Abstract Software Specifications and Automatic Proof of Refinement*

Claire Dross and Yannick Moy

AdaCore, 46 rue d'Amsterdam, F-75009 Paris (France)
{dross, moy}@adacore.com

Abstract. It is common practice in critical software development, and compulsory in railway software developed according to EN 50128 standard, to separate software specification from software implementation. Verification activities should be performed to ensure that the latter is a correct refinement of the former. When the specification is formalized, for example in B method, the refinement relation can even be formally proved. In this article, we present how a similar proof of refinement can be performed at the level of the programming language used for implementation, using the SPARK technology. We describe two techniques to specify abstractly the behavior of a software component in terms of mathematical structures (sequences, sets and maps) and a methodology based on the SPARK tools to prove automatically that an efficient imperative implementation is a correct refinement of the abstract specification.

Keywords Formal methods, Verification and validation, Certification, Dependability, EN 50128

1 Introduction

The railway standard EN 50128 [1] has been the first one in 2001 to recommend formal methods for the development of critical software, an example later followed by other domains such as avionics [17]. In EN 50128, formal methods are recommended at levels SIL 1 and SIL2, and highly recommended at levels SIL3 and SIL4, both for software requirements (table A.2 of EN 50128) and for design and implementation (table A.4 of EN 50128). Among formal methods, formal proof is similarly (highly) recommended at the same levels for verification and testing (table A.5 of EN 50128). Formal proof based on contracts is a particularly good fit to the principles of high integrity software development enumerated at the start of EN 50128 document, as it allows modular verification of individual components with a clear description of dependences between components given by their contract.

* Work partly supported by the Joint Laboratory ProofInUse (ANR-13-LAB3-0007, http://www.spark-2014.org/proofinuse) and project VECOLIB (ANR-14-CE28-0018) of the French national research organization.

Subprogram contracts were popularized in the Design-by-Contract approach [20] as a means to separate responsibilities in software between a caller and a callee. The callee's *precondition* states the responsibility of its caller, while the callee's *postcondition* states the responsibility of the callee itself. For example, the following (incomplete) contract for procedure `Swap` specifies that it should be called with index parameters within the range of the array parameter, and that `Swap` will ensure on return that the corresponding values in the array have been swapped. Attribute `Old` in the postcondition is used to refer to values on entry to the subprogram.

```
procedure Swap (A : in out Arr; X, Y : Idx) with
  Pre  => X in A'Range and Y in A'Range,
  Post => A(X) = A(Y)'Old and A(Y) = A(X)'Old;
```

The procedure declaration above is written in SPARK, a subset of the Ada programming language targeted at safety- and security-critical applications. SPARK builds on the strengths of Ada for creating highly reliable and long-lived software. SPARK restrictions ensure that the behavior of a SPARK program is unambiguously defined, and simple enough that formal verification tools can perform an automatic diagnosis of conformance between a program specification and its implementation. The SPARK language and toolset for formal verification has been applied over many years to on-board aircraft systems, control systems, cryptographic systems, and rail systems [5,21]. The latest version, SPARK 2014 [12,19], builds on the new specification features added in Ada 2012 [4], so formal specifications are now understood by the usual development tools and can be executed. SPARK toolset was qualified as a verification tool (tool class T2) in a railway certification project subject to EN 50128 standard.

Compared to previous versions, the latest version of SPARK is used industrially to prove automatically both absence of run-time errors and properties of programs expressed as contracts. Contracts are mostly used to express low-level specifications, close to the actual implementation, like the one on `Swap` above (although they can be much more complex). In comparison, the B method [2] used in railway industry allows expressing specifications abstractly in terms of mathematical sequences, sets and maps (the abstract machine), while the implementation uses a restricted subset of B called B0 that provides a thin layer over concrete arrays and machine integers/floats (the concrete machine) in order to allow generation of efficient machine code. The proof that the concrete machine refines the abstract machine gives the confidence that the code indeed implements an abstract specification more easily understood by humans and shared among stakeholders. In this article, we show how a similar expression of abstract specifications is possible in SPARK, and that the proof of refinement can be performed automatically.

1.1 SPARK Verification Environment

Key Language Features The most useful feature in SPARK is the ability to specify a contract on subprograms, given by a precondition and a postcondition

as presented on `Swap`. Attribute `Result` in the postcondition of a function is used to refer to the value returned by the function.

Instead of preconditions and postconditions, or in addition to them, subprogram contracts may be specified by a set of disjoint and complete cases. For example, the following contract for procedure `Swap` states separate sub-contracts for the cases where the elements at indexes `X` and `Y` are equal or different. The first case specifies that, if `A(X)` equals `A(Y)` on entry, then `A` should not be modified by the call. The second case specifies that, if `A(X)` is different from `A(Y)` on entry, then `A` should be modified by the call.

```
procedure Swap (A : in out Arr; X, Y : Idx) with
  Contract_Cases =>
    (A(X) = A(Y) => A = A'Old,
     A(X) /= A(Y) => A /= A'Old);
```

Specific kinds of expressions make it easier to express contracts. If-expressions and case-expressions are the expression forms which correspond to the usual if-statements and case-statements. Note that an if-expression without an else-part `(if A then B)` expresses a logical implication of `B` by `A`. Quantified expressions `(for all X in A => P)` and `(for some X in A => P)` correspond to the mathematical universal and existential quantifications, only on a bounded domain. Expression functions define a function using a single expression, like in functional programming languages. As expression functions can be part of the specification of programs (contrary to regular function bodies) in SPARK, they provide a powerful way to abstract complex parts of contracts.

The second most useful feature in SPARK (after contracts) is the ability to specify properties of loops. A loop invariant expresses the cumulated effect of the loop up to that point. For example, the following loop invariant expresses that the array `A` has been zeroed-out up to the current loop index `J`, and that the rest of the array has not been modified. Attribute `Loop_Entry` is used to refer to values on entry to the loop.

```
pragma Loop_Invariant
  (for all K in A'Range =>
    (if K <= J then A(K) = 0 else A(K) = A'Loop_Entry(K)));
```

To show loop termination, one can use a loop variant to express that a quantity varies monotonically at each iteration of the loop. For example, the following loop variant expresses that scalar variable `J` increases at each loop iteration.

```
pragma Loop_Variant (Increases => J);
```

For-loops in SPARK are bounded by construction, so this is only needed for while-loops and plain-loops.

Benefits of Executable Contracts Traditionally, contracts have been interpreted quite differently depending on whether they were used for run-time assertion checking or for formal program verification. For run-time assertion checking, contracts have been interpreted as assertions on entry and exit of subprograms. For formal program verification, assertions have typically been interpreted as formulas in classical first-order logic. This was the situation with SPARK prior

to SPARK 2014. Practitioners have struggled with this interpretation, which was not consistent with the run-time assertion checking semantics [8].

SPARK reconciles the logic semantics and executable semantics of contracts, so users can now execute contracts, debug them like code, and test them when formal verification is too difficult to achieve. Furthermore, by keeping the annotation language the same as the programming language, users don't have to learn another language.

All the previously presented contracts and assertion pragmas lead to run-time assertions. If a property is not satisfied at run time, an exception is raised with a message indicating the failing property, for example on procedure `Swap`:

```
failed precondition from swap.ads:4
```

Another key benefit of executable contracts is that they can be used by other tools working at the level of code. For example, the CodePeer[1] static analysis tool uses contracts and assertion pragmas to issue more precise messages. Most notably, this also allows SPARK users to combine the results of formal verification and testing, when only part of a program is formally analyzed [10].

Key Tool Features GNATprove is the formal verification tool that analyzes SPARK code. It performs two different analyses: (i) flow analysis of the program and (ii) proof of program properties.

Flow analysis checks correct access to data in the program: correct access to global variables and correct access to initialized data. It is a fast static analysis (analysis time typically comparable with compilation time).

Proof is used to demonstrate that the program is free from run-time errors, and that the specified contracts are correctly implemented. It internally generates mathematical formulas for each property, that are given to the automatic provers Alt-Ergo, CVC4 and Z3. If one of the automatic provers manages to prove the formula in the given time, then the property is known to hold. Otherwise, more work is required from the user to understand why the property is not proved.

As proof requires interactions between the user and the tool until the specification can be proved automatically, the efficiency and the granularity at which the tool can be applied are critical. For efficiency, GNATprove uses a compilation-like model where only those parts that are impacted by a change need to be reanalyzed, and a fast generation of formulas. For convenient interaction, GNATprove allows users to focus on a single unit, a single subprogram inside a unit, or even a single line inside a subprogram.

A very useful feature of GNATprove to investigate unproved properties is its ability to display counterexamples along paths that lead to unproved properties. The counterexample and the path can be displayed in GPS[2] or in Eclipse[3], the two Integrated Development Environments which support SPARK. The user can

[1] `http://www.adacore.com/codepeer`
[2] `http://www.adacore.com/gnatpro/toolsuite/gps/`
[3] `http://www.adacore.com/gnatpro/toolsuite/gnatbench/`

also change the parameters of the tool to perform more precise proofs, at the expense of longer analysis time.

Finally, modular verification based on contracts can very easily exploit multi-core architectures, as the generation of Verification Conditions (VCs) for different units, or the proof of different VCs, can both be run in parallel. Typically, projects contain hundreds of units, and lead to the generation of thousands of VCs, which can be run by GNATprove on as many cores as are available. Note also that GNATprove uses file timestamps to avoid re-generating VCs for units which have not been updated, and file hashes to avoid re-proving VCs that have already been proved. This is crucial when developing either the code or the associated annotations, to avoid unnecessary rework.

1.2 Ghost Code in SPARK

Sometimes the variables and functions that are useful for the implementation are not sufficient to specify a property in contracts. One approach is to introduce additional variables and functions, which will then only be used for the purpose of verification. But in a certification context such as EN 50128, the additional code will need to be verified at the same level as the application. This means performing structural coverage analysis, showing traceability to requirements, and demonstrating absence of interference between this verification-related code and the rest of the program if the verification code is to be deactivated in the final executable. A better solution is to use so-called *ghost code*.

Ghost code is identified through an aspect named Ghost that can be attached to variables, types, subprograms and packages to indicate that these entities are only used in verification code. The compiler checks that such code indeed only appears in contracts, assertions, and the definition of other ghost entities. As a benefit, any unintended interference between verification-related and application code is caught automatically, and the verification code can be removed when the final executable is built (hence the name *ghost code*).

Various kinds of ghost code are useful in different situations:

- Ghost functions can express properties used in contracts.
- Global ghost variables can keep track of the current state of a program, or maintain a log of past events. This information can then be referenced in contracts.
- Ghost types are types that are only used for defining ghost variables.

In a SPARK context, the GNATprove tool will check additionally that ghost code cannot have any effect on the behavior of the program. For an overview of the possible uses of ghost code in SPARK, see the SPARK User's Guide[4] and for the detailed rules defining ghost code, see the SPARK Reference Manual[5].

[4] `http://docs.adacore.com/spark2014-docs/html/ug/spark_2014.html#ghost-code`

[5] `http://docs.adacore.com/spark2014-docs/html/lrm/subprograms.html#ghost-entities`

1.3 SPARK Library of Containers

Functional containers are part of the newly redesigned library of standard containers in SPARK. They consist in sequences, sets and maps. Functional containers are specified through a simple API with contracts, based on a few essential functions. For example, the API of functional sets is defined over its effects on the `Mem` function for membership. Here is the contract of function `Inc` that tests inclusion of set `S1` in set `S2`:

```
function Inc (S1, S2 : Set) return Boolean with
  Post => Inc'Result = (for all E in S1 => Mem (S2, E));
```

Quantification over a container content is achieved by means of a generic mechanism in SPARK, which allows users to describe the functions used to iterate over a given datatype. Similarly, the API of functional sequences is defined over its effects on the functions `Length` and `Get`, and the API of functional maps is defined over its effects on the functions `Mem` and `Get`.

SPARK also comes with a library of imperative containers (lists, vectors, sets and maps). In the newly redesigned library, imperative containers are specified through an API with contracts based on functional containers. The benefit of this approach is that there is no need for a dedicated support for containers in the SPARK tools or provers, as they are specified through contracts like any other piece of code.

Naturally, client code that uses imperative containers can be specified using functional containers, and GNATprove can be used to prove that the code implements its specification. In this article, we aim at showing that functional containers can be used to express abstract specifications even when the implementation does not use imperative containers, and that the refinement proof can nonetheless be made automatically with GNATprove. Thus, we are using only functional containers in the rest of this article, in a way that is reminiscent of their use in the contracts of imperative containers.

2 Extracting a Model From the Implementation

As initial example, we consider a simple (inefficient) memory allocator that maintains an array of boolean flags to indicate whether the nth resource is allocated or not. For the purpose of better explaining how a given way of writing specifications is adapted to specific situations, we will present first the implementation and only then the specification. In actual software development, the order would be reversed.

2.1 A Simple Memory Allocator

In fact, in SPARK we can use an enumeration `Status` instead of a boolean, and an array `Data` over a precise range `Valid_Resource` with values in this enumeration as follows:

```
Capacity : constant := 10_000;
type Resource is new Integer range 0 .. Capacity;
subtype Valid_Resource is Resource range 1 .. Capacity;
No_Resource : constant Resource := 0;

type Status is (Available, Allocated);
type A is array (Valid_Resource) of Status;

Data : A := (others => Available);
```

Deallocating a resource consists in setting the corresponding status flag to `Available` when previously allocated:

```
procedure Free (Res : Resource) is
begin
   if Res /= No_Resource and then Data (Res) = Allocated then
      Data (Res) := Available;
   end if;
end Free;
```

Allocating a resource consists in searching for the first available resource if any, and then setting the corresponding status flag to `Allocated` before returning the resource position:

```
procedure Alloc (Res : out Resource) is
begin
   for R in Valid_Resource loop
      if Data (R) = Available then
         Data (R) := Allocated;
         Res := R;
         return;
      end if;
   end loop;
   Res := No_Resource;
end Alloc;
```

2.2 Model as a Ghost Function

In the simple memory allocator, we define a model of the allocator as a ghost function which will be used in the contracts of `Free` and `Alloc`. The *model* of the allocator data consists in two sets of resources: a set of resources available and a set of resources allocated.

```
package S is new Functional_Sets (Element_Type => Resource,
                                  No_Element   => No_Resource);
type T is record
   Available : S.Set;
   Allocated : S.Set;
end record;
```

Ghost function `Model` returns a value of this type, which additionally verifies additional properties relating the abstract model to the concrete data, expressed in function `Is_Valid`:

```
function Is_Valid (M : T) return Boolean;
function Model return T with Post => Is_Valid (Model'Result);
```

Ghost function `Is_Valid` expresses that sets `Available` and `Allocated` define a partition of the range of resources `Valid_Resource`:

```
function Is_Valid (M : T) return Boolean is
  ((for all E in M.Available => E in Valid_Resource)
      and then
   (for all E in M.Allocated => E in Valid_Resource)
      and then
   (for all R in Valid_Resource =>
      (case Data (R) is
         when Available => Mem (M.Available, R) and not Mem (M.Allocated, R),
         when Allocated => not Mem (M.Available, R) and Mem (M.Allocated, R))));
```

All the specification code presented so far in this section could be marked explicitly as ghost code. A better way of achieving the same result is to gather this code in a local package marked ghost as follows:

```
package M with Ghost is
   package S is ...
   type T is ...
   function Is_Valid ...
   function Model ...
end M;
```

With this model, is is straightforward to express the functional contract of `Alloc` and `Free` as contract cases, using the function `Is_Add` from the functional set library, which expresses that a `Result` set is the addition of an element to an input set. The same property could be expressed by using `Add` and equality on sets, but using `Is_Add` results in fewer quantifiers being used, which facilitates automatic verification. The notation `Result => Arg` uses the named parameter passing mechanism instead of the positional one to clarify which call argument corresponds to parameter `Result`.

```
procedure Alloc (Res : out Resource) with
  Contract_Cases =>

    --  When no resource is available, return the special value No_Resource
    --  with the allocator unmodified.

    (Is_Empty (Model.Available) =>
       Res = No_Resource
         and then
       Model = Model'Old,

     --  Otherwise, return an available resource which becomes allocated

     others =>
       Is_Add (Model.Available, Res, Result => Model.Available'Old)
         and then
       Is_Add (Model.Allocated'Old, Res, Result => Model.Allocated));

procedure Free (Res : Resource) with
  Contract_Cases =>

    --  When the resource is allocated, make it available

    (Mem (Model.Allocated, Res) =>
       Is_Add (Model.Available'Old, Res, Result => Model.Available)
         and then
       Is_Add (Model.Allocated, Res, Result => Model.Allocated'Old),

     --  Otherwise, do nothing

     others =>
       Model = Model'Old);
```

Function `Model` is implemented as a simple loop that creates the two sets `Available` and `Allocated` by iterating over the content of the array `Data`.

2.3 Automatic Proof of Refinement

GNATprove can be used to prove automatically that the code of the simple memory allocator presented in Section 2.1 is free of run-time errors and implements the specification presented in Section 2.2. The loop-free implementation of `Free` is proved easily with the default minimal proof settings (only one prover called with a timeout of one second per proof). Indeed, setting `Data(Res)` to `Available` directly maps at model level with removing `Res` from set `Model.Allocated` and adding it to set `Model.Available`. The implementation of `Alloc` contains a loop searching for the first resource available in `Data`, which requires the user to write a loop invariant summarizing the effect of the loop on variables modified in the loop (here `Data` is not modified while looping) and accumulating the information gathered across iterations on all variables (here that no available resource has been encountered yet):

```
pragma Loop_Invariant
  (Data = Data'Loop_Entry
   and then (for all RR in 1 .. R => Data (RR) = Allocated));
```

Once the first available resource `R` has been reached, setting `Data(R)` to `Allocated` directly maps at model level with removing `Res` from set `Model.Available` and adding it to set `Model.Allocated`. Then, the implementation of `Alloc` is proved easily at proof level 2 (all three provers called with a timeout of 10s per proof).

The proof of function `Model` also requires a simple loop invariant expressing that the property `Is_Valid` (from its postcondition) has been respected up to the value of resource for the current iteration of the loop. With this loop invariant, the implementation of `Model` is proved easily with the default minimal proof settings. Overall, the automatic proof of refinement of the simple memory allocator takes 12s on a laptop with 2.7 GHz Intel Core i7 and 16 GB RAM (using a single core).

3 Maintaining a Model Within the Implementation

As a more involved example, we consider a more realistic memory allocator based on a free list. As before, we present first the implementation and then the specification, to facilitate exposure and understanding, in reverse order compared to the actual software development.

3.1 A Free List Memory Allocator

Compared to the simple memory allocator presented in Section 2.1, the free list memory allocator uses an array `Data` of cells consisting of a status (available or allocated) and a pointers to the next resource in a linked list. A variable

`First_Available` points to the head of the linked list of available resources (a.k.a. the free list).

```
Capacity : constant := 10_000;
type Resource is new Integer range 0 .. Capacity;
subtype Valid_Resource is Resource range 1 .. Capacity;
No_Resource : constant Resource := 0;

type Status is (Available, Allocated);
type Cell is record
   Stat : Status;
   Next : Resource;
end record;
type A is array (Valid_Resource) of Cell;

Data : A := (others => Cell'(Stat => Available, Next => No_Resource));
First_Available : Resource := 1;
```

Allocating a resource consists in extracting and returning the free list head:

```
procedure Alloc (Res : out Resource) is
   Next_Avail : Resource;
begin
   if First_Available /= No_Resource then
      Res := First_Available;
      Next_Avail := Data (First_Available).Next;
      Data (Res) := Cell'(Stat => Allocated, Next => No_Resource);
      First_Available := Next_Avail;
   else
      Res := No_Resource;
   end if;
end Alloc;
```

Deallocation is done by adding the deallocated resource to the free list head.

3.2 Model as a Ghost Variable

In the free list memory allocator, unlike the simple memory allocator, not every configuration is a valid configuration of the software, thus we cannot represent the model as a function. For example, the initial value of `Data` as seen in Section 3.1 does not define a valid free list. What is needed is to add the following code to the startup code of the compilation unit (the *package elaboration code* in Ada parlance):

```
for R in Valid_Resource loop
   if R < Capacity then Data (R).Next := R + 1; end if;
end loop;
```

Thus, it is necessary to define what configurations are valid, and to prove both that the configuration is valid at startup and that operations `Alloc` and `Free` maintain the validity of the configuration. This is expressed with a boolean ghost function `Is_Valid`:

```
function Is_Valid return Boolean;
```

Although it would be possible to express the specification of the free list memory allocator based on a ghost function as seen in Section 2.2, this would make it very difficult to prove automatically the refinement property. Indeed, the relation between the abstract model and the concrete data would rely on the reachability of resources in a linked list, thus making it necessary to reason

by induction, something automatic provers are not good at. Instead, we define a model of the allocator as a ghost variable which will be used in the contracts of `Free` and `Alloc`. The *model* of the allocator data consists in a sequence of resources available and a set of resources allocated.

```
package S1 is new Functional_Sequences (Element_Type => Resource);
package S2 is new Functional_Sets (Element_Type => Resource,
                                   No_Element   => No_Resource);
type T is record
   Available : S1.Sequence;
   Allocated : S2.Set;
end record;
```

Ghost variable `Model` holds a value of this type:

```
Model : T;
```

The validity of the abstract model w.r.t. the concrete data at any given time is expressed by ghost function `Is_Valid`:

```
function Is_Valid return Boolean is
  ((if First_Available /= No_Resource then
      Length (Model.Available) > 0 and then
      Get (Model.Available, 1) = First_Available
    else
      Length (Model.Available) = 0)
      and then
   (for all J in 1 .. Length (Model.Available) =>
      Get (Model.Available, J) in Valid_Resource
        and then
      Data (Get (Model.Available, J)).Next =
        (if J < Length (Model.Available) then
           Get (Model.Available, J + 1) else No_Resource)
        and then
      (for all K in 1 .. J - 1 =>
         Get (Model.Available, J) /= Get (Model.Available, K)))
      and then
   (for all E in Model.Allocated => E in Valid_Resource)
      and then
   (for all R in Valid_Resource =>
      (case Data (R).Stat is
         when Available =>
           Mem (Model.Available, R) and not Mem (Model.Allocated, R),
         when Allocated =>
           not Mem (Model.Available, R) and Mem (Model.Allocated, R))));
```

This somewhat impressive (at least at first sight) function consists in a conjunction of four properties:

1. `First_Available` is the first available resource.
2. Sequence `Available` is an accurate image of the free list.
3. Set `Allocated` only contains valid resources.
4. Sequence `Available` and set `Allocated` define a partition of the range of resources `Valid_Resource`.

Like previously, all the specification code presented so far in this section is gathered in a local package marked ghost as follows:

```
package M with Ghost is
   package S is ...
   type T is ...
   Model : T;
   function Is_Valid ...
end M;
```

With this model, it is straightforward to express the functional contracts of `Alloc` and `Free` as contract cases, using the function `Is_Prepend` from the functional sequence library, which expresses that a `Result` sequence is obtained by prepending an element to an input sequence. The main difference with the contracts of the simple memory allocator is that property `Is_Valid` is required in precondition and in postcondition:

```
procedure Alloc (Res : out Resource) with
  Pre  => Is_Valid,
  Post => Is_Valid,
  Contract_Cases =>

    --  When no resource is available, return the special value No_Resource
    --  with the allocator unmodified.

    (Length (Model.Available) = 0 =>
       Res = No_Resource
         and then
       Model = Model'Old,

     --  Otherwise, return an available resource which becomes allocated

     others =>
       Is_Prepend (Model.Available, Res, Result => Model.Available'Old)
         and then
       Is_Add (Model.Allocated'Old, Res, Result => Model.Allocated));

procedure Free (Res : Resource) with
  Pre  => Is_Valid,
  Post => Is_Valid,
  Contract_Cases =>

    --  When the resource is allocated, make it available

    (Mem (Model.Allocated, Res) =>
       Is_Prepend (Model.Available'Old, Res, Result => Model.Available)
         and then
       Is_Add (Model.Allocated, Res, Result => Model.Allocated'Old),

     --  Otherwise, do nothing

     others =>
       Model = Model'Old);
```

Besides requesting that `Alloc` and `Free` maintain the validity of the configuration, we should also express that the configuration should be valid at startup with an initial condition on the package `List_Allocator` enclosing all the code of the free list memory allocator:

```
package List_Allocator with
  Initial_Condition => All_Available and Is_Valid
is
   ...
```

This initial condition expresses both that all resources should be available at startup and that the initial configuration should be valid.

3.3 Automatic Proof of Refinement

GNATprove can be used to prove automatically that the code of the free list memory allocator presented in Section 3.1 is free of run-time errors and implements the specification presented in Section 3.2. First, the implementation of

`Alloc` and `Free` must be augmented to express how the ghost variable `Model` is modified in relation to modifications on concrete data. This is a difference with the simple memory allocator where this was not needed, as `Model` in that case was a function. In procedure `Alloc`, this consists in adding two ghost assignments (in the case where allocation succeeds) to components of ghost variable `Model` expressing that the sequence of available resources is stripped from its first element, while the set of allocated resources is augmented with that same element:

```
Model.Available := Remove_At (Model.Available, 1);
Model.Allocated := Add (Model.Allocated, Res);
```

In procedure `Free`, this consists in adding two ghost assignments (in the case where deallocation succeeds) to components of ghost variable `Model` expressing that the set of allocated resources is stripped from the element passed in argument to `Free`, while the sequence of available resources is prepended with that same element:

```
Model.Allocated := Remove (Model.Allocated, Res);
Model.Available := Prepend (Model.Available, Res);
```

Package `List_Allocator` contains elaboration code to set the initial value of array `Data`. Similarly, local ghost package `M` needs to set the initial value of the ghost variable `Model` in its elaboration code. This initial value needs to be expressed in `M`'s initial condition so that it can be used to prove `List_Allocator`'s initial condition presented in Section 3.2. The code and contracts are not shown here for lack of space but can be found in a public repository (see reference in conclusion).

Despite the complexity of the `Is_Valid` function relating the abstract model to the concrete data, automatic proof is achieved as easily as for the simple memory allocator at proof level 2 (with an additional switch to prevent use of prover steps limit). Overall, the automatic proof of refinement of the free list memory allocator takes 18s on a laptop with 2.7 GHz Intel Core i7 and 16 GB RAM (using a single core).

4 Related Work

B Method [2] has been used extensively in the railway industry over the past 20 years to prove that an implementation is a correct refinement of a specification [6]. While interactive proof was originally the main means to achieve proof, automation of proofs has steadily increased until now [3,11], as well as automatic refinement of abstract specifications [7]. The Isabelle Refinement Framework pursues a similar goal of facilitating the proof of a stepwise refinement in Isabelle/HOL from an abstract functional specification to an imperative implementation [16]. Our work aims at the same goal in the context of a programming language, with all the associated benefits in terms of strong typing, expressivity and tool support. Prior experiments in that directions have been performed in the context of the Eiffel programming language [22]. Our work achieves this

goal in the context of a mature and industrially supported formal verification environment.

Automatic proof that code implements a specification expressed as a contract (precondition and postcondition) is the subject of active research, based on advances in the underlying proof technology and the intermediate verification languages, as visible from the activity in relevant workshops (in particular the SMT workshop and Boogie workshop) and tool competitions (in particular VerifyThis [14]).

Recent works [13,18] show how an abstract specification about mathematical quantities (real or integers) can be implemented efficiently in code (with floating-point numbers or bitvectors), and the refinement relation be proved automatically in Why3 or SPARK.

Automatic proof of refinement with more complex data has lead to the introduction of many concepts, some of which are used in this paper: ghost code, model code, alias management policies (such as ownership, permissions, separation logic) [9,23]. The difference in our approach is that the user can write contracts and intermediate assertions (like loop invariants which are needed in all these techniques) in the same programming language as the implementation. In particular, all contracts and assertions in SPARK can be executed and debugged, which greatly facilitates formal development. This was very useful during the development of the memory allocator examples presented in this paper, to catch bugs early on, before attempting automatic proof.

A recent work [15] examines which programming language features are useful in proofs of refinement, some of which could be included in future versions of the SPARK programming language.

5 Conclusion

This article presents two techniques to specify abstractly the behavior of a software component in terms of mathematical structures (sequences, sets and maps) and a methodology based on the SPARK tools to prove automatically that an efficient imperative implementation is a correct refinement of the abstract specification. The proposed methodology is illustrated with challenging concrete examples of memory allocators.[6] To the best of our knowledge, this is the first time such refinement proof is done automatically with both the specification and the implementation expressed in the same (executable) programming language.

In this article, we define two different abstract specifications for respectively the simple memory allocator and the free list memory allocator: the simpler specification based on sets is implemented by the simple memory allocator while the more involved specification based on sets and sequences is implemented by

[6] The results presented in this article can be reproduced with SPARK GPL 2016, which will be available in June 2016 at `http://libre.adacore.com`. The source code of the examples is available in the SPARK public repository at `https://forge.open-do.org/anonscm/git/spark2014/spark2014.git`, under `testsuite/gnatprove/tests/allocators`.

the free list memory allocator. One could object that, as both allocators deliver the same overall service (ignoring efficiency here), they could be refinements of the same specification. Indeed, it would be interesting to prove that the simple memory allocator is a refinement of the specification given in Section 3.2 and to prove that the free list memory allocator is a refinement of the specification given in Section 2.2. The latter would not be feasible in SPARK as it would require a notion of package invariant to hide property `Is_Valid` (although a sibling notion of type invariant will be supported in future versions of SPARK). The former should be already possible in SPARK.

Although we do not present it in this article, this abstraction also allows proving the correct use of the two allocators in client code, which would otherwise require to expose implementation details to the client. Effects of calling (de)allocation procedures on the concrete data and ghost model are visible from client code, and can be either left implicit (for the tool to generate) or explicitly stated. Note that if multiple allocators are needed in a project, the specification and code presented can be shared by making the package *generic*, in which case the automatic proof will be repeated for each instantiation of the generic.

Acknowledgements We would like to thank Claude Marché, David Mentré, Piotr Trojanek as well as the anonymous reviewers for their useful comments.

References

1. EN 50128:2011 railway applications - communication, signalling and processing systems - software for railway control and protection systems, 2011.
2. J.-R. Abrial. *The B-Book: Assigning programs to meanings*. Cambridge University Press, 1996.
3. Jean-Raymond Abrial. Formal methods in industry: Achievements, problems, future. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 761–768, New York, NY, USA, 2006. ACM.
4. J. Barnes. Ada 2012 Rationale. 2012.
5. J. Barnes. *SPARK: The Proven Approach to High Integrity Software*. Altran Praxis, 2012.
6. Jean-Louis Boulanger, editor. *Formal Methods Applied to Industrial Complex Systems: Implementation of the B Method*. Wiley, 2014.
7. L. Burdy and J.-M. Meynadier. Automatic refinement. In *FM'99 workshop – Applying B in an industrial context : Tools, Lessons and Techniques*, 1999.
8. P. Chalin. Engineering a Sound Assertion Semantics for the Verifying Compiler. *IEEE Trans. Software Eng.*, 36(2):275–287, 2010.
9. Dave Clarke, James Noble, and Tobias Wrigstad, editors. *Aliasing in Object-Oriented Programming: Types, Analysis, and Verification*. Springer-Verlag, Berlin, Heidelberg, 2013.
10. C. Comar, J. Kanig, and Y. Moy. Integrating Formal Program Verification with Testing. In *Proc. ERTS*, 2012.
11. David Delahaye, Catherine Dubois, Claude Marché, and David Mentré. The BWare project: Building a proof platform for the automated verification of B proof obligations. In Yamine Ait Ameur and Klaus-Dieter Schewe, editors, *Abstract State*

Machines, Alloy, B, TLA, VDM, and Z, volume 8477 of *Lecture Notes in Computer Science*, pages 290–293. Springer Berlin Heidelberg, 2014.
12. Claire Dross, Pavlos Efstathopoulos, David Lesens, David Mentré, and Yannick Moy. Rail, space, security: Three case studies for SPARK 2014. In *Proc. ERTS*, 2014.
13. Claire Dross, Clément Fumex, Jens Gerlach, and Claude Marché. High-level functional properties of bit-level programs: Formal specifications and automated proofs. Research Report 8821, Inria, December 2015.
14. Marieke Huisman, Vladimir Klebanov, and Rosemary Monahan, editors. *International Journal on Software Tools for Technology Transfer, special issue, VerifyThis 2012*, volume 17. 2015.
15. Jason Koenig and K. Rustan M. Leino. Programming Language Features for Refinement. *Submitted to EPTCS*, 2015.
16. Peter Lammich. Refinement based verification of imperative data structures. In *Proceedings of the Conference on Certified Programs and Proofs*, 2016.
17. Emmanuel Ledinot, Jean-Paul Blanquart, Jean-Marc Astruc, Philippe Baufreton, Jean-Louis Boulanger, Cyrille Comar, Hervé Delseny, Jean Gassino, Michel Leeman, Philippe Quéré, and Bertrand Ricque. Joint use of static and dynamic software verification techniques: a cross-domain view in safety critical system industries. In *Proceedings of the 7th European Congress on Embedded Real Time Software and Systems (ERTS^2 2014), Toulouse, France, February 5-7, 2014*, 2014.
18. Claude Marché. Verification of the functional behavior of a floating-point program: an industrial case study. *Science of Computer Programming*, 96(3):279–296, March 2014.
19. John W. McCormick and Peter C. Chapin. *Building High Integrity Applications with SPARK*. Cambridge University Press, 2015.
20. B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., 1st edition, 1988.
21. I. O'Neill. SPARK – a language and tool-set for high-integrity software development. In *Industrial Use of Formal Methods: Formal Verification*. Wiley, 2012.
22. Jonathan Ostroff, Chen wei Wang, Eric Kerfoot, and Faraz A. Torshizi. ES-Verify: A tool for automated model-based verification of object-oriented code. In *Formal Methods 2006*. Poster.
23. Asma Tafat, Sylvain Boulmé, and Claude Marché. A refinement methodology for object-oriented programs. In Bernhard Beckert and Claude Marché, editors, *Formal Verification of Object-Oriented Software*, volume 6528 of *Lecture Notes in Computer Science*, pages 153–167. Springer Berlin Heidelberg, 2011.