AdaCore | Build Software that Matters

Tech Paper

Containers for specification in SPARK

Paper is by the International Journal on Software Tools for Technology Transfer

Author: Claire Dross

Containers for specification in SPARK

Claire Dross[1]

Accepted: 4 April 2025

Abstract
The SPARK tool analyzes Ada programs statically. It can be used to verify both that a program is free from runtime exceptions and that it conforms to a specification expressed through contracts. To facilitate dynamic analysis, Ada contracts are regular Ada expressions which can be evaluated at execution. As a result, the annotation language of the SPARK tool is restricted to executable constructs. In this context, high-level concepts necessary for specification by contracts need to be supplied as libraries. For example, the latest version of the Ada language introduces unbounded integers and rational numbers to the standard library. In this article, we present the functional containers library, which provides collections suitable for use in specification. We then explain how they can be used to specify and verify complex programs through concrete examples that have been developed over many years. Finally, we describe how these libraries are supported in the SPARK tool using reusable specification features instead of built-in support, i.e., a hard-coded mapping of library functionalities to axiomatized theories for the underlying provers.

Keywords Containers · Specification · Static verification · Program proof

1 Introduction

With software taking on increasingly large roles in critical embedded systems, it has become critical to verify software in an efficient way. This leads more and more industrial software companies to deploy formal verification techniques [2, 15]. The SPARK tool [4] performs static analysis of Ada programs. It can be used to verify that a program is free from runtime exceptions, including but not limited to division by zero, buffer overflows, null pointer dereferences, etc. High-level functional properties can also be verified by the tool. These properties need to be expressed as contracts — pre- and post-conditions, type invariants, etc.

The SPARK tool performs *deductive analysis*: It takes as its input an Ada program, annotated with contracts, and generates from it logical formulas, called *verification conditions*. These verification conditions are then given to automated solvers. If all the conditions are verified, then the Ada program conforms to its specification. Deductive analysis works modularly on a per-subprogram basis,[1] using the subprogram's contract to summarize its behavior while analyzing callers. As a result, it is necessary for the user to manually annotate her subprograms with contracts for the tool to work. For both the analysis and the annotation process to remain tractable, some features of Ada have to be restricted; the SPARK toolset rejects Ada programs including these features as being non-conformant. In particular, SPARK does not support side-effects in expressions (but they can occur in statements) nor aliasing (when modifying one object can change the value of another object).

Since 2012, contracts are part of the Ada language. They are mostly used for dynamic analysis and can be verified at runtime. Therefore, they have the same semantics as regular Ada expressions. In the SPARK tool, we keep the executable semantics of contracts. It makes it easier for developers to write the contracts, both because they do not have to learn a new language, and because the contracts can be tested and debugged like normal code. However, it has the side-effect of restricting the annotation language to executable constructs. To alleviate this limitation, high-level concepts necessary to write certain specifications can be added as libraries. Unbounded integers and rational numbers have been introduced

✉ C. Dross
dross@adacore.com

[1] AdaCore, 46 rue d'Amsterdam, 75009 Paris, France

[1] In Ada, *subprogram* is a generic term meaning a function or a procedure, that is, a function which does not return a result but works by side-effects.

Published online: 27 May 2025

recently into the Ada runtime. They can be used to avoid overflows in contracts, or to reason about the rounding error in floating point computations [8].

Another concept which is commonly used in specification is a collection: set, sequence, map etc. Collections used for specification are different from their counterparts used during development. They are more of a mathematical concept, and less concerned about efficiency. In Sect. 2, we present the *functional containers library,* which was introduced for this purpose in 2016. In Sect. 3, we explain how it can be used to enhance the specification and verification of complex programs through concrete examples that have been developed through the years. Finally, we describe how this library is supported in the SPARK verification tool in Sect. 4.

2 The formal and functional containers

The standard library of Ada provides implementations of commonly-used standard containers: vectors, doubly-linked lists, as well as sets and maps, both ordered and hashed. These containers come in various flavors: bounded to avoid dynamic allocations on embedded systems, indefinite to hold elements of variable sizes, etc. To allow for efficient access, these containers implement a notion of iterators, named *cursors*. Cursors are basically pointers giving direct access to an element in the container. They provide an easy way to iterate over all the elements of a container. While cursors are desirable in terms of usability, they are unfortunately not compatible with the restrictions imposed on input by the SPARK tool. Indeed, each cursor involves an alias of the container it belongs to, as modifying the container might cause the cursor to become invalid or designate a different element, and SPARK does not support aliases.

To alleviate this issue, SPARK-compatible versions of the standard containers [11] have been implemented. They are called *formal containers* and are designed to be as close to the standard containers as possible. They provide cursors like the Ada containers, but these cursors are nothing more than indices in an array constituting the underlying memory of the container. As a result, the formal container API is slightly different from the standard one, as the container needs to be passed along with the cursor to determine its validity or access the corresponding element, as can be seen in Fig. 1.

Another important distinction between the standard Ada containers and the formal ones is the use of contracts, as introduced by the `Pre` and `Post` aspects. In Ada, they are primarily meant for dynamic verification: they introduce boolean expressions that should evaluate to true either before (for preconditions) or after (for postconditions) a subprogram call. However, the role of contracts in deductive verification is key, as subprograms are analyzed modularly. The precondition is then used to caracterize all possible calling contexts of the subprogram, while the postcondition summarizes the effect of subprogram calls when analyzing their caller. As a consequence, supplying contracts on libraries is necessary to be able to verify code using them. Note that the formal containers are not themselves verified using SPARK, but they are compatible with its restrictions and their primitives have been annotated with contracts allowing user code that leverages them to be analyzed. Though they have not been formally verified, they have been tested like every other runtime unit of Ada.

```
function Element
  (Position : Cursor) return Elem_T;
-- Access an element in a standard map
function Element
  (Container : Map;
   Position  : Cursor) return Elem_T
with Pre  => Has_Element (Container, Position),
     Post => ...;
-- Access an element in a formal map
```

Fig. 1 The `Element` function is used to access an element in a standard or a formal map. As the cursors no longer hold a reference to a container in the formal container library, the `Element` function takes the container as an additional parameter. It is annotated with a pre and a postcondition that can be used to verify user code.

At the beginning, we attempted to use the formal containers in high-level specifications, but we quickly found out that it was not tractable. Indeed, these containers are onerous for verification, as they pull with them numerous secondary considerations, like the order of iteration, or the validity and position of cursors. A new library of containers, named *functional containers*, was introduced to alleviate this issue. They are designed to be light-weight in terms of specifications. They only offer a small number of functional operations, with as few constraints as possible. They are unbounded, might contain any kind of elements (even with variable sizes) and can be used easily (no need to provide a hash or compare function for sets and maps in particular).

The functional containers library provides maps, sequences, and sets. Their API consists of functions for creating new containers, as opposed to procedures for modifying existing ones. As an example, Fig. 2 shows a part of the API of functional maps. The function `Add` can be used to create a new map from all the mappings in an existing map and an additional one. Functional maps are defined in terms of three basic properties, a function `Has_Key` to check whether a key has an association in the map, a function `Get` to retrieve this association, and a function `Length` returning the number of keys with an association in the map. Other primitives are specified in terms of these properties, like the function `Add`.[2]

[2] The operators `and then` and `or else` in Ada are alternative versions of the standard Boolean operators `and` and `or` using short-circuit evaluation.

```
function Length (M : Map) return Big_Natural;
function Has_Key
  (M : Map; K : Key_T) return Boolean;
function Get
  (M : Map; K : Key_T) return Elem_T
with Pre ⇒ Has_Key (M, K);
function Add
  (M : Map; K : Key_T; E : Elem_T) return Map
with
  Pre  ⇒ not Has_Key (M, K),
  Post ⇒ Length (Add'Result) = Length (M) + 1
    and then Has_Key (Add'Result, K)
    and then Get (Add'Result, K) = E
    and then ...;
```

Fig. 2 Part of the API of functional maps. The functions `Length`, `Has_Key`, and `Get` are the only basic properties of a functional map. All other primitives, like `Add`, are specified in terms of these properties.

```
(for some P in N + 1 .. 2 * N ⇒ Is_Prime (P))
--  Chebyshev's theorem

(for all E of A ⇒ E mod 2 = 0)
--  The array A only contains even numbers
```

Fig. 3 Quantified expressions in Ada

Even if they are mostly used for specifications, the functional containers are executable. To remain reasonably efficient, their implementation involves several levels of sharing. The memory is managed through reference counting. As functional containers are unbounded, the function `Length`, which computes their cardinality, returns a mathematical integer using the big integers library of Ada. To make it easier for a user to instantiate functional sets and maps, neither a hash function nor a comparison operator is required. This basically forces the implementation to use a sequence instead of a more appropriate structure, so searching for an element in these containers might be inefficient.

When writing specifications, being able to quantify over the elements of a container is key. Fortunately, this requirement was foreseen when Ada contracts were designed, so *quantified expressions* are supported natively in the language. However, because of the primarily executable semantics of contracts, this quantification is restricted. It is possible to quantify over the values of a range of integers, or over the elements of an array, but not over all values of an arbitrary type. Figure 3 demonstrates quantified expressions in Ada. The `for all .. ` and `for some .. ` syntaxes provide respectively universal and existential quantification. The keyword `in` is used to refer to values in a range, while the keyword `of` is for elements of an array.

It is possible for users to provide their own primitives that can be used to iterate and quantify over data structures. This capability is used in particular in the standard container library of Ada to iterate or quantify either over all cursors valid in a container using the keyword `in` or over its elements using the keyword `of`. The iteration primitives provided by the user are then used by the compiler to expand the quantified expression into a loop. As the Ada feature for user-defined iterators relies on aliases, the SPARK tool defines a variant called the `Iterable` annotation, which we present in more details in Sect. 4. It makes it possible to quantify over the elements of a functional set or the keys of a functional map, as can be seen on Fig. 4.

```
function "≤"
  (Left : Set; Right : Set) return Boolean
is
  (for all E of Left ⇒ Contains (Right, E));
```

Fig. 4 The operator ≤ implements set inclusion.

```
function Count
  (S    : Set;
   Test : not null access
     function (E : Elem_T) return Boolean)
  return Big_Natural
with Post ⇒ Count'Result ≤ Length (S);
```

Fig. 5 Function counting the number of elements in a set on which the input function `Test` returns True

The Ada standard containers allow for providing a user-defined equivalence relation that is used to check inclusion into sets and maps. The functional containers offer the same facility. The `Contains` function on functional sets and the `Has_Key` function on functional maps both work modulo equivalence. As the functional containers primitives are specified in terms of these basic operations only, it is not possible to track the exact value of elements inserted inside functional sets, or of keys inserted inside functional map if the supplied equivalence relation is not the equality.

Finally, the functional container library provides a set of higher-order functionalities. As Ada is not a functional language, they take function pointers as parameters. They provide container comprehension (construction, filtering, transformation...) as well as common operations like summation or counting. The signature of the `Count` function is given in Fig. 5. It counts the number of elements in the set with a given property. This function is only well-defined if function `Test` given as parameter to the call returns the same value on all elements of an equivalence class.

Both the formal and the functional containers libraries are available online.[3]

3 Functional containers as models

Mathematical-like collections are often used in specifications. They appear in particular when describing data-structures, as the common understanding of how they should

[3] https://github.com/AdaCore/SPARKlib/tree/master/src.

behave: a pointer-based singly-linked data-structure might represent a sequence, a red-black-tree implement a set, etc. They can also occur during the verification process to express why an algorithm is correct: we will speak about sets of reachable elements or multisets, representing permutations of an array in a sort. These collections are really conceptual objects, which do not actually exist in the program. They are generally called *models*. As they are lightweight and close to mathematical collections, functional containers are well suited to this usage. In this section, we present common uses of collections as models, and how they can be applied using functional containers.

3.1 Specifying data-structures

When the functional containers were designed, the first objective was to use functional containers to specify the formal containers library. Indeed historically, the formal containers were axiomatized in WhyML, the input language of the Why3 tool used as part of the SPARK backend [12]. This required special handling so the formal containers were recognized specifically and linked to the correct WhyML module. This mechanism had the advantage of making it possible to use the rich specification features offered by WhyML (abstract logic functions, unrestricted quantifiers, axioms, etc.) most of which cannot be mirrored in SPARK as they are not executable. However, the maintenance cost was prohibitive, as the mechanism had to be kept up-to-date through successive updates of both SPARK and WhyML.

Replacing this special handling by regular SPARK contracts without degrading the provability was a challenge. We decided to go for *model functions*, returning functional containers. A model function is a *ghost* function, meaning it can only be used in specifications. It takes as a parameter a concrete object and returns its model: another object, generally simpler to reason with. The operations on the concrete object are then described in terms of their effects on the abstract model. In Fig. 6, the model of a FIFO queue implemented as a ring buffer is a sequence giving the elements of the buffer in the order in which they will be retrieved. Using this model, its primitive operations can be specified in a straightforward way.

As the formal containers are relatively complex, we decided to use several model functions for their specification. Each formal container provides a main model function called `Model`, which returns a functional container giving a high-level view of the data-structure. We use sequences for vectors and doubly-linked lists, and functional sets and maps for ordered and hashed sets and maps respectively. Unfortunately, this high-level model is not enough to verify subprograms using cursors to iterate over a formal container. Indeed, it does not represent the cursors, nor the order in which the elements occur during an iteration over a set or a map. To

```
function Model
  (R : Ring_Buffer) return Sequence
with Ghost;

procedure Enqueue
  (R : in out Ring_Buffer;
   E : Integer)
with
  Pre  => not Is_Full (R),
  Post => Model (R) = Add (Model (R)'Old, E);
  --  The new model of R is its old model with
  --  E added at the end.

procedure Dequeue
  (R : in out Ring_Buffer;
   E : out Integer)
with
  Pre  => not Is_Empty (R),
  Post => Model (R) = Remove (Model (R)'Old, 0)
    and then E = Get (Model (R)'Old, 0);
  --  The new model of R is its old model
  --  without the first element. E is set to
  --  the first element of the old model of R.
```

Fig. 6 The model of a ring buffer is a functional sequence of elements in the order in which they were added to the buffer. The `Dequeue` and `Enqueue` functions are defined in terms of their effect on the model of their parameter.

alleviate this issue, one or two additional model functions are defined for each container. The `Positions` function returns a functional map which associates the cursors that are valid in a container to an integer standing for their position in the container. For sets and maps, the `Elements` or `Keys` function returns a sequence of elements or keys to model their order in the container. Note that this order is defined both for the hashed and ordered containers, as both define an order of iteration on their elements.

This layered approach allows users of the formal containers library to choose the level of granularity they need. As an example, the procedure `Set_All_To_Zero` sets the elements associated with each key in a map to 0. Its postcondition is given in Fig. 7. It only uses the high-level model of the map, as it does not care about the cursors or order of iteration in the container. However, these considerations are necessary when verifying its implementation. In particular, many algorithms involving containers use loops to iterate over them. Verifying a loop in deductive verification requires the use of *loop invariants*. They are propositions which should summarize precisely the effect of the loop up to the current iteration. Verification tools use them as cut points to flatten the control flow graph when generating verification conditions. The loop invariant in Fig. 8 is one of the annotations used to verify the loop setting each element to 0 in its body. It uses the `Positions` function to get the position of the current cursor, and then the `Keys` function to state that the elements associated with all the keys occurring before this position have already been replaced.

```
procedure Set_All_To_Zero (M : in out Map)
with
  Post =>
    (for all K of Model (M)'Old =>
        Has_Key (Model (M), K))
    and then
      (for all K of M =>
          Has_Key (Model (M)'Old, K)
          and then Get (Model (M), K) = 0);
```

Fig. 7 The procedure `Set_All_To_Zero` has a postcondition which states that the keys of `M` are preserved by the call and that every key in the map is associated to 0 after the call. This can be expressed using only the `Model` function. The order of iteration and the validity of cursors is not relevant here.

```
pragma Loop_Invariant
  (for all P in
      1 .. Get (Positions (M), Cu) - 1
   => Get (Model (M), Get (Keys (M), P)) = 0);
```

Fig. 8 A loop invariant used to verify the procedure `Set_All_To_Zero`. It uses the `Positions` map to query the position of the current cursor `Cu` and the `Keys` sequence to retrieve the keys situated before this position in the map.

3.2 Verifying data-structures

Even though it was not done for the formal containers library, it is possible to use SPARK to verify these annotations. For the verification to be possible, it is necessary to describe precisely the link between the underlying data-structure and its model in the postcondition of the model function. In Fig. 9, the function `Valid_Model` links the value of a ring buffer implemented as an array with a first index and length field to the value of the sequence that models it. It can be used in the postcondition of `Model`, so the contracts in Fig. 6 can be verified.

This method was used successfully in case-studies of various sizes through the years. The first use of the functional containers to verify a SPARK program was developed to showcase the capability in 2016 [9]. It features a simple allocator inside a memory array modeled using a set of allocated cells and a sequence for the free list. A substantially more complex example is the proof of the insertion inside a red-black tree encoded inside a memory array [10]. The complexity of the specification is handled by building the concrete structure incrementally, starting from binary trees, to search trees, to finally implement and verify the insertion in a red-black tree. More recent examples use functional containers to model pointer-based data-structures, which have been supported by the SPARK tool only for the last couple of years [7]. To support this new use-case, the functional containers library had to be updated. Indeed, sequences are bounded by the machine integer type used to index them, and functional sets and maps used to have a theoretical bound on their cardinality due to the machine integer type used for their

```
type Ring_Buffer is record
   Content : Content_Array;
   First   : Positive range 0 .. Max - 1 := 0;
   Length  : Natural  range 0 .. Max := 0;
end record;

function Valid_Model
  (R : Ring_Buffer;
   M : Sequence) return Boolean
is
  (Length (M) = R.Length
   and then
     (for all I in 0 .. R.Length - 1 =>
        R.Content ((R.First + I) mod Max) =
        Get (M, I)))
with Ghost;

function Model
  (R : Ring_Buffer) return Sequence
with
  Ghost,
  Post => Valid_Model (R, Model'Result);
```

Fig. 9 The function `Valid_Model` links the element of the ring buffer to their corresponding value in the model sequence. It is used as a postcondition of the `Model` function. It makes it possible to verify the implementation of `Enqueue` and `Dequeue` procedures presented in Fig. 6.

`Length` function. The restriction on sets and maps has been lifted by replacing the return type of their `Length` functions by unbounded integers, and a new type of sequence indexed by unbounded integers was introduced in the library.

On complex data-structures, it is possible to use several levels of models to perform a proof by refinement. Basically, a lower-level model, close to the concrete data-structure, is used to annotate and verify the basic operations. Then, one or several higher-level models might be introduced to further abstract away the operations. As an example, we are currently working on using SPARK to verify the implementation of the bounded formal hashed sets. As schematized in Fig. 10, these sets are implemented in an array. A hash function is used to choose a bucket for each element of the set. Each bucket is the head of a list implemented through a `Next` field in the memory array. For the verification to remain tractable, we have introduced two levels of models. The lower-level model keeps a memory array, but only to store the values, as represented in Fig. 11. The buckets contain functional sequences that store the corresponding allocated indices. At this level, an invariant enforces that the list structure encoded in the memory array is well-formed (there are no cycles), that the reachable indices in each bucket and the free list are disjoint, and that they cover the whole memory (there are no leaks). The notion of buckets disappears completely in the higher level model, see Fig. 12. It simply represents the set as the memory for values and a big sequence, containing the allocated indices in the order in which they will be traversed when iterating over the set. Here, we verify that the sequence

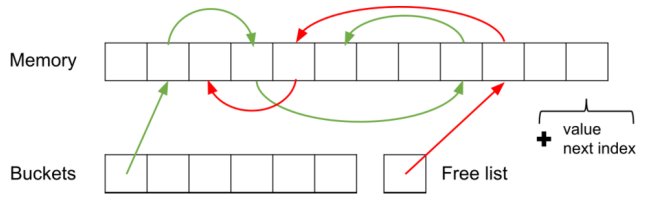

Fig. 10 Concrete implementation of a formal hashed set. The content of the set is stored in a bounded memory array. Each cell of this array contains both a value and a `Next` field, used to represent linked lists in the memory. The array of buckets holds the heads of the lists associated to each hashed value. The cells which are not allocated yet are linked together in the same way, and their head is stored separately.

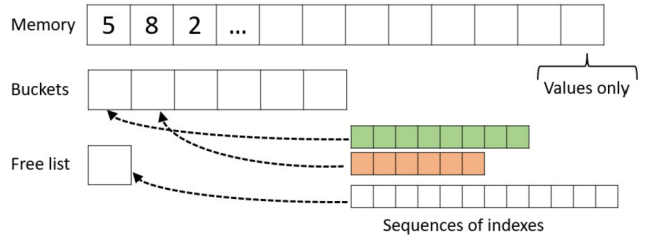

Fig. 11 Low-level model of a formal hashed set. The values contained in the set are still stored in a bounded array. The linked structure however has been removed from the memory. Instead, each bucket now uses a functional sequence to store the indices of the corresponding values in the order of iteration. The free list is also represented as a sequence.

models the correct order of iteration over the set, and that elements stored in the set are unique up to an equivalence relation. The final objective is to be able to verify the specification written in terms of the three model functions `Model`, `Elements`, and `Positions` presented in Sect. 2.

3.3 Going further

Modeling the content of other data-structures is by far the most common use-case for functional containers in our experience. However, more exotic use-cases also exist. In particular, the container might model a state which is not actually present in the program, but represents a concept used in the specification only. It makes it possible to address properties which are not generally in the domain of SPARK. In this section, we focus on two such use cases.

The SPARK tool enforces an ownership policy to ensure non-aliasing when dealing with pointers. As a result, the built-in support for pointers does not allow verifying programs that rely on aliasing. In particular, data-structures involving sharing or cycles—doubly-linked lists, directed acyclic graphs, etc.—cannot be handled by the SPARK tool. To work around this restriction, it is possible to hide the pointers and model them as indices in a memory map. The implementation still uses pointers though, so the map does not represent any actual structure in the code. It makes it possible to reason about pointers with aliasing by annotating explicitly which pointers can be aliases of each other. Using a single memory object standing for all allocated data makes both annotation and verification more difficult however, so the built-in support of pointers stays more efficient when it applies.

Figure 13 shows the contracts provided for the procedure `Allocate`. It allocates a memory region, initialized with the provided value, and returns a pointer to this memory region. Its contract expresses that it modifies a global ghost object called `Memory`. This object is the functional map standing for the model of the actual program heap. In the postcondition of `Allocate`, it is necessary to describe its effect on the whole abstract memory map. We use two universally quantified formulas to state that `P` is the only newly allocated cell, and that the values designated by other pointers are preserved. For comparison, the procedure `Allocate` in Fig. 14 uses the built-in support for pointers in SPARK. It is not considered to read or modify any global state as allocated cells are treated as parts of the pointer that owns them, so its contract is far simpler.

As another example, functional sequences can be used to model a restricted form of temporal logic. Here, a history is represented as a ghost sequence of events, where events would be, for example, a call to a particular subprogram or the reception of a message. Each time an event occurs, it is added at the end of the sequence. Using the ghost sequence, it is then possible to express properties over the order in which the events occurred. The SPARK tool can be used to verify that these properties are maintained through the program. The snippet in Fig. 15 is extracted from the code of OpenUxAS, a framework developed by Air Force Research Labs for mission-level autonomy for teams of cooperating unmanned vehicles [1]. This framework is implemented as several services communicating through message passing. In this example, the history records the emission and reception of messages. The function `No_Route_Request_Lost` uses the history to express that all received messages of kind `Route_Request` have been handled: they are either in the set of pending requests or a response has been sent. As can be seen in the contract of `Handle_Route_Request`, this property is stated both in preconditions and in postconditions of subprograms handling messages in the service. It allows the SPARK tool to verify that it is an invariant maintained by the service.

4 Tool support — a hybrid solution

Libraries that introduce high-level concepts for use in specifications usually define one or more private types for this concept along with functions providing the necessary functionalities. Their actual implementation is in general hidden from the proof tool, both to keep the concepts as simple as

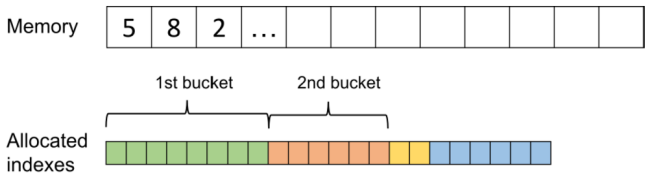

Fig. 12 High-level model of a formal hashed set. The memory array of values remains the same as in the low-level model. The buckets and the free list are not represented anymore. Instead, we use a single functional sequence containing all the allocated indices in the order of iteration.

```
type Pointer is private;
procedure Allocate
   (O : Object; P : out Pointer)
with
   Global ⇒ (In_Out ⇒ Memory),
   --  P is a valid pointer in Memory
   -- designating  the value O.
   Post ⇒ Has_Key (Memory, Address (P))
   and then Get (Memory, Address (P)) = O
   --  Every pointer previously valid in Memory
   --  remains valid and keeps designating the
   --  same value.
   and then (for all K of Memory'Old ⇒
               Has_Key (Memory, K)
               and then Get (Memory, K) =
                        Get (Memory'Old, K))
   --  P is the only address allocated by the
   --  call.
   and then (for all K of Memory ⇒
               Has_Key (Memory'Old, K)
               or else K = Address (P));
```

Fig. 13 The function `Allocate` allocates a new memory region for its input object `O`. After the call, its parameter `P` is a pointer to this newly allocated region. The fact that `P` is a pointer is hidden from the SPARK tool using privacy. The effect on the program heap is modeled through a ghost `Memory` map.

```
type Builtin_Pointer is access Object;
procedure Allocate
   (O : Object;
    P : out Builtin_Pointer)
with
   -- P is not null and designates the value O
   Post ⇒ P ≠ null and then P.all = O;
```

Fig. 14 The function `Allocate` allocates a new memory region for its input object `O`. After the call, its parameter `P` is a pointer to this newly allocated region. As built-in pointers are handled through ownership by the SPARK tool, the part of the program heap designated by `P` is treated as a part of `P` for the verification. Therefore, there is no need to model the memory in the contracts of `Allocate`.

possible and because they use features that are out of the scope of SPARK (sharing, finalization...). Some of these libraries, like the big numbers library of Ada, are specifically recognized by the tool. It allows mapping them to built-in concepts in the underlying logic. As an example, the big integers library is recognized and mapped to mathematical integers and arithmetic operations over them. Libraries

```
function Route_Response_Sent
   (Id : Int64) return Boolean
is
   (for some E of History ⇒
      E.Kind = Send_Route_Response
      and then E.Id = Id);

function No_Route_Request_Lost
   (Pending_Requests : Set) return Boolean
is
   (for all E of History ⇒
       (if E.Kind = Receive_Route_Request then
             Contains (Pending_Requests, E.Id)
          or else Route_Response_Sent (E.Id)));

procedure Handle_Route_Request
   (Data    : Configuration_Data;
    Mailbox : in out Mailbox_Type;
    State   : in out State_Type;
    Request : Route_Request)
with
    Pre ⇒ ...
      --  History invariants
      and then No_Route_Request_Lost
        (State.Pending_Routes)
      and then ...,
    Post ⇒ ...
      --  The request has been added to the
      --  history.
      and then History'Old < History
      and then
        Get (History, Last (History)).Kind
        = Receive_Route_Request
      and then
        Get (History, Last (History)).Id =
        Request.Request_ID
      --  History invariants
      and then No_Route_Request_Lost
        (State.Pending_Routes)
      and then ...;
```

Fig. 15 Extract of the OpenUxAS code base. The function `No_Route_Request_Lost` uses the `History` sequence to express a safety invariant of the service: Every `Route_Request` received by the service is either pending or a response has been sent. The procedure `Handle_Route_Request` performs the treatment when a `Route_Request` is received. It stores the event in the history. Its contract states that it maintains the `No_Route_Request_Lost` invariant.

which are not recognized specifically by the tool are handled like user code: private types are black boxes and their functions are uninterpreted. The only information available to

the analysis tool comes from the Ada contracts of the library. Hardcoding a library in the tool results in better provability when there is a simple mapping between its functionalities and a built-in concept in the underlying logic, so SPARK users can benefit from it. However, it requires more development work, as well as maintenance, when there is a change either in the library or in the theory it is mapped to in the solvers. Choosing to stick to the default handling based on Ada contracts for SPARK proof for non-hardcoded libraries also has advantages. The libraries are easier to extend and the contracts, giving the precise semantics of each functionality, are available in the source code as additional documentation to the user.

The first version of the containers library used to benefit from a specialized handling, but it was dropped in the more recent versions as it was not effective. Mostly, this choice comes from the fact that there is no good fit in the underlying logic used by the background solvers of SPARK for these concepts. It is liable to change as these solvers evolve. In particular, investigating the theories offered for sequences and sets in the most recent versions of the underlying provers would be of interest [3, 17]. Instead, the handling of the containers library relies on a hybrid solution. The library is not hardcoded, so the container types and their primitives are treated as abstract types with uninterpreted functions and the only available information comes from the Ada contracts in the functional container API. However, it makes use of several advanced annotation features that have been introduced in the tool along the years especially for this purpose. This schema has the advantage of being light-weight, as it mostly relies on the default handling, and of scaling well as new containers are added. In addition, the annotation features introduced for the containers can be reused in user code, even if some care is sometimes needed, as they can introduce additional assumptions which are not discharged by the verification tool. In the rest of this section, we go over some of these features to explain and motivate them.

4.1 Iteration and quantification

In Ada, `for` loops can be used to iterate over the valid cursors or the elements of a standard container. Unfortunately, this feature is not compatible with the SPARK tool as it relies on aliasing between cursors and containers. A similar functionality has been introduced for formal and functional containers. As explained in Sect. 2, cursors for SPARK containers are implemented as indices in the array constituting the underlying memory of the container to avoid aliasing. To enable iteration over SPARK containers, iteration primitives can be associated to a container type thanks to the `Iterable` annotation. In its most basic form, it provides three functions `First`, `Next`, and `Has_Element`, which allow iterating over the container using a cursor type. A `for` loop over the valid

```
type Container is private with
  Iterable => (First       => First,
               Next        => Next,
               Has_Element => Has_Element,
               Element     => Element);

type Cursor is ...;
function First (C : Container) return Cursor;
function Next
  (C : Container; P : Cursor) return Cursor;
function Has_Element
  (C : Container; P : Cursor) return Boolean;
function Element
  (C : Container; P : Cursor) return Elem_T;

--  A for loop over valid cursors in a
--  container.
for P in C loop
  Process (C, P);
end loop;

--  A for loop over elements of a container
for E of C loop
  Process_E (E);
end loop;

--  The expansion of the previous loop
declare
  P : Cursor := First (C);
begin
  while Has_Element (C, P) loop
    declare
      E : constant Elem_T := Element (C, P);
    begin
      Process_E (E);
    end;
    P := Next (C, P);
  end loop;
end;
```

Fig. 16 The `Iterable` annotation supplies iteration primitives for a container type. They are used to expand `for` loops over valid cursors or elements of a container into `while` loops.

cursors in a container is then syntactic sugar for a `while` loop using these primitives. If an additional `Element` function is supplied, it is also possible to iterate directly over the elements of the container. Different keywords are used to tell the difference between the two forms, `in` for the cursors, like for the values of an integer range, and `of` for the elements. Figure 16 demonstrates this annotation.

It is also possible to quantify over valid cursors and elements of a container whose type is annotated with `Iterable`. Just like `for` loops, quantified expressions can be expanded into `while` loops to be evaluated at runtime. For formal analysis, however, an expansion into a loop is inadequate, as loops are difficult to handle in deductive verification. As first-order logic is supported by the background solvers, it is better to keep the quantifiers. To be compliant with the executional semantics, we would need to quantify over cursors which are reachable from `First` through `Next` in the container. How-

```
type Interval is record
  First, Last : Big_Int;
end record
  with Iterable =>
    (First       => First,
     Next        => Next,
     Has_Element => In_Rng);

function First (I : Interval) return Big_Int
is (I.First);

function Next (I : Interval; X : Big_Int)
  return Big_Int is (X + 1);

function In_Rng (I : Interval; X : Big_Int)
  return Boolean
is (I.First ≤ X and X ≤ I.Last);

function In_Rng_2 (I : Interval; X : Big_Int)
  return Boolean
is (X ≤ I.Last);
```

Fig. 17 Use of the `Iterable` annotation to supply iteration for a range of mathematical integers. Using `In_Rng_2` instead of `In_Rng` would give the same executable semantics, but would invalidate the approximation made by SPARK to handle quantification over intervals.

ever, as automated solvers are not very good with reachability, the SPARK tool approximates this property using the `Has_Element` function: a quantified expression over the valid cursors in a container `C` is expanded into a quantification over all cursors `P` for which `Has_Element (C, P)` returns True. More precisely, the universally quantified expression `(for all P in C => Prop (C, P))` is transformed into the first order formula `(for all P : Cursor => (if Has_Element (C, P) then Prop (C, P)))`. Note that the second expression is not valid Ada, as quantification is only allowed over integer ranges or content of arrays. The containers library has been designed so that this approximation is valid. Namely, every cursor `P` for which `Has_Element (C, P)` returns True is reachable in the container `C` using `First` and `Next`. As the SPARK tool does not attempt to verify this property in general, users should make sure that the approximation is correct if they reuse the `Iterable` annotation on their own data-structures. As an example, consider a range of mathematical integers encoded as a pair. Iteration over values of this range can be provided using the `Iterable` aspect as shown in Fig. 17. The approximation performed by proof is correct here, as `In_Rng` returns True if and only if the integer is in the range. However, using the simpler `In_Rng_2` function would be incorrect, as `In_Rng_2` returns true on integers that cannot be accessed using `First` and `Next`.

As quantifiers tend to be complex to handle in automated verification, it is important to keep them as simple as possible. As an example, when quantifying over the elements of a functional set, refering to cursors valid in the set for whichever definition of cursors is suitable for iterating over the underlying data-structure is not appropriate. It is more efficient to consider all elements on which the `Contains` function returns True. The `Iterable_For_Proof` annotation can be used to tune further the expansion of quantification over elements of a container. It comes in two variants. The first one allows specifying a function which can be used to decide directly whether the element is in the container. It is used in particular with the `Contains` function of functional sets and the `Has_Key` function of functional maps. For example, a quantified expression over the elements of a functional set `(for all E of S => Prop (E))` is transformed into the first-order formula `(for all E : Elem_T => (if Contains (C, E) then Prop (E)))`. The other provides a model function which computes another (simpler) container with the same type of elements. The quantification is done over the elements of the model instead. This is used in particular for formal containers, so that the quantification is done on the functional container working as its model. Here again, the expansion might not be correct if the refined quantification does not traverse the same elements as the basic one.

As explained in Sect. 2, the functional sets and maps are parametrized by a user-provided equivalence relation on elements or keys. This relation is used when defining which elements or keys are in the container. This makes designing an appropriate way to evaluate quantified expressions over functional sets and maps at runtime complex and ill-advised. The iteration would have to traverse all elements of the equivalence classes, which might theoretically not even be finite, for example if we store big integers inside a set. For this reason, the iteration primitives over functional sets and maps have been restricted so that trying to execute them would result into a build failure. They can only be used inside disabled ghost code.[4]

To allow iterating over the elements of these containers, an alternative definition of the `Iterable` annotation is provided. It relies on a function `Choose` that returns an unspecified witness of one of the equivalence classes contained in the container. Note that `Choose` is underspecified but not nondeterministic. It will always return the same witness, it is just unspecified for formal verification which one it will be. A loop using this alternative iteration schema to traverse a functional set is given in Fig. 18. The cursor here is a subset containing all the elements which have not been traversed yet. At each iteration, the element to process is the one returned by `Choose`. Its equivalence class is then removed from the set used as a cursor before the next iteration.

Because functional sets and maps are implemented as sequences with no repetitions,[5] this iteration schema is in fact

[4] Ghost code can be either executed, for runtime monitoring for example, or erased by the compiler. Contracts are a special case of ghost code.

[5] This is because neither a hash function nor a comparison operator is required to instantiate functional sets as explained in Sect. 2.

```
-- A loop iterating over a functional set
for E of Iterate (S) loop
  Process (E);
end loop;

--  The expansion for the previous loop
declare
  P : Set := Get_Set (Iterate (S));
begin
  while not Is_Empty (P) loop
    declare
      E : constant Elem_T := Choose (P);
    begin
      Process (E);
    end;
    P := Remove (P, Choose (P));
  end loop;
end;
```

Fig. 18 A `for` loop over the elements of a functional set. The `Iterate` function returns an object of the `Iterable_Set` type which is basically a wrapper over sets used to redefine the iteration primitives. In the expanded loop, cursors are sets. At each iteration, the element to process is returned by the `Choose` function. Its equivalence class is then removed from the set. The loop stops when the set is empty.

efficient. Indeed, the `Choose` function returns the last element of the sequence so that removing this element from the sequence will not cause any copy. Instead, the underlying sequence will be shared and the index designating the last element of the particular sequence will be decreased. Using subsets or submaps as cursors is also convenient for users when verifying the loop, as it makes it easy to express which elements have already been traversed. Figure 19 gives an example of a loop creating a new map from an existing one by applying a function `F` to each of its elements. It has been annotated with a loop invariant which sumarizes what was done in the loop so it can be verified by SPARK. Using a submap as a cursor makes it easy to express which associations have been added for the keys that have already been traversed, even though the iteration order over functional maps is unspecified.

4.2 Logical equality

Another advanced annotation feature used inside the containers library has to do with equality. In contracts, it is rather common to use the equality symbol to express that some parts of an object are preserved by a function. It is the case in particular in the container library, as most functionalities on containers preserve the vast majority of their elements. It is the case, for example, of the `Add` function on functional sequences, which is shown in Fig. 20. In its postcondition, the equality symbol is used both to say that the last element is the one that was inserted and to express that other elements come from the input sequence.

As contracts use the normal semantics of Ada, occurrences of the equality symbol in the specification are interpreted as standard Ada equality. The language defines equality operators for most types. Their semantics is arguably what a programmer would expect in most contexts. However, when two objects are Ada-equal, it does not mean that they are indistinguishable in general. For example, floating-point equality returns True on +0 and −0, and array equality only compares elements, even though arrays in Ada can start at different indices. In fact, Ada equality is not even necessarily an equivalence relation, as users can redefine the equality symbol on types as they see fit. This redefined equality can even be used implicitly in the predefined equality of composite types.

This interpretation of the equality symbol is less than ideal to express the preservation of elements in the containers library for the following reasons. First, it might not be strong enough to verify user programs calling the container primitives. Indeed, it does not permit to deduce that all functions applied to preserved objects return the same result before and after the call, as this is not entailed by Ada equality in general.

```
for P in Iterate (M) loop
  --  Keys of P have not been traversed yet
  pragma Loop_Invariant
    (for all K of P ⇒ not Has_Key (R, K));
  --  For each traversed key of M, a mapping
  --  has been added to R.
  pragma Loop_Invariant
    (for all K of M ⇒ Has_Key (P, K) or
      (Has_Key (R, K)
       and Get (R, K) = F (Get (M, K))));

  --  Add a mapping for Choose (P) in R
  declare
    K : constant Key_T := Choose (P);
  begin
    R := Add (R, K, F (Get (P, K)));
  end;
end loop;
```

Fig. 19 A loop creating a new map from an existing one by applying a function `F` to each of its elements. It has been annotated with loop invariants which summarize what has been done since the beginning of the loop. Using a submap as a cursor makes it easy to express even though the iteration order over functional maps is unspecified. Note that, if it was not for the loop invariant, a loop over the elements of the map M could have been used instead of a loop of its valid cursors.

```
function Add
  (S : Sequence; E : Elem_T) return Sequence
with
  Post ⇒ Last (Add'Result) = Last (S) + 1
    and Get (Add'Result, Last (S) + 1) = E
    and (for all I in 1 .. Last (S) ⇒
           Get (Add'Result, I) = Get (S, I));
```

Fig. 20 Contract of the `Add` function, which adds an element at the end of an existing sequence. The last element of the sequence is the new element and all the other elements are preserved.

```
function Logic_Eq
  (X, Y : Elem_T) return Boolean
with Annotate => (GNATprove, Logical_Equal);

function Copy (E : Elem_T) return Elem_T is
   (E);

function Add
  (S : Sequence; E : Elem_T) return Sequence
with
  Post => Last (Add'Result) = Last (S) + 1
    and Logic_Eq
          (Get (Add'Result, Last (S) + 1),
           Copy (E))
    and (for all I in 1 .. Last (S) =>
           Logic_Eq (Get (Add'Result, I),
                     Get (S, I)));
```

Fig. 21 Contract of the `Add` function using logical equality to state that elements are preserved or copied. Note the use of a `Copy` function, which simply returns its parameter. It is necessary to express that the element added is a copy of the parameter, which might be of a different dynamic type after the copy, in the case of tagged types (equivalent of "classes" in Ada).

The second reason concerns the efficiency of the verification process. Ada equality is not, and cannot be in general, supported in a built-in way by the underlying solvers. Therefore, when the Ada contract contains a simple equality, what the provers see is a big predicate with quantifiers, arithmetic symbols, and possibly user-defined function calls. So even if Ada equality happens to be enough to deduce that some properties are preserved by a container primitive, the tool might not effectively be able to deduce it given the level of complexity of the generated formula.

There is a built-in equality symbol which is supported in the underlying logic of SPARK. It is called *logical equality*. Two values are logically equal if they are exactly the same, namely, there is no way to tell the difference between them. Based on this definition, the solvers can efficiently deduce that all function calls on logically equal parameters return the same result. Using this equality instead of the Ada one for preserved values in the contracts of the container library is therefore highly desirable.

The `Logical_Equal` annotation can be supplied on a function with an equality profile to tell the SPARK tool that it should be recognized as the logical equality symbol in the underlying logic. It is effectively as if such functions had an implicit postcondition stating that they return True if and only if their parameters are logically equal. Figure 21 shows how such a function can be defined and used in the postcondition of the `Add` function on sequences. Note that such a function cannot always be implemented in an executable way. In particular, it can occur that the model of an Ada concept in the underlying logic introduces elements which have no counterparts in Ada. As for quantification over functional sets, uses of the logical equality functions introduced

```
--  Contains returns the same result on all
--  equivalent elements.
procedure Lemma_Contains_Equivalent
  (S : Set; E : Elem_T)
with
  Ghost,
  Pre  => not Contains (S, E),
  Post =>
    (for all F of S => not Equivalent (E, F));
```

Fig. 22 Lemma used to state that the `Contains` function on functional sets works modulo equivalence

in the functional containers library are restricted to disabled ghost code, and an incorrect usage results in a build failure. Because of this dependency on the internal encoding of the SPARK tool, the `Logical_Equal` annotation can hardly be used for anything but preserved and copied values, even in user code. However, unlike iteration, this feature does not introduce any additional assumptions, as the compatibility of the supplied body, if any, is checked by the tool.

4.3 Automatically instantiated lemmas

The last annotation feature discussed in this section is the ability to turn ghost procedures into lemmas in the underlying logic. In deductive verification, using a ghost function with no effects as a lemma is common practice [14, 18]. Such functions are called lemma functions or, in the case of SPARK, *lemma procedures*. The idea is to use modular analysis as a way to extract a part of the reasoning and therefore simplify the verification. To do so, the lemma procedure is annotated with a precondition, containing the premises of the lemma and a postcondition expressing its result. As an example, Fig. 22 shows the lemma which is used to state that the `Contains` function on functional sets returns the same value on all elements of an equivalence class. It reads: if a given element is not contained in a functional set, then the set does not contain any equivalent element. Since deductive analysis is modular on a per subprogram basis, when verifying the procedure itself, the tool tries to ascertain that its postcondition, the result of the lemma, follows from the precondition, its premises, in every context. Said otherwise, it verifies the lemma. Then, each time the lemma procedure is called, it verifies its precondition, the premises, and assumes the result without trying to reprove it.

In this design, lemma procedures suffer from a big drawback, they need to be called explicitly each time the property is needed. This can be cumbersome, as it requires a fine grained understanding of how the proof works for the user to know when to actually call the lemma. To workaround this issue, lemma procedures can be annotated with the `Automatic_Instantiation` annotation. It instructs the tool to turn a particular lemma procedure into an axiom so that the property is readily available to the background

```
--  Test shall return the same value on
--  equivalent elements.
function Eq_Compatible
  (S    : Set;
   Test : not null access
     function (E : Elem_T) return Boolean)
   return Boolean
is
  (for all E1 of S ⇒
    (for all E2 of S ⇒
      (if Equivalent_Elements (E1, E2)
       then Test (E1) = Test (E2))))
with Ghost;

--  Count returns the same value on sets
--  containing the same equivalence classes.
procedure Lemma_Count_Eq
  (S1, S2 : Set;
   Test   : not null access
     function (E : Elem_T) return Boolean)
with Ghost,
  Annotate ⇒
    (GNATprove, Automatic_Instantiation),
  Pre  ⇒ Eq_Compatible (S1, Test)
    and S1 = S2,
  Post ⇒ Count (S1, Test) = Count (S2, Test);
```

Fig. 23 Lemma stating that the `Count` function returns the same result on equivalent sets. It uses the Ada equality on functional sets, which is defined to return True iff both sets contain the same equivalence classes.

solvers. As provers can be overwhelmed by too many axioms, it is important to avoid including unnecessary axioms in verification conditions. To this aim, each lemma procedure annotated with `Automatic_Instantiation` should be declared directly after a function with which it is associated. The generated axiom is only made available in formulas in which the associated function is called. As an example, the lemma procedure in Fig. 22 is associated to the function `Contains`. Therefore, it is only available in verification conditions in which `Contains` is called. This is similar to how postconditions of functions are handled. The advantage of using a lemma procedure instead of a direct definition in the postcondition is that it is implicitly quantified over all possible values of the procedure's parameters. It is interesting when the lemma procedure takes more parameters than its associated function. As an example, Fig. 23 shows the lemma procedure used to state that the `Count` higher-order function, which counts elements with a given property in a functional set, returns the same result on equivalent sets. As it mentions two different sets, it could not easily be turned into a postcondition on the result of the function `Count`.

As lemma procedures are verified using the SPARK tool, the `Automatic_Instantiation` annotation cannot compromise the soundness of the verification. However, adding too many such axioms can lead to a loss of efficiency as they increase the size of the verification conditions. Therefore, using this annotation is a trade-off and should be reserved to lemmas used often enough to warrant the additional burden.

5 Related work

Collections are generally considered to be useful when writing program specifications. As a result, most verification tools support some kind of specification-oriented collections in their input language. For example, the standard library of the Why3 language used as a backend of SPARK provides polymorphic lists implemented as abstract data types, as well as polymorphic sets and multisets, specified through an axiomatization. Dafny, a verification language developed at Microsoft Research and currently used to verify protocols at Amazon Web Services, also provides collections such as sequences and sets with comprehensions [13].

Satisfiability Modulo Theory (SMT) solvers are generally used as a backend of program verifiers. As a consequence, significant research effort has been invested in coming up with efficient decision procedures for collections in these solvers. For example, a decision procedure for a theory of finite sets has been supported by the CVC4 solver [3] since 2016, and the last version of the prover, cvc5, also supports sequences that can be used to model arrays and vectors [17].

Using simple collections as ghost models of complex data-structures is a widely used technique in proof of programs. As an example, the specification of the Eiffel-Base2 general-purpose containers library verified by Polikarpova et al. uses collections coming from the Mathematical Model Library (MML) : sequences as models of linked lists, maps for hash tables, etc. These collections benefit from custom support in the underlying solver AutoProof [16]. This technique was used successfully to verify numerous data-structures in Why3, such as hashed tables or AVL trees [6]. Blanchard et al. also resorted to using ghost arrays as a model of their lists for their verification of the linked list module of the Contiki OS with the WP plugin of Frama-C [5].

6 Conclusion

Functional containers are used to represent collections in the specification of SPARK programs. They offer a simple, high-level representation of a container, that is both easy to understand for a reader and easy to reason with for the solvers. They are executable, so it is possible to test the specification, or even to use them in actual code.

Through the years, they have been used to annotate or verify various kinds of containers, from the formal container library of SPARK (which are specified with functional containers but not proved) to ownership based recursive data-structures. Proof by refinement, though not natively sup-

ported in SPARK, can be achieved using these containers by creating several layers of models.

Going further, functional containers provide a way to describe state which only exists in the specification. It makes it possible to model properties which are not generally in the domain of SPARK, like some restricted form of temporal logic, through a history, or aliasing and simple memory separation.

The containers library does not currently benefit from a built-in handling in the SPARK tool. However, several annotation features have been introduced so they can be treated as efficiently as possible. This hybrid handling has the advantage of making the container library easier to maintain and to extend. The annotation features can also be reused on user-defined data-structures.

References

1. Aiello, M.A., Dross, C., Rogers, P., Humphrey, L., Hamil, J.: Practical application of SPARK to OpenUxAS. In: International Symposium on Formal Methods, pp. 751–761. Springer, Berlin (2019)
2. Backes, J., Bolignano, P., Cook, B., Gacek, A., Luckow, K.S., Rungta, N., Schaef, M., Schlesinger, C., Tanash, R., Varming, C., et al.: One-click formal methods. IEEE Softw. **36**(6), 61–65 (2019)
3. Bansal, K., Reynolds, A., Barrett, C., Tinelli, C.: A new decision procedure for finite sets and cardinality constraints in SMT. In: International Joint Conference on Automated Reasoning, pp. 82–98. Springer, Berlin (2016)
4. Barnes, J.: SPARK: The Proven Approach to High Integrity Software. Altran Praxis (2012)
5. Blanchard, A., Kosmatov, N., Loulergue, F.: Ghosts for lists: a critical module of Contiki verified in Frama-C. In: NASA Formal Methods Symposium, pp. 37–53. Springer, Berlin (2018)
6. Clochard, M.: Automatically verified implementation of data structures based on AVL trees. In: Working Conference on Verified Software: Theories, Tools, and Experiments, pp. 167–180. Springer, Berlin (2014)
7. Dross, C., Kanig, J.: Recursive data structures in SPARK. In: International Conference on Computer Aided Verification, pp. 178–189. Springer, Berlin (2020)
8. Dross, C., Kanig, J.: Making proofs of floating-point programs accessible to regular developers. In: Software Verification, pp. 7–24. Springer, Berlin (2021)
9. Dross, C., Moy, Y.: Abstract software specifications and automatic proof of refinement. In: International Conference on Reliability, Safety, and Security of Railway Systems, pp. 215–230. Springer, Berlin (2016)
10. Dross, C., Moy, Y.: Auto-active proof of red-black trees in SPARK. In: NASA Formal Methods Symposium, pp. 68–83. Springer, Berlin (2017)
11. Dross, C., Filliâtre, J.C., Moy, Y.: Correct code containing containers. In: International Conference on Tests and Proofs, pp. 102–118. Springer, Berlin (2011)
12. Filliâtre, J.C., Paskevich, A.: Why3—where programs meet provers. In: European Symposium on Programming, pp. 125–128. Springer, Berlin (2013)
13. Ford, R.L., Leino, K.R.M.: (2017). Dafny reference manual
14. Leino, K.R.M., Moskal, M.: Co-induction simply: automatic co-inductive proofs in a program verifier. In: FM 2014: Formal Methods: 19th International Symposium, Proceedings 19, Singapore, May 12–16, 2014, pp. 382–398. Springer, Berlin (2014)
15. O'Hearn, P.W.: Continuous reasoning: scaling the impact of formal methods. In: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, pp. 13–25 (2018)
16. Polikarpova, N., Tschannen, J., Furia, C.A.: A fully verified container library. In: International Symposium on Formal Methods, pp. 414–434. Springer, Berlin (2015)
17. Sheng, Y., Nötzli, A., Reynolds, A., Zohar, Y., Dill, D., Grieskamp, W., Park, J., Qadeer, S., Barrett, C., Tinelli, C.: Reasoning about vectors using an SMT theory of sequences (2022). arXiv:2205.08095
18. Volkov, G., Mandrykin, M., Efremov, D.: Lemma functions for Frama-C: C programs as proofs. In: 2018 Ivannikov Ispras Open Conference (ISPRAS), pp. 31–38. IEEE Press, New York (2018)


AdaCore
Build Software
that Matters
adacore.com