elements until it finds one with a higher priority value. This leads us to an inductive definition. For the base case List (Ext).Suc, the function Loop_Condition returns true.

```
priority_list(14): loop_condition(L,fld_suc(element(L,[buf__ext]))) may_be_deduced.
```

The inductive step is valid if the priority value of the current element is smaller than variable Priority and the induction hypothesis is satisfied.

```
priority_list(15): loop_condition(L,fld_suc(element(L,[I]))) may_be_deduced_from
                   [ loop_condition(L,I), fld_pri(element(L,[I])) < priority, I <> buf__ext ].
```

We require another proof rule which gives us the information that the priority value of an element is smaller than variable Priority in the loop condition.

```
priority_list(16): fld_pri(element(L,[I])) < priority may_be_deduced_from
                   [ loop_condition(L,fld_suc(element(L,[I]))), I <> buf__ext ].
```

SPARK does not support proofs of termination, so we do not prove the termination of the loop. Nevertheless we point out that the original COCO implementation uses a less-than-equal sign instead of a less-than sign in the loop condition. For very large dynamic priority values this could lead to a non-terminating loop.

6.3.2 Annotations in the Declaration

The content of this section is analogous to the declaration part of the protected type Buffer_Guard in Section 6.2.2.

The complete annotated source code of the declaration part is:

```
protected type Priority_List
--# own State : State_Type;
--# invariant Priority_List_Invariant  (State);
is
   --# type State_Type is abstract;

   --# type Set_Type is abstract;
   --# function Set_Member (S : in Set_Type; E : in List_Index) return Boolean;
   --# function Set_Delete (S : in Set_Type; E : in List_Index )  return Set_Type;
   --# function Set_Insert (S : in Set_Type; E :  in  List_Index )  return Set_Type;
   --# function Set_Element (E : in List_Index) return Set_Type;
   --# function Empty_Set return Set_Type;

   --# function L (List : in  Priority_Queue) return Set_Type;
   --# function Sorted (List : in Priority_Queue) return Boolean;
   --# function Min (List : in Priority_Queue) return System_Priority;
   --# function Loop_Condition (List : in Priority_Queue;
   --#                          Index :  in  List_Index )  return Boolean;

   --# function Priority_List_Invariant  (S :  in  State_Type) return Boolean;
   --# function Can_Be_Removed (S : in State_Type; E : in Buffer_Index) return Boolean;

   procedure Initialize ;
   --# global in out State;
   --# derives State from State;

   procedure Enter (Element_Index : in Buffer_Index; Priority   :  in  System_Priority);
   --# global in out State;
   --# derives State from Element_Index, Priority, State;
   --# pre Can_Be_Removed (State, Element_Index);
```

```
   entry Remove (Element_Index : out Buffer_Index);
   --# global in out State;
   --# derives Element_Index from State & State from State;

private
   List : Priority_Queue;
end Priority_List ;
```

As for the protected type Buffer_Guard, we name the state variable simply State. The invariant of the declaration part of the protected type Priority_List has the same semantics as in the protected type body.

```
priority_list(17): priority_list_invariant(S) may_be_replaced_by
                   sorted(fld_list(S))
                   and fld_pri(element(fld_list(S), [buf__ext])) = buf__system_priority__last.
```

The declaration part also contains the declarations of the proof functions L, Sorted, Min and Loop_Condition, in addition to the set proof functions of Section 4.5.

As for the protected type Buffer_Guard, we limit the pre- and post-conditions to the bare necessities. Therefore only procedure Enter requires a pre-condition due to refinement. It has the same semantics as the pre-condition of procedure Enter in the body:

```
priority_list(18): can_be_removed(S,E) may_be_replaced_by E not_in l(fld_list(S)).
```

6.3.3 Proof Sessions for Procedure Initialize

Proof Session from Start to Finish:

No hypotheses are required for this verification condition; The conclusions are always true.

```
CHECK|: list.
   C1: sorted(update(list, [0], upf_pri(upf_suc(element(list, [0]), 0),
          9223372036854775807)))
   C2: l(update(list, [0], upf_pri(upf_suc(element(list, [0]), 0), 9223372036
```

The Simplifier replaced every occurrence of the list extension buf__ext by its value 0. The proof rule which enables this is automatically generated by the Examiner:

```
initialize_rules(3): buf__ext may_be_replaced_by 0.
```

Our proof rules reference buf__ext. We need to replace every 0 in the verification condition by buf__ext first because otherwise we could not apply a single one of our proof rules.

```
CHECK|: infer 0 = buf__ext using initialize_rules(3).
*** New H3: 0 = buf__ext
CHECK|: replace c#1−2: 0 by buf__ext using eq(1).
>>> New goal C1: sorted(update(list, [buf__ext], upf_pri(upf_suc(element(
        list, [buf__ext]), buf__ext), 9223372036854775807)))
>>> New goal C2: l(update(list, [buf__ext], upf_pri(upf_suc(element(list,
        [buf__ext]), buf__ext), 9223372036854775807))) = (set [buf__ext])
```

We exchange the occurrences of upf_pri and upf_suc. The built-in proof rule

```
record(3): upf_F(upf_G(R,VG),VF) may_be_replaced_by upf_G(upf_F(R,VF),VG)
           if [ "F" <> "G" ].
```

enables this. We should point out that this definition of record(3) is taken verbatim from the manual [Spa04a] but in reality, the Checker uses another, more complicated, definition. The atoms upf_F and upf_G are not Prolog variables, so this definition would not work in practice.

```
CHECK|: replace c#1−2: upf_pri(upf_suc(A,B),C) by upf_suc(upf_pri(A,C),B) using record(3).
>>> New goal C1: sorted(update(list, [buf__ext], upf_suc(upf_pri(element(
        list, [buf__ext]), 9223372036854775807), buf__ext)))
>>> New goal C2: l(update(list, [buf__ext], upf_suc(upf_pri(element(list,
        [buf__ext]), 9223372036854775807), buf__ext))) = (set [buf__ext])
```

C1 can be inferred from priority_list(10) because it updates the successor of element Ext to point to Ext. The update of the priority value is irrelevant.

```
CHECK|: infer c#1 using priority_list(10).
*** PROVED C1: sorted(update(list, [buf__ext], upf_suc(upf_pri(element(list,
        [buf__ext]), 9223372036854775807), buf__ext)))
```

The left-hand side of C2 can be replaced by the set element Ext with rule priority_list(6). The resulting conclusion is trivially true.

```
CHECK|: replace c#2: l(update(_,[buf__ext],upf_suc(_,buf__ext))) by (set [buf__ext]) using
        priority_list(6).
NEW EXPRESSION: (set [buf__ext]) = (set [buf__ext])
CHECK|: done.
*** PROVED C2: (set [buf__ext]) = (set [buf__ext])
*** VC PROVED −− Well done!
```

Proof Session for Refinement Integrity

```
CHECK|: list.
H7: fld_pri(element(fld_list(state), [0])) = 9223372036854775807
−−>
```

```
C1: fld_pri(element(fld_list(state), [buf__ext])) =
      buf__system_priority__last
```

The Simplifier is not able to replace the values of System_Priority'Last and Ext. So we infer both constants from the rules which are automatically generated by the Examiner:

```
initialize_rules(3): buf__ext may_be_replaced_by 0.

initialize_rules(17): buf__system_priority__last may_be_replaced_by 9223372036854775807.
```

```
CHECK|: infer 0 = buf__ext using initialize_rules(3).
*** New H9: 0 = buf__ext
CHECK|: infer 9223372036854775807 = buf__system_priority__last using initialize_rules(17).
*** New H10: 9223372036854775807 = buf__system_priority__last
```

We replace these constants in the verification condition.

```
CHECK|: replace h#7: 0 by buf__ext using eq(1).
NEW EXPRESSION: fld_pri(element(fld_list(state), [buf__ext])) = 9223372036854
      775807
CHECK|: replace h#7: 9223372036854775807 by buf__system_priority__last using eq(1).
NEW EXPRESSION: fld_pri(element(fld_list(state), [buf__ext])) =
      buf__system_priority__last
```

After this replacement the inference engine of the Checker is able to prove this verification condition.

```
CHECK|: done.
*** PROVED C1: fld_pri(element(fld_list(state), [buf__ext])) =
      buf__system_priority__last
*** VC PROVED -- Well done!
```

6.3.4 Proof Sessions for Entry Remove

Proof Session from Start to Finish:

Before we show the hypotheses and the conclusions of the verification condition, we replace the 0 in all rules by buf__ext in analogy to the proof of procedure Initialize.

```
CHECK|: infer 0 = buf__ext using remove_rules(3).
CHECK|: replace all: 0 by buf__ext using eq(1).
CHECK|: list.
H1: sorted(list)
H2: fld_pri(element(list, [buf__ext])) = 9223372036854775807
H5: buf__ext < fld_suc(element(list, [buf__ext]))
H7: for_all(i___1 : integer, buf__ext <= i___1 and i___1 <= 65536 -> buf__ext
```

```
        <= fld_pri(element(list, [i___1])) and fld_pri(element(list, [i___1])
        ) <= 9223372036854775807)
H8: fld_suc(element(list, [buf__ext])) <= 65536
-->
  C1: sorted(update(list, [buf__ext], upf_suc(element(list, [buf__ext]),
        fld_suc(element(list, [fld_suc(element(list, [buf__ext]))])))))
  C2: l(update(list, [buf__ext], upf_suc(element(list, [buf__ext]), fld_suc(
        element(list, [fld_suc(element(list, [buf__ext]))])))))) = l(list)
        \ (set [fld_suc(element(list, [buf__ext]))])
  C3: fld_pri(element(list, [fld_suc(element(list, [buf__ext]))])) = min(
        list)
  C4: min(update(list, [buf__ext], upf_suc(element(list, [buf__ext]),
        fld_suc(element(list, [fld_suc(element(list, [buf__ext]))])))))
        >= min(list)
```

There are four conclusions. In contrast to the previous proofs, we take a more bottom-up approach. We must occasionally enter a different prove frame. All hypotheses inferred in one proof frame get lost on exit of that frame. We aim to infer as many hypotheses as possible before a proof frame is entered, so we can re-use them for the remaining conclusions.

We start with C1. First we infer that buf__ext and its successor are in the list:

```
CHECK|: infer buf__ext in l(list) using priority_list(1).
*** New H12: buf__ext in l(list)
CHECK|: infer fld_suc(element(list,[buf__ext])) in l(list) using priority_list(2) from [12].
*** New H13: fld_suc(element(list, [buf__ext])) in l(list)
```

Our strategy is to craft a hypothesis with the same content as C1.

```
CHECK|: standardise sorted(update(list,[buf__ext],
        upf_suc(element(list,[buf__ext]),fld_suc(element(list,[fld_suc(element(list,[buf__ext]))])))))).
*** New H14: sorted(update(list, [buf__ext], upf_suc(element(list, [buf__ext
        ]), fld_suc(element(list, [fld_suc(element(list, [buf__ext]))])))))
        <-> sorted(update(list, [buf__ext], upf_suc(element(list, [buf__ext]),
        fld_suc(element(list, [fld_suc(element(list, [buf__ext]))])))))
```

H12 and H13 provide the necessary conditions that must be satisfied before we can apply priority_list(9):

1. buf__ext in l(list)
2. fld_suc(element(list, [buf__ext])) in l(list)
3. fld_suc(element(list, [buf__ext])) = fld_suc(element(list, [buf__ext]))

The third condition of priority_list(9) is in this configuration trivially true.

```
CHECK|: replace h#14: sorted(update(L,[I],upf_suc(element(L,[I]),fld_suc(element(L,[E])))))
        by sorted(L) using priority_list(9).
Change which occurrence (number/none/all)? |: 1.
NEW EXPRESSION: sorted(list) <−> sorted(update(list, [buf__ext], upf_suc(
        element(list, [buf__ext]), fld_suc(element(list, [fld_suc(element(
        list, [buf__ext]))]))))))
```

The left-hand side of the equivalence in H14 is the same as H1. Thus, we can eliminate the left-hand side.

```
CHECK|: forwardchain h#14.
** New H14: sorted(update(list, [buf__ext], upf_suc(element(list, [buf__ext
        ]), fld_suc(element(list, [fld_suc(element(list, [buf__ext]))]))))))
```

Hypothesis H14 is now identical to C1.

```
CHECK|: done.
*** PROVED C1: sorted(update(list, [buf__ext], upf_suc(element(list, [
        buf__ext]), fld_suc(element(list, [fld_suc(element(list, [buf__ext]))
        ])))))
```

In a top-down approach, we would apply rule priority_list(9) directly on C1. This would simplify C1 to sorted(list) which is already stated by hypothesis H1 and, therefore, true. But with the bottom-up approach we have won the new hypothesis H14, which we can re-use for C4.

To eliminate C2 we apply priority_list(5). The necessary conditions are the same as for the application of priority_list(9) and the Checker satisfies them by using H12 and H13.

```
CHECK|: replace c#2: I(update(L,[I],upf_suc(element(L,[I]),fld_suc(element(L,[E]))))) by
        I(L) \ (set [E]) using priority_list(5).
NEW EXPRESSION: I(list) \ (set [fld_suc(element(list, [buf__ext]))]) =
        I(list) \ (set [fld_suc(element(list, [buf__ext]))])
```

The resulting equation is true.

```
CHECK|: done.
*** PROVED C2: I(list) \ (set [fld_suc(element(list, [buf__ext]))]) = I(list)
        \ (set [fld_suc(element(list, [buf__ext]))])
```

For C3 and C4 we first resolve the predicate min with priority_list(13). This rule can be applied because the list is sorted as stated by H1.

```
CHECK|: replace c#3−4: min(L) by fld_pri(element(L,[fld_suc(element(L,[buf__ext]))])) using
        priority_list(13).
Change which subexpression (number/none)? |: 1.
```

```
>>> New goal C3: fld_pri(element(list, [fld_suc(element(list, [buf__ext]))])
        ) = fld_pri(element(list, [fld_suc(element(list, [buf__ext]))]))
>>> New goal C4: min(update(list, [buf__ext], upf_suc(element(list, [
        buf__ext]), fld_suc(element(list, [fld_suc(element(list, [buf__ext]))
        ])))) >= fld_pri(element(list, [fld_suc(element(list, [buf__ext]))])
        )
```

The resulting equation of C3 is trivially true.

```
CHECK|: done.
*** PROVED C3: fld_pri(element(list, [fld_suc(element(list, [buf__ext]))]))
        = fld_pri(element(list, [fld_suc(element(list, [buf__ext]))]))
```

C4 contains the predicate min a second time. The condition for the application of priority_list(13) is now:

```
update(list, [buf__ext], upf_suc(element(list, [buf__ext]),
       fld_suc(element(list, [fld_suc(element(list, [buf__ext]))])))
```

which is satisfied by the previously inferred hypothesis H14.

```
CHECK|: replace c#4: min(L) by fld_pri(element(L,[fld_suc(element(L,[buf__ext]))])) using
        priority_list(13).
NEW EXPRESSION: fld_pri(element(update(list, [buf__ext], upf_suc(element(list,
        [buf__ext]), fld_suc(element(list, [fld_suc(element(list, [buf__ext])
        )])))), [fld_suc(element(update(list, [buf__ext], upf_suc(element(
        list, [buf__ext]), fld_suc(element(list, [fld_suc(element(list,
        [buf__ext]))])))), [buf__ext]))])) >= fld_pri(element(list, [fld_suc(
        element(list, [buf__ext]))]))
```

We simplify the resulting conclusion C4 by eliminating a redundant array update and a redundant record update. The build-in proof rules which we use here are:

```
array(1): element(update(A,I,X),I) may_be_replaced_by X.

record(1): fld_F(upf_F(_, VALUE)) may_be_replaced_by VALUE.
```

```
CHECK|: replace c#4: element(update(A,I,X),I) by X using array(1).
NEW EXPRESSION: fld_pri(element(update(list, [buf__ext], upf_suc(element(list,
        [buf__ext]), fld_suc(element(list, [fld_suc(element(list, [buf__ext])
        )])))), [fld_suc(upf_suc(element(list, [buf__ext]), fld_suc(element(
        list, [fld_suc(element(list, [buf__ext]))])))])) >= fld_pri(element(
        list, [fld_suc(element(list, [buf__ext]))]))
CHECK|: replace c#4: fld_suc(upf_suc(_,X)) by X using record(1).
NEW EXPRESSION: fld_pri(element(update(list, [buf__ext], upf_suc(element(list,
        [buf__ext]), fld_suc(element(list, [fld_suc(element(list, [buf__ext])
        )])))), [fld_suc(element(list, [fld_suc(element(list, [buf__ext]))]))
        ])) >= fld_pri(element(list, [fld_suc(element(list, [buf__ext]))]))
```

Before we can proceed we must make a case distinction on the list extension buf__ext. First we infer the cases as new hypothesis.

```
CHECK|: infer fld_suc(element(list,[fld_suc(element(list,[buf__ext]))])) = buf__ext or
        not fld_suc(element(list,[fld_suc(element(list,[buf__ext]))])) = buf__ext using inference(1).
*** New H15: fld_suc(element(list, [fld_suc(element(list, [buf__ext]))]))
        = buf__ext or not fld_suc(element(list, [fld_suc(element(list,
        [buf__ext]))])) = buf__ext
```

Then we start the proof of C4 by cases.

```
prove c#4 by cases on h#15.
CASE 1: fld_suc(element(list, [fld_suc(element(list, [buf__ext]))])) =
        buf__ext
    *** New H16: fld_suc(element(list, [fld_suc(element(list, [buf__ext]))]))
            = buf__ext
    >>> New goal C1: fld_pri(element(update(list, [buf__ext], upf_suc(element(
            list, [buf__ext]), fld_suc(element(list, [fld_suc(element(list,
            [buf__ext]))])))), [fld_suc(element(list, [fld_suc(element(list,
            [buf__ext]))]))])) >= fld_pri(element(list, [fld_suc(element(list,
            [buf__ext]))]))
```

C4 becomes C1 in this new proof frame. We simplify C1 with H16.

```
    CHECK|: replace c#1: fld_suc(element(list,[fld_suc(element(list,[buf__ext]))])) by buf__ext
            using eq(1).
    Change which occurrence (number/none/all)? |: all.
    NEW EXPRESSION: fld_pri(element(update(list, [buf__ext], upf_suc(element(list,
            [buf__ext]), buf__ext)), [buf__ext])) >= fld_pri(element(list,
            [fld_suc(element(list, [buf__ext]))]))
```

We simplify C1 further by removing a redundant array update and a redundant record update. The built-in rule record(4) is defined as follows:

```
record(4): fld_F(upf_G(R, V)) may_be_replaced_by fld_F(R) if [ "F" <> "G" ].
```

```
    CHECK|: replace c#1: element(update(A,I,X),I) by X using array(1).
    NEW EXPRESSION: fld_pri(upf_suc(element(list, [buf__ext]), buf__ext)) >=
            fld_pri(element(list, [fld_suc(element(list, [buf__ext]))]))
    CHECK|: replace c#1: fld_pri(upf_suc(R,V)) by fld_pri(R) using record(4).
    NEW EXPRESSION: fld_pri(element(list, [buf__ext])) >= fld_pri(element(list,
            [fld_suc(element(list, [buf__ext]))]))
```

H2 permits a further simplification of C1.

```
    CHECK|: replace c#1: fld_pri(element(list,[buf__ext])) by 9223372036854775807 using
            eq(1).
    NEW EXPRESSION: 9223372036854775807 >= fld_pri(element(list, [fld_suc(element(
            list, [buf__ext]))]))
```

We instantiate the universal quantifier of H7 with the successor element of buf__ext.

```
CHECK|: unwrap h#7.
*** New H17: buf__ext <= int_l___1_1 and int_l___1_1 <= 65536 -> buf__ext
        <= fld_pri(element(list, [int_l___1_1])) and fld_pri(element(list,
        [int_l___1_1])) <= 9223372036854775807
CHECK|: instantiate int_l___1_1 with fld_suc(element(list,[buf__ext])).
*** New H17: buf__ext <= fld_suc(element(list, [buf__ext])) and fld_suc(
        element(list, [buf__ext])) <= 65536 -> buf__ext <= fld_pri(element(list,
        [fld_suc(element(list, [buf__ext]))])) and fld_pri(element(list,
        [fld_suc(element(list, [buf__ext]))])) <= 9223372036854775807
```

The left-hand side of this implication is satisfied by H5 and H8; therefore we can eliminate it.

```
CHECK|: forwardchain h#17.
*** New H17: buf__ext <= fld_pri(element(list, [fld_suc(element(list,
        [buf__ext]))])) and fld_pri(element(list, [fld_suc(element(list,
        [buf__ext]))])) <= 9223372036854775807
```

We separate the conjunction of H17 into H17 and H18. The command simplify normalizes hypotheses and conclusions. As documented in the Checker on-line help, this command ignores its optional arguments, on which hypotheses or conclusions it should be applied, so the result can sometimes be a surprise.

```
CHECK|: simplify.
*** New H17: buf__ext <= fld_pri(element(list, [fld_suc(element(list,
        [buf__ext]))]))
*** New H18: fld_pri(element(list, [fld_suc(element(list, [buf__ext]))]))
        <= 9223372036854775807
```

This new hypothesis H18 satisfies C1.

```
CHECK|: done.
*** PROVED C1: 9223372036854775807 >= fld_pri(element(list, [fld_suc(element(
        list, [buf__ext]))])) FOR CASE 1
```

In the second case C4 becomes C1 again.

```
CASE 2: not fld_suc(element(list, [fld_suc(element(list, [buf__ext]))]))
          = buf__ext
*** New H16: not fld_suc(element(list, [fld_suc(element(list, [buf__ext]))])
        ) = buf__ext
>>> New goal C1: fld_pri(element(update(list, [buf__ext], upf_suc(element(
        list, [buf__ext]), fld_suc(element(list, [fld_suc(element(list,
        [buf__ext]))])))), [fld_suc(element(list, [fld_suc(element(list,
        [buf__ext]))]))])) >= fld_pri(element(list, [fld_suc(element(list,
        [buf__ext]))]))
```

We eliminate the superfluous array update with the built-in rule array(3):

```
array(3): element(update(A,J,X),K) & element(A,K) are_interchangeable if [ J<>K ].
```

```
CHECK|: replace c#1: element(update(A,J,X),K) by element(A,K) using array(3).
NEW EXPRESSION: fld_pri(element(list, [fld_suc(element(list, [fld_suc(element(
        list, [buf__ext]))]))])) >= fld_pri(element(list, [fld_suc(element(
        list, [buf__ext]))]))
```

H1 states that the list is sorted. With this information and rule priority_list(12) we infer a new hypothesis. We are interested in the successor element of buf__ext.

```
CHECK|: infer for_all(x : integer,x in l(list) and x <> buf__ext ->
        fld_pri(element(list,[x])) <= fld_pri(element(list,[fld_suc(element(list,[x]))])))
        using priority_list(12).
*** New H17: for_all(x : integer, x in l(list) and x <> buf__ext -> fld_pri(
        element(list, [x])) <= fld_pri(element(list, [fld_suc(element(list,
        [x]))])))
```

We instantiate the universal quantifier with the successor element of buf__ext.

```
CHECK|: unwrap h#17.
*** New H18: int_X_1 in l(list) and int_X_1 <> buf__ext -> fld_pri(element(
        list, [int_X_1])) <= fld_pri(element(list, [fld_suc(element(list,
        [int_X_1]))]))
CHECK|: instantiate int_X_1 with fld_suc(element(list,[buf__ext])).
*** New H18: fld_suc(element(list, [buf__ext])) in l(list) and fld_suc(
        element(list, [buf__ext])) <> buf__ext -> fld_pri(element(list,
        [fld_suc(element(list, [buf__ext]))])) <= fld_pri(element(list,
        [fld_suc(element(list, [fld_suc(element(list, [buf__ext]))]))]))
```

The left-hand side of the implication is satisfied by H13 and H16. We can eliminate it.

```
CHECK|: forwardchain h#18.
*** New H18: fld_pri(element(list, [fld_suc(element(list, [buf__ext]))]))
        <= fld_pri(element(list, [fld_suc(element(list, [fld_suc(element(list,
        [buf__ext]))]))]))
```

The resulting inequality in hypothesis H18 is similar to the inequality in conclusion C1. The inference engine of the Checker is able to recognize this.

```
CHECK|: done.
*** PROVED C1: fld_pri(element(list, [fld_suc(element(list, [fld_suc(element(
        list, [buf__ext]))]))])) >= fld_pri(element(list, [fld_suc(element(
        list, [buf__ext]))])) FOR CASE 2
*** VC PROVED -- Well done!
```

Proof Session for Refinement Integrity

There are two verification conditions to prove. In analogy to the refinement integrity proof of procedure Initialize, only the constants buf__system_priority__last and buf__ext must be replaced. We do not show this here.

6.3.5 Proof Sessions for Procedure Enter

Proof Session from Start to Loop Invariant

As in the proof of procedure Remove we must replace 0 by buf__ext before we can apply any of our proof rules.

```
CHECK|: infer 0 = buf__ext using enter_rules(3).
CHECK|: replace all: 0 by buf__ext using eq(1).
CHECK|: list.
C1: buf__ext in I(list)
C2: loop_condition(list, fld_suc(element(list, [buf__ext])))
```

C1 and C2 can be inferred directly; no hypotheses are required. Both are the base cases of an induction.

```
CHECK|: infer c#1 using priority_list(1).
*** PROVED C1: buf__ext in I(list)
CHECK|: infer c#2 using priority_list(14).
*** PROVED C2: loop_condition(list, fld_suc(element(list, [buf__ext])))
*** VC PROVED -- Well done!
```

Proof Session for Loop Invariant

This verification condition is the inductive step of the loop.

```
CHECK|: infer 0 = buf__ext using enter_rules(3).
CHECK|: replace all: 0 by buf__ext using eq(1).
CHECK|: list.
H1: i in I(list__OLD)
H2: loop_condition(list__OLD, fld_suc(element(list__OLD, [i])))
H16: fld_pri(element(list__OLD, [fld_suc(element(list__OLD, [i]))])) <
        priority
-->
  C1: fld_suc(element(list__OLD, [i])) in I(list__OLD)
  C2: loop_condition(list__OLD, fld_suc(element(list__OLD, [fld_suc(element(
        list__OLD, [i]))])))
```

In the verification condition file, generated by the Examiner, the predicate list~ is used instead of list__OLD. Obviously the Checker replaces all tildes by the the suffix __OLD.

The successor of the list extension is always a member of the set. We can infer this using priority_list(2).

```
CHECK|: infer c#1 using priority_list(2).
*** PROVED C1: fld_suc(element(list__OLD, [i])) in I(list__OLD)
```

Before we start a case distinction on the list extension, we create the cases as a new hypothesis.

```
CHECK|: infer fld_suc(element(list__OLD,[i])) = buf__ext or
            not fld_suc(element(list__OLD,[i])) = buf__ext using inference(1).
CHECK|: simplify.
*** New H19: fld_suc(element(list__OLD, [i])) = buf__ext or fld_suc(element(
        list__OLD, [i])) <> buf__ext
```

We start a proof by cases; a new proof frame is entered and C2 becomes C1.

```
CHECK|: prove c#2 by cases on h#19.
CASE 1: fld_suc(element(list__OLD, [i])) = buf__ext
    *** New H20: fld_suc(element(list__OLD, [i])) = buf__ext
    >>> New goal C1: loop_condition(list__OLD, fld_suc(element(list__OLD,
            [fld_suc(element(list__OLD, [i]))])))
```

In the case of H20 we can replace the successor element and get again the base case of loop_condition.

```
    CHECK|: replace c#1: fld_suc(element(list__OLD,[i])) by buf__ext using eq(1).
    NEW EXPRESSION: loop_condition(list__OLD, fld_suc(element(list__OLD, [
            buf__ext])))
    CHECK|: infer c#1 using priority_list(14).
    *** PROVED C1: loop_condition(list__OLD, fld_suc(element(list__OLD, [buf__ext
            ]))) FOR CASE 1
```

In Case 2 we have to deal with the inductive step of loop_condition.

```
CASE 2: fld_suc(element(list__OLD, [i])) <> buf__ext
    *** New H20: fld_suc(element(list__OLD, [i])) <> buf__ext
    >>> New goal C1: loop_condition(list__OLD, fld_suc(element(list__OLD,
            [fld_suc(element(list__OLD, [i]))])))
```

The application of priority_list(15) is justified by case H20, the inductive hypothesis H2 and the the less-than-equal relation of H16.

```
      CHECK|: infer c#1 using priority_list(15) from [2,16,20].
      *** PROVED C1: loop_condition(list__OLD, fld_suc(element(list__OLD, [fld_suc(
            element(list__OLD, [i]))]))) FOR CASE 2
*** VC PROVED -- Well done!
```

Proof Session from Loop Invariant to Finish

```
CHECK|: infer 0 = buf__ext using enter_rules(3).
CHECK|: replace all: 0 by buf__ext using eq(1).
CHECK|: list.
H1: i in l(list__OLD)
H2: loop_condition(list__OLD, fld_suc(element(list__OLD, [i])))
H11: not element_index in l(list__OLD)
H16: priority <= fld_pri(element(list__OLD, [fld_suc(element(list__OLD,
       [i]))]))
-->
  C1: sorted(update(update(list__OLD, [element_index],
       mk__buf__priority_element(pri := priority, suc := fld_suc(element(
       list__OLD, [i])))), [i], upf_suc(element(update(list__OLD, [
       element_index], mk__buf__priority_element(pri := priority, suc
       := fld_suc(element(list__OLD, [i])))), [i]), element_index)))
  C2: l(update(update(list__OLD, [element_index], mk__buf__priority_element(
       pri := priority, suc := fld_suc(element(list__OLD, [i])))), [i],
       upf_suc(element(update(list__OLD, [element_index],
       mk__buf__priority_element(pri := priority, suc := fld_suc(element(
       list__OLD, [i])))), [i]), element_index))) = l(list__OLD) \/ (set
       [element_index])
```

The central proof rules we want to apply to C1 and C2 are the insertion rules priority_list(8) and priority_list(4). Both share some conditions that must be satisfied before their application. Therefore we try to infer as many hypotheses as possible before we enter any proof frame, so we can re-use hypotheses.

We start with C1. Our strategy is to apply rule priority_list(8). This proof rule can only be applied if certain conditions are met. These are:

1. i in l(L)
2. element_index not_in l(L)
3. fld_suc(element(L,[element_index])) = fld_suc(element(L,[i]))
4. i = buf__ext or
   fld_pri(element(L, [i])) <= fld_pri(element(L,[element_index]))
5. fld_pri(element(L,[element_index])) <= fld_pri(element(L,[fld_suc(element(L,[i]))]))

where:

```
L = l(update(list__OLD, [element_index],
             mk__buf__priority_element(pri := priority, suc := fld_suc(element(list__OLD, [i])))))
```

For Condition 4 we require a case distinction.

Firstly we infer Condition 1 as hypothesis.

```
CHECK|: infer l(list__OLD) = l(list__OLD) using inference(1).
*** New H18: l(list__OLD) = l(list__OLD)
```

We modify H11 such that the application of priority_list(7) to H18 is justified.

```
CHECK|: replace h#11: not (X in A) by X not_in A using sets(2).
NEW EXPRESSION: element_index not_in l(list__OLD)
CHECK|: replace h#18: l(L) by l(update(L,[element_index], mk__buf__priority_element(
        pri := priority,suc := fld_suc(element(list__OLD,[i])))))) using priority_list(7).
Change which occurrence (number/none/all)? |: 2.
NEW EXPRESSION: l(list__OLD) = l(update(list__OLD, [element_index],
        mk__buf__priority_element(pri := priority, suc := fld_suc(element(
        list__OLD, [i])))))
```

This hypothesis H18 and hypothesis H1 satisfy the necessary conditions to apply sets(12). With this rule we infer Condition 1 for the application of priority_list(8).

```
CHECK|: infer i in l(update(list__OLD,[element_index], mk__buf__priority_element(pri := priority,
        suc := fld_suc(element(list__OLD,[i])))))) using sets(12) from [1,18].
*** New H19: i in l(update(list__OLD, [element_index],
        mk__buf__priority_element(pri := priority, suc := fld_suc(element(
        list__OLD, [i])))))
```

The choice of sets(12) might come as a surprise. We found this rule by letting the inference engine do a wildcard search. It is possible to specify the expression WILD instead of a rule name to the command infer. The Checker then searches its rule database for suitable rules and asks the user to choose one.

Rule sets(12) is defined as follows:

```
sets(12): X in (A /\ B) may_be_deduced_from [ X in A, X in B ].
```

We re-use H18 from above together with H11 to infer the hypothesis of Condition 2 with the built-in rule sets(6), which we found with a wildcard search, too. It is defined as follows:

```
sets(6): X not_in (A \/ B) may_be_deduced_from [ X not_in A, X not_in B ].
```

```
CHECK|: infer element_index not_in l(update(list__OLD,[element_index], mk__buf__priority_element(
        pri := priority,suc := fld_suc(element(list__OLD,[i])))))) using sets(6) from [11,18].
*** New H20: element_index not_in l(update(list__OLD, [element_index],
        mk__buf__priority_element(pri := priority, suc := fld_suc(element(
        list__OLD, [i])))))
```

We start the inference of Condition 3 by crafting a new hypothesis.

```
CHECK|: standardise fld_suc(element(update(list__OLD,[element_index],mk__buf__priority_element(
        pri := priority,suc := fld_suc(element(list__OLD,[i])))),[element_index])).
*** New H21: fld_suc(element(update(list__OLD, [element_index],
        mk__buf__priority_element(pri := priority, suc := fld_suc(element(
        list__OLD, [i])))), [element_index])) = fld_suc(element(update(
        list__OLD, [element_index], mk__buf__priority_element(pri := priority,
        suc := fld_suc(element(list__OLD, [i])))), [element_index]))
```

We simplify the left-hand side of H21.

```
CHECK|: replace h#21: element(update(A,I,X),I) by X using array(1).
NEW EXPRESSION: fld_suc(mk__buf__priority_element(pri := priority, suc
        := fld_suc(element(list__OLD, [i])))) = fld_suc(element(update(
        list__OLD, [element_index], mk__buf__priority_element(pri := priority,
        suc := fld_suc(element(list__OLD, [i])))), [element_index]))
CHECK|: replace h#21: fld_suc(mk__buf__priority_element(pri := _,suc := S)) by S using
        mk__record(3).
NEW EXPRESSION: fld_suc(element(list__OLD, [i])) = fld_suc(element(update(
        list__OLD, [element_index], mk__buf__priority_element(pri := priority,
        suc := fld_suc(element(list__OLD, [i])))), [element_index]))
```

The built-in proof rule mk__record(3) is defined as follows:

```
mk__record(3): fld_FIELD(mk__RECORDTYPE(LARGS...(FIELD := VALUE)...))
               may_be_replaced_by VALUE
               if [ not ((FIELD := ...) is_in LARGS), not ((FIELD := ...) is_in RARGS) ]
```

We require the information that element Element_Index is not already in the list, i. e. element_index <> i. We infer this from priority_list(3) whose conditions are satisfied by H11.

```
CHECK|: infer for_all(x : integer,x in l(list__OLD) -> element_index <> x) using priority_list(3)
        from [11].
*** New H22: for_all(x : integer, x in l(list__OLD) -> element_index <>
        x)
```

We instantiate the universal quantifier with element i.

```
CHECK|: unwrap h#22.
*** New H23: int_X_1 in l(list__OLD) -> element_index <> int_X_1
CHECK|: instantiate int_X_1 with i.
```

```
*** New H23: i in I(list__OLD) -> element_index <> i
```

The left-hand side is true because of H1. We eliminate it.

```
CHECK|: forwardchain h#23.
*** New H23: element_index <> i
```

Due to H23 we can expand H21 further which yields Condition 3 for the application of priority_list(8).

```
CHECK|: replace h#21: element(list__OLD,[i]) by element(update(list__OLD,[element_index],
        mk__buf__priority_element(pri := priority,suc := fld_suc(element(list__OLD,[i]))))),[i])
        using eq(1).
Change which occurrence (number/none/all)? |: 1.
NEW EXPRESSION: fld_suc(element(update(list__OLD, [element_index],
        mk__buf__priority_element(pri := priority, suc := fld_suc(element(
        list__OLD, [i])))), [i])) = fld_suc(element(update(list__OLD, [
        element_index], mk__buf__priority_element(pri := priority, suc
        := fld_suc(element(list__OLD, [i])))), [element_index]))
```

We defer Condition 4 because it requires a proof by cases and, therefore, a new proof frame.

Hypothesis H16 should become the basis for Condition 5. We make preparations to replace predicate priority in H16.

```
CHECK|: standardise fld_pri(element(update(list__OLD,[element_index],mk__buf__priority_element(
        pri := priority,suc := fld_suc(element(list__OLD,[i]))))),[element_index])).
*** New H24: fld_pri(element(update(list__OLD, [element_index],
        mk__buf__priority_element(pri := priority, suc := fld_suc(element(
        list__OLD, [i])))), [element_index])) = fld_pri(element(update(
        list__OLD, [element_index], mk__buf__priority_element(pri := priority,
        suc := fld_suc(element(list__OLD, [i])))), [element_index]))
```

We simplify the left-hand side of the new hypothesis H24.

```
CHECK|: replace h#24: element(update(A,I,X),I) by X using array(1).
Change which occurrence (number/none/all)? |: 1.
NEW EXPRESSION: fld_pri(mk__buf__priority_element(pri := priority, suc
        := fld_suc(element(list__OLD, [i])))) = fld_pri(element(update(
        list__OLD, [element_index], mk__buf__priority_element(pri := priority,
        suc := fld_suc(element(list__OLD, [i])))), [element_index]))
CHECK|: replace h#24: fld_pri(mk__buf__priority_element(pri := P,suc := _)) by P using
        mk__record(3).
NEW EXPRESSION: priority = fld_pri(element(update(list__OLD, [element_index],
        mk__buf__priority_element(pri := priority, suc := fld_suc(element(
        list__OLD, [i])))), [element_index]))
```

We use H24 to replace predicate priority in H16.

```
CHECK|: replace h#16: priority by fld_pri(element(update(list__OLD,[element_index],
        mk__buf__priority_element(pri := priority,suc := fld_suc(element(list__OLD,[i])))),
        [element_index])) using eq(1).
NEW EXPRESSION: fld_pri(element(update(list__OLD, [element_index],
        mk__buf__priority_element(pri := priority, suc := fld_suc(element(
        list__OLD, [i])))), [element_index])) <= fld_pri(element(list__OLD,
        [fld_suc(element(list__OLD, [i]))]))
```

Before we can adjust H16 further we need to infer another hypothesis.

```
CHECK|: standardise fld_pri(element(update(list__OLD,[element_index],
        mk__buf__priority_element(pri := priority,suc := X)),[X])) where
        X = fld_suc(element(list__OLD,[i])).
*** New H25: fld_pri(element(update(list__OLD, [element_index],
        mk__buf__priority_element(pri := priority, suc := fld_suc(element(
        list__OLD, [i])))), [fld_suc(element(list__OLD, [i]))])) = fld_pri(
        element(update(list__OLD, [element_index], mk__buf__priority_element(
        pri := priority, suc := fld_suc(element(list__OLD, [i])))), [fld_suc(
        element(list__OLD, [i]))]))
```

The syntax with where allows us to use Prolog variables. We use this variant because firstly it saves typing and secondly the Checker seems to cut off everything that is longer than the size of its input buffer. In this case, the Checker prints the misleading error message "!!! Command not recognized. Please retype."

We require the information element_index <> fld_suc(element(list__OLD, [i])) for the application of rule array(3). That is why we instantiate H22 a second time, this time with fld_suc(element(list__OLD,[i])).

```
CHECK|: unwrap h#22.
*** New H26: int_X_2 in l(list__OLD) -> element_index <> int_X_2
CHECK|: instantiate int_X_2 with fld_suc(element(list__OLD,[i])).
*** New H26: fld_suc(element(list__OLD, [i])) in l(list__OLD) ->
        element_index <> fld_suc(element(list__OLD, [i]))
```

We eliminate the left-hand side of the implication using priority_list(2).

```
CHECK|: infer fld_suc(element(list__OLD,[i])) in l(list__OLD) using priority_list(2) from [1].
*** New H27: fld_suc(element(list__OLD, [i])) in l(list__OLD)
CHECK|: forwardchain h#26.
*** New H26: element_index <> fld_suc(element(list__OLD, [i]))
```

Hypothesis H26 justifies the application of array(3).

```
CHECK|: replace h#25: element(update(A,J,X),K) by element(A,K) using array(3).
NEW EXPRESSION: fld_pri(element(list__OLD, [fld_suc(element(list__OLD,
```

```
        [i]))])) = fld_pri(element(update(list__OLD, [element_index],
        mk__buf__priority_element(pri := priority, suc := fld_suc(element(
        list__OLD, [i]))))), [fld_suc(element(list__OLD, [i]))]))
```

Now we can adjust H16 further. We expand it with the help of H25.

```
CHECK|: replace h#16: fld_pri(element(list__OLD,[X])) by fld_pri(element(update(list__OLD,
        [element_index],mk__buf__priority_element(pri := priority,suc := X)),[X])) where
        X = fld_suc(element(list__OLD,[i])) using eq(1).
NEW EXPRESSION: fld_pri(element(update(list__OLD, [element_index],
        mk__buf__priority_element(pri := priority, suc := fld_suc(element(
        list__OLD, [i]))))), [element_index])) <= fld_pri(element(update(
        list__OLD, [element_index], mk__buf__priority_element(pri := priority,
        suc := fld_suc(element(list__OLD, [i]))))), [fld_suc(element(list__OLD,
        [i]))]))
```

Now we only need to replace fld_suc(element(list__OLD,[i])) in H16. We craft another hypothesis for it.

```
CHECK|: standardise fld_suc(element(update(list__OLD,[element_index],mk__buf__priority_element(
        pri := priority,suc := fld_suc(element(list__OLD,[i]))))),[i])).
*** New H28: fld_suc(element(update(list__OLD, [element_index],
        mk__buf__priority_element(pri := priority, suc := fld_suc(element(
        list__OLD, [i]))))), [i])) = fld_suc(element(update(list__OLD, [
        element_index], mk__buf__priority_element(pri := priority, suc
        := fld_suc(element(list__OLD, [i]))))), [i]))
```

Because H23 states that element_index <> i, we can apply array(3).

```
    CHECK|: replace h#28: element(update(A,J,X),K) by element(A,K) using array(3).
    Change which occurrence (number/none/all)? |: 1.
    NEW EXPRESSION: fld_suc(element(list__OLD, [i])) = fld_suc(element(update(
            list__OLD, [element_index], mk__buf__priority_element(pri := priority,
            suc := fld_suc(element(list__OLD, [i]))))), [i]))
```

The last modification of H16 results in Condition 5 for the application of priority_list(8).

```
    CHECK|: replace h#16: fld_suc(element(list__OLD,[i])) by fld_suc(element(update(list__OLD,
            [element_index],mk__buf__priority_element(pri := priority,
            suc := fld_suc(element(list__OLD,[i]))))),[i])) using eq(1).
    Change which occurrence (number/none/all)? |: 3.
    NEW EXPRESSION: fld_pri(element(update(list__OLD, [element_index],
            mk__buf__priority_element(pri := priority, suc := fld_suc(element(
            list__OLD, [i]))))), [element_index])) <= fld_pri(element(update(
            list__OLD, [element_index], mk__buf__priority_element(pri := priority,
            suc := fld_suc(element(list__OLD, [i]))))), [fld_suc(element(update(
            list__OLD, [element_index], mk__buf__priority_element(pri := priority,
            suc := fld_suc(element(list__OLD, [i]))))), [i]))])
```

We prepare the case distinction for C1.

```
CHECK|: infer i = buf__ext or not i = buf__ext using inference(1).
*** New H29: i = buf__ext or not i = buf__ext
```

We start the prove by cases for C1.

```
CHECK|: prove c#1 by cases on h#29.
CASE 1: i = buf__ext
    *** New H30: i = buf__ext
    >>> New goal C1: sorted(update(update(list__OLD, [element_index],
        mk__buf__priority_element(pri := priority, suc := fld_suc(element(
        list__OLD, [i]))))), [i], upf_suc(element(update(list__OLD, [
        element_index], mk__buf__priority_element(pri := priority, suc
        := fld_suc(element(list__OLD, [i]))))), [i]), element_index)))
```

Case H30 is Condition 4 for the application of priority_list(8).

We have collected the necessary conditions H16, H19, H20, H21 and H30 to apply priority_list(8) to C1.

```
    CHECK|: replace c#1: sorted(update(L,[I],upf_suc(element(L,[I]),E))) by sorted(L) using
        priority_list(8).
    NEW EXPRESSION: sorted(update(list__OLD, [element_index],
        mk__buf__priority_element(pri := priority, suc := fld_suc(element(
        list__OLD, [i])))))
```

C1 has now a much simpler form and can be simplified further by priority_list(11), which is justified by H11.

```
    CHECK|: replace c#1: sorted(update(L,[element_index],_)) by sorted(L) using
        priority_list(11).
    Possible replacements for sorted(update(list__OLD, [element_index],
        mk__buf__priority_element(pri := priority, suc := fld_suc(element(
        list__OLD, [i])))))) are:
      1. sorted(list__OLD)
      2. sorted(update(update(list__OLD, [element_index],
        mk__buf__priority_element(pri := priority, suc := fld_suc(element(
        list__OLD, [i]))))), [element_index], _6))
    Select (number/none): |: 1.
    NEW EXPRESSION: sorted(list__OLD)
```

The Checker asks us which of these two possible replacements we want to use. Only the first option makes sense and is our intended result. The second option contains the atom _6. If we choose number two the Checker prints the error message "!!! ERROR: New expression does not type-check properly."

Nevertheless, the new conclusion C1 is satisfied by H1.

```
CHECK|: done.
*** PROVED C1: sorted(list__OLD) FOR CASE 1
```

We start the second case. We re-use H16, H19, H20 and H21 for the second application of priority_list(8). We must infer the hypothesis for Condition 4 of priority_list(8).

```
CASE 2: not i = buf__ext
    *** New H29: not i = buf__ext
    >>> New goal C1: sorted(update(update(list__OLD, [element_index],
            mk__buf__priority_element(pri := priority, suc := fld_suc(element(
            list__OLD, [i])))), [i], upf_suc(element(update(list__OLD, [
            element_index], mk__buf__priority_element(pri := priority, suc
            := fld_suc(element(list__OLD, [i])))), [i]), element_index)))
```

We make use of the existence of predicate loop_condition in hypothesis H2 by applying priority_list(16).

```
    CHECK|: infer fld_pri(element(list__OLD,[i])) < priority using priority_list(16) from [2,30].
    *** New H31: fld_pri(element(list__OLD, [i])) < priority
```

Hypothesis H31 is the basis for Condition 4 of priority_list(8). The replacement of the predicate priority is justified by H24.

```
    CHECK|: replace h#31: priority by fld_pri(element(update(list__OLD,[element_index],
            mk__buf__priority_element(pri := priority,
            suc := fld_suc(element(list__OLD,[i]))))),[element_index])) using eq(1).
    NEW EXPRESSION: fld_pri(element(list__OLD, [i])) < fld_pri(element(update(
            list__OLD, [element_index], mk__buf__priority_element(pri := priority,
            suc := fld_suc(element(list__OLD, [i])))), [element_index]))
```

Before we can modify H31 further, we must infer another hypothesis.

```
    CHECK|: standardise fld_pri(element(update(list__OLD,[element_index],
            mk__buf__priority_element(pri := priority,suc := fld_suc(element(list__OLD,[i])))),[i])).
    *** New H32: fld_pri(element(update(list__OLD, [element_index],
            mk__buf__priority_element(pri := priority, suc := fld_suc(element(
            list__OLD, [i])))), [i])) = fld_pri(element(update(list__OLD, [
            element_index], mk__buf__priority_element(pri := priority, suc
            := fld_suc(element(list__OLD, [i])))), [i]))
    CHECK|: replace h#32: element(update(A,J,X),K) by element(A,K) using array(3).
    Change which occurrence (number/none/all)? |: 1.
    NEW EXPRESSION: fld_pri(element(list__OLD, [i])) = fld_pri(element(update(
            list__OLD, [element_index], mk__buf__priority_element(pri := priority,
            suc := fld_suc(element(list__OLD, [i])))), [i]))
```

The application of array(3) is justified by the information element_index <> i from hypothesis H23. With H23 we can modify H31 further, which is then Condition 4 of priority_list(8).

```
CHECK|: replace h#31: fld_pri(element(list__OLD,[i])) by fld_pri(element(update(list__OLD,
        [element_index],mk__buf__priority_element(pri := priority,
        suc := fld_suc(element(list__OLD,[i]))),[i])) using eq(1).
NEW EXPRESSION: fld_pri(element(update(list__OLD, [element_index],
        mk__buf__priority_element(pri := priority, suc := fld_suc(element(
        list__OLD, [i]))), [i])) < fld_pri(element(update(list__OLD, [
        element_index], mk__buf__priority_element(pri := priority, suc
        := fld_suc(element(list__OLD, [i]))), [element_index]))
```

The necessary conditions to apply priority_list(8) are H16, H19, H20, H21 and H31.

```
CHECK|: replace c#1: sorted(update(L,[I],upf_suc(element(L,[I]),E))) by sorted(L) using
        priority_list(8).
NEW EXPRESSION: sorted(update(list__OLD, [element_index],
        mk__buf__priority_element(pri := priority, suc := fld_suc(element(
        list__OLD, [i]))))
```

C1 can be simplified further with priority_list(11). This is justified by H11. We are nearly done with C1.

```
CHECK|: replace c#1: sorted(update(L,[element_index],_)) by sorted(L) using
        priority_list(11).
Possible replacements for sorted(update(list__OLD, [element_index],
        mk__buf__priority_element(pri := priority, suc := fld_suc(element(
        list__OLD, [i])))) are:
    1. sorted(list__OLD)
           according to rule priority_list(11)
    2. sorted(update(update(list__OLD, [element_index],
        mk__buf__priority_element(pri := priority, suc := fld_suc(element(
        list__OLD, [i]))), [element_index], _10))
           according to rule priority_list(11)
Select (number/none): |: 1.
NEW EXPRESSION: sorted(list__OLD)
CHECK|: done.
*** PROVED C1: sorted(list__OLD) FOR CASE 2
```

Again the second option of the question makes no sense and would lead to an error message by the Checker.

We eliminated C1 and go on to C2. We can re-use hypotheses H16, H19 and H20 to apply priority_list(4) to C2.

```
CHECK|: replace c#2: l(update(L,[I],upf_suc(element(L,[I]),E))) by l(L) \/ (set [E]) using
        priority_list(4).
NEW EXPRESSION: l(update(list__OLD, [element_index],
        mk__buf__priority_element(pri := priority, suc := fld_suc(element(
        list__OLD, [i])))) \/ (set [element_index]) = l(list__OLD) \/
        (set [element_index])
```

We apply priority_list(11), which is justified by H11, and finish the proof of this verification condition.

```
CHECK|: replace c#2: I(update(L,[element_index],_)) by I(L) using priority_list(7).
Possible replacements for I(update(list__OLD, [element_index],
        mk__buf__priority_element(pri := priority, suc := fld_suc(element(
        list__OLD, [i])))))) are:
    1. I(list__OLD)
           according to rule priority_list(7)
    2. I(update(update(list__OLD, [element_index],
        mk__buf__priority_element(pri := priority, suc := fld_suc(element(
        list__OLD, [i]))))), [element_index], _12))
           according to rule priority_list(7)
Select (number/none): |: 1.
NEW EXPRESSION: I(list__OLD) \/ (set [element_index]) = I(list__OLD) \/
        (set [element_index])
CHECK|: done.
*** PROVED C2: I(list__OLD) \/ (set [element_index]) = I(list__OLD) \/
        (set [element_index])
*** VC PROVED -- Well done!
```

Proof Session For Refinement Integrity

```
CHECK|: list.
H1: element_index not_in I(fld_list(state))
H9: fld_pri(element(fld_list(state), [buf__ext])) =
        buf__system_priority__last
-->
  C1: fld_pri(element(fld_list(state), [0])) = 9223372036854775807
  C2: not element_index in I(fld_list(state))
```

We eliminate C1 by replacing the constants buf__system_priority__last and buf__ext.

```
CHECK|: infer 0 = buf__ext using enter_rules(3).
*** New H10: 0 = buf__ext
CHECK|: replace c#1: 0 by buf__ext using eq(1).
NEW EXPRESSION: fld_pri(element(fld_list(state), [buf__ext])) = 9223372036854
        775807
CHECK|: infer 9223372036854775807 = buf__system_priority__last using enter_rules(21).
*** New H11: 9223372036854775807 = buf__system_priority__last
CHECK|: replace c#1: 9223372036854775807 by buf__system_priority__last using eq(1).
NEW EXPRESSION: fld_pri(element(fld_list(state), [buf__ext])) =
        buf__system_priority__last
CHECK|: done.
*** PROVED C1: fld_pri(element(fld_list(state), [buf__ext])) =
        buf__system_priority__last
```

We rewrite C2 such that the Checker recognizes the correlation between H1 and C2.

```
CHECK|: replace c#2: not (X in A) by X not_in A using sets(2).
NEW EXPRESSION: element_index not_in l(fld_list(state))
CHECK|: done.
*** PROVED C2: element_index not_in l(fld_list(state))
*** VC PROVED -- Well done!
```

There is one other verification condition to prove. The constants have to be replaced in analogy to C1. We do not show this here.

6.4 Protected Type Dynamic_Priority

Procedure Increment is the main operation of the protected type Dynamic_Priority. It increments the dynamic priority. Ideally, the dynamic priority value would be of an unbounded integer type. But Ada provides no unbounded integer type, so the priority value will definitely overflow after more than 2^{31} years of run time. Ada does provide circular integer types but these would destroy the semantics of the priority values.

SPARK has no support for an unbounded integer type in its annotations, either. The Examiner would automatically generate verification conditions to prevent an overflow of the dynamic priority value. But the priority value will definitely overflow. We have no other choice but to hide the incrementation from SPARK with the hide annotation. Hiding the main operation leaves us with nothing to do, so we choose to omit the proof annotations for the protected type Dynamic_Priority in the first place.

Chapter 7

Conclusion

The specification, design and implementation of Coco turned out to be mostly straightforward (Chapters 2 and 5). Concurrent Pascal and Ada have a very similar semantics. Ada allowed us to lift some design restrictions which were imposed by Concurrent Pascal.

We gave an overview of the Spark tools Examiner, Simplifier and Checker (Section 3.3). We laid the foundations for the understanding of the subsequent chapters by introducing the Spark annotations, the FDL language and the proof rule language (Chapter 3).

We discussed why RavenSpark lags behind sequential Spark (Section 4.1). The main obstacles are the missing proof annotations in protected types and the restricted task model.

We proposed a different approach to protected types in Spark that is not based on the Ravenscar Profile (Section 4.2). We call it PassauSpark.

To be able to apply our approach to the proof of Coco, we implemented a preprocessor for the Spark tools (Section 4.3).

Spark does not support abstract data structures like sets, lists, trees or unbounded integers in the annotations. To a certain degree it is possible to work around this with proof functions. We did this for set types (Section 4.5).

We specified pre- and post-conditions for the operations of the protected types Buffer_Guard and Priority_List. We declared proof functions and defined proof rules for them (Chapter 6).

We proved in proof sessions with the Checker that the Buffer_Guard manages the list of free frames as intended. Frames cannot be occupied and free at the same time. The Buffer_Guard always knows the state of all frames. We proved that the Priority_List works as a priority queue: the queue is always ordered and the element with the highest priority is always at the top. We also proved that the linked-list structure of

the queue is really a linked list and does not contain inner loops. Moreover, SPARK enabled us to find a missing pre-condition in the original manual proof of COCO.

Appendix A

Source Code

A.1 The COCO System

```
package Conf is
   Buffer_Size : constant := 65536;   -- number of buffer cells
   Num_Ports : constant := 8;         -- number of hardware I/O ports (also needs to be
                                      -- adjusted in file main.adb)
   Min_Prio : constant := 63;         -- lowest packet priority
   Max_Length : constant := 1500;     -- maximum size of packet in bytes
   Timer_Delay : constant := 0.01;    -- tick frequency (100 Hz)
end Conf;

package Stacks is

   type Stack is private;

   subtype Stack_Range is Positive range 1 .. Positive'Last - 1;

   --# function Ptr (S : in Stack) return Integer;
   --# function Max_Ptr (S : in Stack) return Integer;
   --# function Ptr_In_Range (S : in Stack) return Boolean;

   function Empty (S : in Stack) return Boolean;
   --# return Empty (S);

   function Full (S : in Stack) return Boolean;
   --# return Full (S);

   procedure Initialize (S : out Stack; Max : in Stack_Range);
   --# derives S from Max;
   --# post Ptr_In_Range (S) and Max_Ptr (S) = Max and Ptr (S) = Max + 1;

   procedure Push (S : in out Stack; P : out Integer);
   --# derives P, S from S;
   --# pre Ptr_In_Range (S) and not Full (S);
   --# post Ptr_In_Range (S) and Max_Ptr (S) = Max_Ptr (S~) and
```

```
   --#      Ptr (S) = Ptr (S~) + 1 and P = Ptr (S~);

   procedure Pop (S : in out Stack; P : out Integer);
   --# derives P, S from S;
   --# pre Ptr_In_Range (S) and not Empty (S);
   --# post Ptr_In_Range (S) and Max_Ptr (S) = Max_Ptr (S~) and
   --#      Ptr (S) = Ptr (S~) - 1 and P = Ptr (S);

private
   type Stack is record
      Pointer : Positive;
      Max : Stack_Range;
   end record;
end Stacks;

package body Stacks is

   function Empty (S : in Stack) return Boolean is
   begin
      return (S.Pointer = 1);
   end Empty;

   function Full (S : in Stack) return Boolean is
   begin
      return (S.Pointer = S.Max + 1);
   end Full;

   procedure Initialize (S : out Stack; Max : in Stack_Range) is
   begin
      S := Stack'(Max + 1, Max);
   end Initialize ;

   procedure Push (S : in out Stack; P : out Integer) is
   begin
      P := S.Pointer;
      S.Pointer := S.Pointer + 1;
   end Push;

   procedure Pop (S : in out Stack; P : out Integer) is
   begin
      S.Pointer := S.Pointer - 1;
      P := S.Pointer;
   end Pop;

end Stacks;

with Conf, Stacks;
--# inherit Conf, Stacks;
package Buf
--# own Shared_Buffer;
--# initializes Shared_Buffer;
```

```
is
   subtype Port_Adr is Positive range 1 .. Conf.Num_Ports;

   subtype Packet_Priority is Natural range 0 .. Conf.Min_Prio;

   subtype Data_Length is Natural range 0 .. Conf.Max_Length;

   subtype Data_Index is Positive range 1 .. Conf.Max_Length;

   type Byte is range 0 .. 255;
   for Byte'Size use 8;

   type Packet_Data is array (Data_Index) of Byte;

   type Packet is record
      Dest : Port_Adr;
      Prio : Packet_Priority ;
      Length : Data_Length;
      Data : Packet_Data;
   end record;

   subtype Buffer_Index is Positive range 1 .. Conf.Buffer_Size;

   type Packet_Array is array (Buffer_Index) of Packet;

   Shared_Buffer : Packet_Array;

   type Buffer_Index_Array is array (Buffer_Index) of Buffer_Index;

   protected type Buffer_Guard
   --# own State : State_Type;
   --# invariant Buffer_Guard_Invariant (State);
   is
      --# type State_Type is abstract;

      --# type Set_Type is abstract;
      --# function Set_Member (S : in Set_Type; E : in Buffer_Index) return Boolean;
      --# function Set_Delete (S : in Set_Type; E : in Buffer_Index) return Set_Type;
      --# function Set_Insert (S : in Set_Type; E : in Buffer_Index) return Set_Type;
      --# function Empty_Set return Set_Type;

      --# function F (Free_List : in Buffer_Index_Array;
      --#             Pointer: in Stacks.Stack_Range) return Set_Type;

      --# function Buffer_Guard_Invariant (S : in State_Type) return Boolean;
      --# function Can_Be_Released (S : in State_Type; E : in Buffer_Index) return Boolean;

      procedure Initialize ;
      --# global in out State;
      --# derives State from State;
```

```
   entry Request (Element_Index : out Buffer_Index);
   --# global in out State;
   --# derives Element_Index, State from State;
   procedure Release (Element_Index : in Buffer_Index);
   --# global in out State;
   --# derives State from Element_Index, State;
   --# pre Can_Be_Released (State, Element_Index);
private
   Free_List : Buffer_Index_Array;
   Stack : Stacks.Stack;
end Buffer_Guard;

Ext : constant := 0;

subtype List_Index is Natural range Ext .. Conf.Buffer_Size;

type Priority_Element is private;

type Priority_Queue is array (List_Index) of Priority_Element;

subtype System_Priority is Long_Long_Integer range 0 .. Long_Long_Integer'Last;

protected type Priority_List
--# own State : State_Type;
--# invariant Priority_List_Invariant (State);
is
   --# type State_Type is abstract;

   --# type Set_Type is abstract;
   --# function Set_Member (S : in Set_Type; E : in List_Index) return Boolean;
   --# function Set_Delete (S : in Set_Type; E : in List_Index) return Set_Type;
   --# function Set_Insert (S : in Set_Type; E : in List_Index) return Set_Type;
   --# function Set_Element (E : in List_Index) return Set_Type;
   --# function Empty_Set return Set_Type;

   --# function L (List : in Priority_Queue) return Set_Type;
   --# function Sorted (List : in Priority_Queue) return Boolean;
   --# function Min (List : in Priority_Queue) return System_Priority;
   --# function Loop_Condition (List : in Priority_Queue;
   --#                          Index : in List_Index) return Boolean;

   --# function Priority_List_Invariant (S : in State_Type) return Boolean;
   --# function Can_Be_Removed (S : in State_Type; E : in Buffer_Index) return Boolean;

   procedure Initialize ;
   --# global in out State;
   --# derives State from State;
   procedure Enter (Element_Index : in Buffer_Index; Priority : in System_Priority);
   --# global in out State;
   --# derives State from Element_Index, Priority, State;
```

```
      --# pre Can_Be_Removed (State, Element_Index);
      entry Remove (Element_Index : out Buffer_Index);
      --# global in out State;
      --# derives Element_Index from State & State from State;
   private
      List : Priority_Queue;
   end Priority_List ;

private
   type Priority_Element is record
      Pri : System_Priority;
      Suc : List_Index;
   end record;
end Buf;


package body Buf is

   protected body Buffer_Guard
   --# invariant Stacks.Ptr_In_Range (Stack) and Stacks.Max_Ptr (Stack) = Conf.Buffer_Size;
   is

      procedure Initialize
      --# global in out Free_List; out Stack;
      --# derives Free_List from Free_List & Stack from ;
      --# post (for all J in Buffer_Index => (Free_List (J) = J));
      is
      begin
         for I in Buffer_Index
         --# assert (for all J in Buffer_Index range 1 .. I - 1 => (Free_List (J) = J));
         loop
            Free_List (I) := I;
         end loop;
         Stacks.Initialize (Stack, Conf.Buffer_Size);
      end Initialize ;

      entry Request (Element_Index : out Buffer_Index) when not Stacks.Empty (Stack)
      --# global in out Stack; in Free_List;
      --# derives Element_Index from Free_List, Stack & Stack from Stack;
      --# post Set_Member (F (Free_List, Stacks.Ptr (Stack~)), Element_Index)
      --#      and F (Free_List, Stacks.Ptr (Stack))
      --#          = Set_Delete (F (Free_List, Stacks.Ptr (Stack~)), Element_Index);
      is
         Top : Buffer_Index;
      begin
         Stacks.Pop (Stack, Top);
         Element_Index := Free_List (Top);
      end Request;

      procedure Release (Element_Index : in Buffer_Index)
      --# global in out Free_List, Stack;
```

```
   --# derives Free_List from Element_Index, Free_List, Stack & Stack from Stack;
   --# pre not Set_Member (F (Free_List, Stacks.Ptr (Stack)), Element_Index)
   --#     and not Stacks.Full (Stack);
   --# post F (Free_List, Stacks.Ptr (Stack))
   --#      = Set_Insert (F (Free_List~, Stacks.Ptr (Stack~)), Element_Index);
   is
      Top : Buffer_Index;
   begin
      Stacks.Push (Stack, Top);
      Free_List (Top) := Element_Index;
   end Release;

end Buffer_Guard;

protected body Priority_List
--# invariant Sorted (List) and List (Ext). Pri = System_Priority'Last;
is

   procedure Initialize
   --# global in out List;
   --# derives List from List;
   --# post L (List) = Set_Element (Ext);
   is
   begin
      List (Ext). Pri := System_Priority'Last;
      List (Ext). Suc := Ext;
   end Initialize ;

   procedure Enter (Element_Index : in Buffer_Index; Priority : in System_Priority)
   --# global in out List;
   --# derives List from List, Element_Index, Priority;
   --# pre not Set_Member (L (List), Element_Index);
   --# post L (List) = Set_Insert (L (List~), Element_Index);
   is
      I : List_Index;
   begin
      I := Ext;
      while (List (List (I). Suc).Pri < Priority)
      --# assert Set_Member (L (List), I) and Loop_Condition (List, List (I). Suc)
      --#        and List~ = List;
      loop
         I := List (I). Suc;
      end loop;
      List (Element_Index) := Priority_Element'( Pri => Priority, Suc => List (I). Suc);
      List (I). Suc := Element_Index;
   end Enter;

   entry Remove (Element_Index : out Buffer_Index) when List (Ext).Suc /= Ext
   --# global in out List;
   --# derives Element_Index from List & List from List;
```

```
      --# post L (List) = Set_Delete (L (List~),  Element_Index)
      --#      and List (Element_Index).Pri = Min (List~) and Min (List) >= Min (List~);
      is
      begin
         Element_Index := List (Ext). Suc;
         List (Ext). Suc := List (List (Ext). Suc).Suc;
      end Remove;

   end Priority_List ;

begin
   --# hide Buf;
   null;
end Buf;

with Conf, Buf;
package Timer is

   protected type Dyn_Priority is
      procedure Initialize ;
      function Count return Buf.System_Priority;
      procedure Increment;
   private
      Dyn_Count : Buf.System_Priority;
   end Dyn_Priority;

   Dyn_Priority_Monitor : Dyn_Priority;

   task type Timer_Process is
   end Timer_Process;

end Timer;

package body Timer is

   protected body Dyn_Priority is

      procedure Initialize is
      begin
         Dyn_Count := 0;
      end Initialize ;

      function Count return Buf.System_Priority is
      begin
         return Dyn_Count;
      end Count;

      procedure Increment is
      begin
         Dyn_Count := Dyn_Count + 1;
      end Increment;
```

```
   end Dyn_Priority;

   task body Timer_Process is
   begin
      loop
         delay Conf.Timer_Delay;
         Dyn_Priority_Monitor. Increment;
      end loop;
   end Timer_Process;

end Timer;

with Buf, Timer;
package Tasks is

   Buffer_Guard_Monitor : Buf.Buffer_Guard;
   In_Priority_List_Monitor : Buf.Priority_List;
   Out_Priority_List_Monitor : array (Buf.Port_Adr) of Buf.Priority_List;

   task type Input_Process (Port : Buf.Port_Adr) is
   end Input_Process;

   task type Data_Process is
   end Data_Process;

   task type Output_Process (Port : Buf.Port_Adr) is
   end Output_Process;

end Tasks;

package body Tasks is

   task body Input_Process is
      Element_Index : Buf.Buffer_Index;
      Priority : Buf.System_Priority;
   begin
      loop
         -- listen at port specified by variable Port for incoming packet here
         Buffer_Guard_Monitor.Request (Element_Index);
         -- receive incoming packet into Buf.Shared_Buffer (Element_Index) here
         Priority := Buf.System_Priority (Buf.Shared_Buffer (Element_Index).Prio)
                        + Timer.Dyn_Priority_Monitor.Count;
         In_Priority_List_Monitor.Enter (Element_Index, Priority);
      end loop;
   end Input_Process;

   task body Data_Process is
      Element_Index : Buf.Buffer_Index;
      Priority : Buf.System_Priority;
      Destination : Buf.Port_Adr;
```

```
   begin
      loop
         In_Priority_List_Monitor.Remove (Element_Index);
         -- process element Buf.Shared_Buffer (Element_Index) here
         Destination := Buf.Shared_Buffer (Element_Index).Dest;
         Priority := Buf.System_Priority (Buf.Shared_Buffer (Element_Index).Prio)
                     + Timer.Dyn_Priority_Monitor.Count;
         Out_Priority_List_Monitor (Destination).Enter (Element_Index, Priority);
      end loop;
   end Data_Process;

   task body Output_Process is
      Element_Index : Buf.Buffer_Index;
   begin
      loop
         Out_Priority_List_Monitor (Port).Remove (Element_Index);
         -- send Buf.Shared_Buffer (Element_Index) as packet to port specified by variable Port here
         Buffer_Guard_Monitor.Release (Element_Index);
      end loop;
   end Output_Process;

end Tasks;


with Buf, Tasks, Timer;
procedure Main is
begin
   -- initialize protected types:
   Tasks.Buffer_Guard_Monitor.Initialize;
   Tasks.In_Priority_List_Monitor.Initialize;
   for I in Buf.Port_Adr loop
      Tasks.Out_Priority_List_Monitor (I).Initialize;
   end loop;
   Timer.Dyn_Priority_Monitor.Initialize;
   -- start tasks:
   declare
      Input_Process_1 : Tasks.Input_Process (1);
      Input_Process_2 : Tasks.Input_Process (2);
      Input_Process_3 : Tasks.Input_Process (3);
      Input_Process_4 : Tasks.Input_Process (4);
      Input_Process_5 : Tasks.Input_Process (5);
      Input_Process_6 : Tasks.Input_Process (6);
      Input_Process_7 : Tasks.Input_Process (7);
      Input_Process_8 : Tasks.Input_Process (8);

      Output_Process_1 : Tasks.Output_Process (1);
      Output_Process_2 : Tasks.Output_Process (2);
      Output_Process_3 : Tasks.Output_Process (3);
      Output_Process_4 : Tasks.Output_Process (4);
      Output_Process_5 : Tasks.Output_Process (5);
      Output_Process_6 : Tasks.Output_Process (6);
```

```
      Output_Process_7 : Tasks.Output_Process (7);
      Output_Process_8 : Tasks.Output_Process (8);

      Data_Process : Tasks.Data_Process;
      Timer_Process : Timer.Timer_Process;
   begin
      null; -- null statement as in RavenSPARK
   end;
end Main;
```

A.2 Proof Rules

```
rule_family setfun:
        set_insert(S,E) requires [ S:any, E:i ] &
        set_delete(S,E) requires [ S:any, E:i ] &
        set_member(S,E) requires [ S:any, E:i ] &
        set_element(E) requires [ E:i ] &
        empty_set requires [ ].

setfun(1): set_insert(S,E) may_be_replaced_by S \/ (set [E]).
setfun(2): set_delete(S,E) may_be_replaced_by S \ (set [E]).
setfun(3): set_member(S,E) may_be_replaced_by E in S.
setfun(4): set_element(E) may_be_replaced_by (set [E]).
setfun(5): empty_set may_be_replaced_by (set []).

rule_family stacks:
        ptr(S) requires [ S:any ] &
        max_ptr(S) requires [ S:any ] &
        ptr_in_range (S) requires [ S:any ] &
        empty(S) requires [ S:any ] &
        full(S) requires [ S:any ] &
        stacks__ptr_in_range(S) requires [ S:any ] &
        stacks__empty(S) requires [ S:any ] &
        stacks__full(S) requires [ S:any ].

stacks(1): ptr(S) may_be_replaced_by fld_pointer(S).
stacks(2): max_ptr(S) may_be_replaced_by fld_max(S).
stacks(3): ptr_in_range(S) may_be_replaced_by
            fld_pointer(S) >= 1 and fld_pointer(S) <= fld_max(S) + 1.
stacks(4): empty(S) may_be_replaced_by fld_pointer(S) = 1.
stacks(5): full(S) may_be_replaced_by fld_pointer(S) = fld_max(S) + 1.

stacks(6): stacks__ptr_in_range(S) may_be_replaced_by
            stacks__ptr(S) >= 1 and stacks__ptr(S) <= stacks__max_ptr(S) + 1.
stacks(7): stacks__empty(S) may_be_replaced_by stacks__ptr(S) = 1.
stacks(8): stacks__full(S) may_be_replaced_by stacks__ptr(S) = stacks__max_ptr(S) + 1.

rule_family buffer_guard:
        f(F,P) requires [ F:any, P:i ] &
```

```
        buffer_guard_invariant(S) requires [ S:any ] &
        can_be_released(S,E) requires [ S:any, E:i ] &
        A \/ B requires [ A:any, B:any ] &
        A \ B requires [ A:any, B:any ] &
        A in B requires [ A:any, B:any ] &
        element(A,B) requires [ A:any, B:any ] &
        A < B requires [ A:ire, B:ire ].

buffer_guard(1): f(F,P) \/ (set [E]) may_be_replaced_by f(update(F,[P],E),P+1)
                 if [ E not_in f(F,P) ].
buffer_guard(2): f(F,P) \ (set [element(F,[P−1])]) may_be_replaced_by f(F,P−1)
                 if [ element(F,[P−1]) in f(F,P) ].

buffer_guard(3): element(F,[J]) in f(F,P) may_be_deduced_from [ J>=1, J<P ].

buffer_guard(4): buffer_guard_invariant(S) may_be_replaced_by
                 stacks__ptr_in_range(fld_stack(S))
                 and stacks__max_ptr(fld_stack(S)) = conf__buffer_size.

buffer_guard(5): can_be_released(S,E) may_be_replaced_by
                 E not_in f(fld_free_list(S),stacks__ptr(fld_stack(S)))
                 and not stacks__full(fld_stack(S)).


rule_family priority_list:
        l(L) requires [ L:any ] &
        sorted(L) requires [ L:any ] &
        loop_condition(L,I) requires [ L:any, I:any ] &
        min(X) requires [ X:any ] &
        priority_list_invariant(S) requires [ S:any ] &
        can_be_removed(S,E) requires [ S:any, E:i ] &
        A \/ B requires [ A:any, B:any ] &
        A \ B requires [ A:any, B:any ] &
        A in B requires [ A:any, B:any ] &
        set A requires [ A:any ] &
        element(A,B) requires [ A:any, B:any ] &
        update(A,B,C) requires [ A:any, B:any, C:any ] &
        A < B requires [ A:ire, B:ire ] &
        for_all(A,B) requires [ A:any, B:any ].

priority_list(1): buf__ext in l(L) may_be_deduced.
priority_list(2): fld_suc(element(L,[I])) in l(L) may_be_deduced_from [ I in l(L) ].
priority_list(3): for_all(x : integer, x in l(L) −> I <> x) may_be_deduced_from [ I not_in l(L) ].
priority_list(4): l(L) \/ (set [E]) & l(update(L,[I],upf_suc(element(L,[I]),E)))
                  are_interchangeable
                  if [ I in l(L), E not_in l(L), fld_suc(element(L,[E])) = fld_suc(element(L,[I])) ].
priority_list(5): l(L) \ (set [E]) & l(update(L,[I],upf_suc(element(L,[I]),fld_suc(element(L,[E])))))
                  are_interchangeable
                  if [ I in l(L), E in l(L), fld_suc(element(L,[I])) = E ].
priority_list(6): l(update(_, [buf__ext], upf_suc(_, buf__ext))) & (set [buf__ext])
                  are_interchangeable.
```

```
priority_list(7): l(update(L,[l],_)) & l(L) are_interchangeable if [ l not_in l(L) ].

priority_list(8): sorted(update(L,[l],upf_suc(element(L,[l]),E))) & sorted(L)
                  are_interchangeable
                  if [ l in l(L), E not_in l(L),
                       fld_suc(element(L,[E])) = fld_suc(element(L,[l])),
                       (l = buf__ext or fld_pri(element(L, [l])) <= fld_pri(element(L,[E]))),
                       fld_pri(element(L,[E])) <= fld_pri(element(L,[fld_suc(element(L,[l]))]))
                  ].
priority_list(9): sorted(update(L,[l],upf_suc(element(L,[l]),fld_suc(element(L,[E])))))
                  may_be_replaced_by sorted(L)
                  if [ l in l(L), E in l(L), fld_suc(element(L,[l])) = E ].
priority_list(10): sorted(update(_,[buf__ext],upf_suc(_,buf__ext))) may_be_deduced.
priority_list(11): sorted(update(L,[l],_)) & sorted(L) are_interchangeable if [ l not_in l(L) ].
priority_list(12): for_all(x : integer, x in l(L) and x <> buf__ext ->
                     fld_pri(element(L,[x])) <= fld_pri(element(L,[fld_suc(element(L,[x]))])))
                   may_be_deduced_from [ sorted(L) ].

priority_list(13): min(L) may_be_replaced_by
                   fld_pri(element(L,[fld_suc(element(L,[buf__ext]))])) if [ sorted(L) ].

priority_list(14): loop_condition(L,fld_suc(element(L,[buf__ext]))) may_be_deduced.
priority_list(15): loop_condition(L,fld_suc(element(L,[l]))) may_be_deduced_from
                   [ loop_condition(L,l), fld_pri(element(L,[l])) < priority, l <> buf__ext ].
priority_list(16): fld_pri(element(L,[l])) < priority may_be_deduced_from
                   [ loop_condition(L,fld_suc(element(L,[l]))), l <> buf__ext ].

priority_list(17): priority_list_invariant(S) may_be_replaced_by
                   sorted(fld_list(S))
                   and fld_pri(element(fld_list(S), [buf__ext])) = buf__system_priority__last.

priority_list(18): can_be_removed(S,E) may_be_replaced_by E not_in l(fld_list(S)).
```

A.3 Examiner Configuration

```
package Standard is
   type Integer is range −2147483648 .. 2147483647;
   type Long_Long_Integer is range −9223372036854775808 .. 9223372036854775807;
end Standard;
```

A.4 Checker Configuration

```
set auto_done to on.
set simplify_in_infer to on.
```

A.5 Additional Material

Additional material of this work can be downloaded from the following World Wide Web address:

`http://www.fmi.uni-passau.de/~haller/coco.tar.gz`

The archive contains

- the source code of Coco,
- the source code of the pre-processor with pre-generated scanner source code,
- the pre-processed source code of Coco,
- the output and log files generated by the Examiner,
- the output and log files generated by the Simplifier,
- the proof command logs for the Checker,
- the output and log files generated by the Checker and
- the environment configuration files.

Bibliography

[ACT05] Ada Core Technologies, Inc. *GNAT GPL Edition*, 2005. `http://libre.adacore.com/`.

[Ada95] International Standard. *Ada 95 Reference Manual: Language and Standard Libraries*. ISO/IEC-8652:1995(E) with COR.1:2000.

[ALT01] Ada for Linux Team. *aflex and ayacc*, 2001. Archived: `http://web.archive.org/web/20040806234051/www.gnuada.org/rpms313p.html`.

[Bar03] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.

[BB+95] J. Barnes, B. Brosgol, et al. *Ada 95 Rationale: The Language – The Standard Libraries*. Intermetrics, Inc., 733 Concord Ave, Cambridge, MA 02138, Jan 1995.

[BDV03] A. Burns, B. Dobbing, and T. Vardanega. *Guide for the use of the Ada Ravenscar Profile in high integrity systems*. Technical Report YCS-2003-348, University of York, Jan 2003.

[BH75] P. Brinch-Hansen. *The programming language Concurrent Pascal*. IEEE Trans. on Software Engineering, 1(2):199–207, Jun 1975.

[Hoa74] C. A. R. Hoare. *Monitors: An Operating System Structuring Concept*. Comm. ACM, 17(10):549–557, Oct 1974.

[How76] J. H. Howard. *Signaling in Monitors*. Proc. 2nd Int. Conf. on Software Engineering (ICSE), pages 47–52, San Francisco, Cal., Oct 1976. IEEE Computer Society.

[Kee79] J. L. Keedy. *On Structuring Operating Systems With Monitors*. ACM SIGOPS Operating Systems Review, 13(1):5–9, Jan 1979.

[Len77] C. Lengauer. *Strukturierter Betriebssystementwurf mit Concurrent PASCAL*. Diploma thesis, Technical Report HMI-B 236, Hahn-Meitner-Institut für Kernforschung Berlin GmbH, Jul 1977.

[Pra06] Praxis High Integrity Systems Ltd. *SPARKAda homepage*, 2006. `http://www.praxis-his.com/sparkada/`.

[Spa04a] SPARK Team. *SPADE Proof Checker: Rules Manual*, Issue 5.2. Praxis High Integrity Systems Ltd., 20 Manvers Street, Bath BA1 1PX, UK, Nov 2004.

[Spa04b] SPARK Team. *SPADE Proof Checker: User Manual*, Issue 3.4. Praxis High Integrity Systems Ltd., 20 Manvers Street, Bath BA1 1PX, UK, Nov 2004.

[Spa05] SPARK Team. *SPARK Examiner: The SPARK Ravenscar Profile*, Issue 1.4. Praxis High Integrity Systems Ltd., 20 Manvers Street, Bath BA1 1PX, UK, Nov 2005.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet habe, sowie dass diese Diplomarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt wurde.

Passau, den 29. März 2006

(Michael Haller)