Explicit Assumptions - A Prenup for Marrying Static and Dynamic Program Verification

Johannes Kanig[2], Rod Chapman[1], Cyrille Comar[2], Jerôme Guitton[2], Yannick Moy[2], and Emyr Rees[1]

[1] Altran UK, 22 St Lawrence Street, Bath BA1 1AN (United Kingdom)
{rod.chapman, emyr.rees}@altran.com
[2] AdaCore, 46 rue d'Amsterdam, F-75009 Paris (France)
{comar, guitton, kanig, moy}@adacore.com

Abstract. Formal modular verification of software is based on assume-guarantee reasoning, where each software module is shown to provide some guarantees under certain assumptions and an overall argument linking results for individual modules justifies the correctness of the approach. However, formal verification is almost never applied to the entire code, posing a potential soundness risk if some assumptions are not verified. In this paper, we show how this problem was addressed in an industrial project using the SPARK formal verification technology, developed at Altran UK. Based on this and similar experiences, we propose a partial automation of this process, using the notion of explicit assumptions. This partial automation may have the role of an enabler for formal verification, allowing the application of the technology to isolated modules of a code base while simultaneously controlling the risk of invalid assumptions. We demonstrate a possible application of this concept for the fine-grain integration of formal verification and testing of Ada programs.

Keywords Formal methods, Program Verification, Test and Proof, Assumptions

1 Introduction

Formal modular verification of software is based on assume-guarantee reasoning, where each software module is shown to provide some guarantees under certain assumptions, and an overall argument linking results for individual modules justifies the correctness of the approach. Typically, the assumptions for the analysis of one module are part of the guarantees which are provided by the analysis of other modules. The framework for assume-guarantee reasoning should be carefully designed to avoid possible unsoundness in this circular justification. For software, a prevalent framework for assume-guarantee reasoning is Hoare logic, where subprograms[3] are taken as the software modules, and subprogram contracts (precondition and postcondition) define the assumptions and guarantees.

[3] In this paper, we use the term *subprogram* to designate procedures and functions, and reserve the more common term of *function* to subprograms with a return value.

Formal verification tools based on Hoare logic analyze a subprogram without looking at the implementation of other subprograms, but only at their contract.

Although verification is done modularly, it is seldom the case that the results of verification are also presented modularly. It is tempting to only show which components have been verified (the guarantees), omitting the assumptions on which these results depend. This is indeed what many tools do, including the SPARK tools co-developed by Altran UK and AdaCore. In theory, the correctness of the approach would depend, among other things, on formal verification being applied to all parts of the software, which is never the case for industrial projects. Even when considered desirable to maximize formal verification, there are various reasons for not applying it to all components: too difficult, too costly, outside the scope of the method or tool, *etc.* In practice, expertise in the formal verification method and tool is required to manually justify that the implicit assumptions made by the tool are valid.

The care with which this manual analysis must be carried out is an incentive for system designers to minimize boundaries between formally verified modules and modules that are verified by other means. For example, this can be achieved by formally verifying the entire code except some difficult-to-verify driver code, or by formally verifying only a very critical core component of the system. However, such a monolithic approach is hindering a wider adoption of formal methods. Modules that are not formally verified are usually verified using other methods, often by testing. If combining verification results of *e.g.,* proof and test was easy, projects could freely choose the verification method to apply to a given component, based on tool capabilities and verification objectives. We propose to facilitate the effective combination of modular formal verification and other methods for the verification of critical software by extending the application of assume-guarantee reasoning to these other methods.

1.1 SPARK

SPARK is 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 [3,11].

In the versions of SPARK up to SPARK 2005, specifications are written as special annotations in comments. Since version SPARK 2014 [10], specifications are written as special Ada constructs attached to declarations. In particular, various contracts can be attached to subprograms: data flow contracts (introduced by `global`), information flow contracts, and functional contracts (preconditions and postconditions, introduced respectively by `pre` and `post`). An important difference between SPARK 2005 and SPARK 2014 is that functional contracts

are executable in SPARK 2014, which greatly facilitates the combination between test and proof (see Section 4). The definition of the language subset is motivated by the simplicity and feasability of formal analysis and the need for an unambiguous semantics. Tools are available that provide flow analysis and proof of SPARK programs.

Flow analysis checks correct access to data in the program: correct access to global variables (as specified in data and information flow contracts) and correct access to initialized data. Proof is used to demonstrate that the program is free from run-time errors such as arithmetic overflow, buffer overflow and division-by-zero, and that the functional contracts are correctly implemented.

The different analyses support each other - for example, proof assumes that data flow analysis has been run without errors, which ensures that all variables are initialized to a well-defined value before use, that no side-effects appear in expressions and function calls, and that variables are not aliased. The latter point is partly achieved by excluding access (pointer) types from the language, and completed by a simple static analysis. For the purposes of this paper, we consider the SPARK analysis as a whole in Section 4 and will discuss interaction between the different analyses in Section 5.

1.2 Related Work

Neither the idea of explicit assumptions nor the idea of combining different types of analyses on a project are new. However, the focus of this line of research has been to show how different verification techniques can collaborate *on the same code* and support each other's assumptions. Examples are the explicit assumptions of Christakis *et al.* [5], the combination of analyses [7] in Frama-C [9], the EVE tool for Eiffel [13] and the work of Ahrendt *et al.* [1]. In contrast, we focus on the combination of verification results for *different* modules. Another line of research is the Evidential Tool Bus (ETB [8]), which concentrates on how to build a safe infrastructure for combining verification results from different sources and tracking the different claims and supporting evidence. An ETB could be used as the backbone for the framework that we describe in this paper.

1.3 Outline

In Section 2, we describe how the problem of heterogeneous verification was addressed in an industrial project using the SPARK formal verification technology, developed at Altran UK, using an ad-hoc methodology. In Section 3, we propose a framework for combining the results of different verification methods that can be partly automated, and thus lends itself to a more liberal combination of verification methods. In Sections 4 and 5, we present our experiments to combine at coarse-grain and fine-grain levels proof and test on Ada programs, using the SPARK technology that we develop.

2 Assumptions Management in a Large Safety-Critical Project

Project X[4] is a large, mission-critical, distributed application developed by Altran UK, and now in operational service. The software consists of several programs that execute concurrently on a network of servers and user workstations. The latter machines include a user-interface that is based on the X11/Motif UI framework.

Almost all of the software for Project X is written in SPARK 2005 and is subject to extensive formal verification with the SPARK 2005 toolset. Two other languages are used, though:

- Ada (not SPARK subset, but still subject to a project coding standard) is used where SPARK units need to call operating-system or compiler-defined run-time libraries.
- C code is used to form a layer between the SPARK code and the underlying X11/Motif libraries, which is implemented in C.

One program, called the UI Engine, is a multi-task SPARK program that uses the RavenSPARK subset of Ada's tasking features [3]. This mitigates many common problems with concurrent programming, such as deadlock and priority inversion. The C code is only ever called from a single task of the main SPARK program - a major simplification which prevents interference between the implementation languages, because the C code does not have global side effects. Also, in this way the C code does not need to worry about reentrance. The UI engine component is 87kloc (logical lines of code), comprising of 61kloc SPARK and 26kloc MISRA C.

2.1 The Requirements Satisfaction Argument

The fitness-for-purpose of Project X is justified at the top-level by a "Requirements Satisfaction Argument". This is essentially structured as a tree of goals, justifications, assumptions, and evidence, expressed in the Goal Structured Notation (GSN).

A large section of the GSN is devoted to non-interference arguments that form the core of the safety argument for Project X. Part of that non-interference argument includes detailed justifications for the integration of software written in multiple languages, and the prevention of defects that could arise. The leaves of the GSN typically refer to verification evidence (*e.g.,* test specifications, results, provenance of COTS components, static analysis results and so on) or standards (such as the coding standards for SPARK and C used by the project).

2.2 From SPARK to C (and Back Again)

In Project X, SPARK code calls C code to implement various user-interface elements. Beyond that point, the formal analyses offered by the SPARK toolset

[4] This is not its actual name, which we cannot mention.

Assumption	How verified
Parameter types match	AUTO
Variables initialized	MISRA
Outputs in expected subtype	REVIEW, TEST
No side effects	MISRA
No aliasing	MISRA, REVIEW
Data flow contract respected	REVIEW
No thread/task interaction	REVIEW
No dynamic allocation	MISRA
Functional contract respected	REVIEW, TEST
Absence of run-time errors	TEST

Table 1: SPARK to C assumptions and verification

are not available, so we cannot rely on these analyses to prove the assumptions made to analyze the SPARK code. Instead, the project manages an explicit list of assumptions that must be enforced across each such boundary. Essentially, the SPARK code assumes that the called C function is "well-behaved" according to a set of implicit project-wide rules (*e.g.,* the function terminates, and parameters are passed using the expected types and mechanism) and the explicit SPARK contract (*e.g.,* precondition and postcondition) applied to the SPARK specification of that function.

Each of these assumptions is verified on the C code through a combination of one or more of:

- AUTO. Automated code generation. In particular, the Ada and C type declarations that are used to communicate across the language boundary are automatically generated from a single description.
- MISRA. Automated static analysis using the MISRA C:2004 rules [2].
- REVIEW. Checklist-driven manual code review. In particular, parameter passing mechanisms and types are carefully reviewed to ensure they match across such a language boundary.
- TEST. Specific unit test objectives.

The set of MISRA rules enforced and the review checklist items were chosen to cover the assumptions needed to support the verification of the SPARK units. The project maintains a detailed analysis of every MISRA rule and how its use meets the assumptions required by the SPARK analysis. A small number of MISRA rules are not used by the project, or deviations may be justified on a case-by-case basis. Again, detailed records are maintained to make sure that these deviations do not undermine the analysis of the SPARK code. Table 1 shows how each major assumption made by the SPARK code is verified in the C code.

```
procedure Set_Off_Button
   (Button_Enabled    : Boolean;
    Background_Colour : Background_Colour_T);
--# global in out Shutdown.Error_Flag;
--#        in out Shutdown.Do_Shutdown_SO;
```

(a) The SPARK specification of `Set_Off_Button`

```
void TB_Set_Off_Button_SC (
   const bool                              Button_Enabled,
   const HMI_Types__Background_Colour_T    Background_Colour,
         HMI_Types__Status_T*              Error);
```

(b) The C specification of `TB_Set_Off_Button_SC`

```
procedure Set_Off_Button
   (Button_Enabled    : Boolean;
    Background_Colour : Background_Colour_T)
is
   Button_Enabled_C    : C_Base_Types.C_Bool;
   Background_Colour_C : HMI_Types.C.Background_Colour_T;
   Error               : HMI_Types.C.Status_T;

   -- Here is the interface to C function TB_Set_Off_Button_SC
   procedure TB_Set_Off_Button
      (Button_Enabled_C    :     C_Base_Types.C_Bool;
       Background_Colour_C :     HMI_Types.C.Background_Colour_T;
       Error               : out HMI_Types.C.Status_T);
   pragma Import (C, TB_Set_Off_Button, "TB_Set_Off_Button_SC");
begin
   Button_Enabled_C  := C_Base_Types.To_C_Bool (Button_Enabled);
   Background_Colour_C :=
      HMI_Types.C.To_C.Background_Colour_T (Background_Colour);

   -- Call to C here
   TB_Set_Off_Button
      (Button_Enabled_C    => Button_Enabled_C,
       Background_Colour_C => Background_Colour_C,
       Error               => Error);

   Common_Error.Log_And_Handle_If_Error
      (Message => Error,
       Gate    => HMI_DM_Fatal_Error_In_C_Code);
end Set_Off_Button;
```

(c) The SPARK body of `Set_Off_Button`

Fig. 1: Excerpt of mixed SPARK/C code in Project X.

```
/* PRQA S:R14_1_S002 1503 1 */
void TB_Set_Off_Button_SC (
   const bool                              Button_Enabled,
   const HMI_Types__Background_Colour_T  Background_Colour,
         HMI_Types__Status_T*            Error)
{
   CF_Set_OK (Error);
   CF_Set_Widget_Sensitivity(tb_OffButton, Button_Enabled);

   switch (Background_Colour)
   {
      case HMI_Types__Active_Colour:
      {
         XtVaSetValues (tb_OffButton,
                        XmNbackground,     /* PRQA S:R11_5_S001 0311 */
                        Alert_Colour,
                        NULL);
         break;
      }
      case HMI_Types__No_Colour:
      {
         XtVaSetValues (tb_OffButton,
                        XmNbackground,     /* PRQA S:R11_5_S001 0311 */
                        Background_Colour,
                        NULL);
         break;
      }
      default:  /* PRQA S:R14_1_S001 2018 */
      {
         CF_Set_Not_OK_Str_Int (Error,
                                "Invalid Off Background Colour, Enum ",
                                (int) Background_Colour);
         /* Note that analysis deemed that a failure to decode an */
         /* enumeration is likely to have been caused by a memory */
         /* corruption, and to continue processing would be unsafe. */
         /* Assign the category as force shutdown. */
         CF_Set_Category (Error,
                          HMI_Common_Types__Force_Shutdown,
                          CF_On_Error_Abort);
      } break;
   }
}
```

(d) The C implementation of `TB_Set_Off_Button_SC`

Fig. 1: (continued) Excerpt of mixed SPARK/C code in Project X.

2.3 Example

This section shows an example of how SPARK code interfaces to a UI function that is written in C. The SPARK specification is given Listing 1a. Note the data flow contract introduced by `global`. This specifies the frame condition of the procedure - stating exactly the set of objects that may be referenced and/or updated by the procecure and (implicitly) defining that *no other* objects are used. In this case, we see that the procedure may read and update two objects in package Shutdown, both of which record the need to terminate the system in response to a fatal error.

The SPARK implementation is given in Listing 1c. Ada's pragma Import here specifies Convention "C" for the nested procedure - this instructs the compiler to pass parameters as would be expected in C, according to a set of rules given in the Ada Reference Manual.

The corresponding C header is provided in Listing 1b. Note the careful use of naming conventions here to ease the task of both generating and reviewing the Ada/C interface. Finally, the C implementation is given in Listing 1d. Note that the coding is overtly defensive, dealing with the possibility of a memory corruption leading to the default branch of the switch statement being executed. The "PRQA" comments are instructions to the MISRA analysis tool to suppress particular warnings. All of these comments are collated and verified as part of the satisfaction argument.

Consider one particular verification objective for this code: the output parameter `Error` on the SPARK declaration of the imported procedure `TB_Set_Off_Button`. In SPARK, output parameters *must* be defined by the called procedure in all cases. This ensures that `Error` is properly initialized when calling `Log_And_Handle_Error` inside the body of `Set_Off_Button`. If the body of `TB_Set_Off_Button` were written in SPARK, then the flow analysis engine would verify this obligation, but since the body is in C, additional steps are needed. In this case, an explicit review checklist item requires C function parameters that correspond to SPARK output parameters to be unconditionally initialized - hence the call to `CF_Set_OK` that initializes `Error` at the top of the function body, for cases which do not result in an error.

2.4 Summary

This approach has proven reliable in practice, owing to judicious architectural design, a strong desire to minimize the volume of C code (although 26kloc does still feel a little too large for comfort in an 87kloc application), and strict adherence to design and coding disciplines through automated analysis, review checklists and focussed testing.

The main drawback is the time, expense and paperwork required to maintain the satisfaction argument and its supporting evidence across a long-lived project, which has absorbed several major UI re-designs in its lifetime.

3 Tool Assisted Assumptions Management

Reading the previous section, the reader may ask the questions of how we came up with the left column of Table 1 and how all subprograms at the interface have been identified. This is in fact the result of expert knowledge of the SPARK technology as well as specificities of the project. We want to present here a more systematic way to achieve the same goal.

The work done on Project X to develop Table 1 and to apply it at the boundary between formal verification and other methods can be broken down into three steps:

- listing all assumptions of formal verification,
- verifying the non-formally-verified modules using some other method, so that the previous assumptions are verified, and
- checking that all assumptions have been taken care of.

It is clear that for the first and the last step, tool support is possible and welcome, and this is the topic of this paper. We propose to enhance formal verification tools to not only output verification results, but also the assumptions these results rely on - a more detailed version of the left column of Table 1. As non-formal methods may rely on assumptions as well, we may also require that these other methods explicitly list all their assumptions and guarantees when applied to a module. These should be precise enough to avoid holes in the justification, or subtly different interpretations of the properties in different methods. As an extreme example, "module M is correct" is not at the appropriate level of precision. For formal methods, this requires an explicit enumeration of the usually implicit assumptions made for the verification of a component, like non-aliasing of subprogram parameters, non-interference of subprogram, validity of the data accessed, *etc.* For informal methods, this requires defining methodological assumptions and guarantees that the method relies upon.

Formally, each verification activity is a process whose output is a list of Horn clauses, that is, implications of the form

$$A_1 \wedge A_2 \wedge \cdots \wedge A_n \rightarrow \ C$$

where C is a claim[5], and the A_i are assumptions. The exact form of claims and assumptions differs for each method and tool. The next sections will provide examples based on SPARK.

Compared to the manual process in the previous section, the advantage is that the "checklist" of verification activities at the boundary between verification methods is simply provided by each tool, and does not require that users possess this level of expertise about the tool. The assumptions still need to be verified of course - that is the right column of Table 1.

[5] We prefer to use the word "claim" here over "guarantee", as it more clearly conveys the idea that until the corresponding assumptions are verified, no guarantee can be given.

If explicit assumptions are gathered for all verification activities on the project, we can simply consider together all Horn clauses, and simple queries such as

- Is claim C completely verified?
- On which unverified assumptions is claim C based on?
- Which verification activities are left to do to verify claim C?
- If assumption A turns out to be invalid, which claims does this impact?

can be answered simply by analyzing these Horn clauses.

Other forms of explicit assumptions are possible. Christakis *et al.* [5] and Correnson and Signoles [7] describe assumptions and claims as formulas at program points, which is more precise than our approach. The drawback is the complexity to generate and exploit these assumptions, as their formulation implies the use of a weakest precondition calculus in the former work, or trace semantics in the latter work, in order to interpret the meaning of a formula at a program point. Also, simply formulating these formulas already requires choosing a memory model. Instead, we prefer to see claims and assumptions as well-defined, parameterized properties uniquely identified by a tag. For example, where Christakis *et al.* use the formula $c \neq d$ to express non-aliasing of two parameters at subprogram entry, we prefer to write it as the property $\mathit{Nonaliased}(c, d)$ where the names c and d are given in a format which allows their unique identification, and *Nonaliased* is a tag which is unambiguously defined to mean that two variables do not share any memory. Similarly, to denote an assumption on the precondition of some subprogram p, we would write $\mathit{Pre}(p)$ instead of the formula which constitutes the precondition.

This choice, together with the choice of Horn clauses as data format, makes it much simpler to feed assumptions to existing tools such as the ETB [8] and opens the door to tool assisted assumptions management.

4 Coarse-Grain Assumptions Management

We have described in some previous work [6] a coarse-grain application of the framework described in the previous section, to combine the results of test and proof on Ada programs (proof being restricted to SPARK subprograms) in the context of certification of avionics software following the DO-178C [12] certification standard. In this context, tests with MC/DC coverage [4] or proofs are two acceptable methods to verify a module, where modules here are subprograms. We can reexpress the goal of verifying a subprogram P using Horn clauses as described in the previous section:

$$\begin{aligned} \mathit{Tests\_Passed}(P) \wedge \mathit{MCDC\_Covered}(P) &\rightarrow \mathit{Verified}(P) \\ \mathit{Contract\_Proved}(P) \wedge \mathit{No\_Runtime\_Errors}(P) &\rightarrow \mathit{Verified}(P) \end{aligned}$$

Note that assumptions made during proof are still implicit in the Horn clauses above. Assumptions related to functional contracts, like the guarantee that called

subprograms respect their postcondition, or that the subprogram proved is only called in a context where its precondition holds, are discharged either by the proof of the callees/callers, or by executing the corresponding contracts during the test of the callees/callers. Thus, it is essential for the combination that functional contracts are executable. Other assumptions related to non-aliasing of parameters, or validity of variables, are discharged by having the compiler instrument the tested programs to check the assumptions. Finally, the assumptions that cannot be tested are guaranteed by the combination of a coding standard and a static analysis on the whole program. The coding standard forbids in particular calls through subprogram pointers, so that the call-graph is statically known. The static analysis forbids aliasing between parameters of a call and global variables that appear in the data flow contract for the called subprogram.

We built a prototype tool in Python implementing this approach, allowing users to specify the generic Horn clauses above in some special syntax. This monolithic specially crafted approach for SPARK was not completely satisfying, as it was not easily extensible or customizable by users. This is why we switched to a finer-grain approach where assumptions are explicit.

5 Fine-Grain Assumptions Management

We consider now the combination of verification results for individual subprograms whose declaration is in SPARK. As described in Section 1.1, various contracts can be attached to such subprograms: data flow contracts, information flow contracts, and functional contracts (preconditions and postconditions).

5.1 Claims and Assumptions

We provide a detailed definition of claims and assumptions for SPARK. We assume subprograms are uniquely identified, for example by using their name and the source location of the declaration. We also assume that *calls* to subprograms are uniquely identified, again using *e.g.,* the source location of the call. We use capital letters such as P for subprograms, and write $P@$ to indicate a specific call to a subprogram.

It should be noted that some fundamental assumptions, *e.g.,* correctness of the verification tool, compiler and hardware, are out of scope of this framework and are not taken into account here.

SPARK formal verification tools may be used to ensure that the following claims are satisfied by a subprogram P or a call to P:

- $\mathit{Effects}(P)$ - the subprogram P only reads input variables and writes output variables according to its data flow contract.
- $\mathit{Init}(P)$ - the subprogram P is only called when all its input parameters, and all the input variables in its data flow contract, are initialized.
- $\mathit{Init}(P@)$ - in this specific calling context of P, all its input parameters, and all the input variables in its data flow contract, are initialized.

- $Nonaliasing(P)$ - the subprogram P is not called with parameters which would create aliasing.
- $Nonaliasing(P@)$ - in this specific calling context of P, the values of parameters do not create aliasing.
- $AoRTE(P)$ - the subprogram P is free of run-time errors.
- $Contract(P)$ - the subprogram P respects its contract, that is, the precondition is sufficient to guarantee the postcondition.
- $Pre(P)$ - the subprogram P is only called in a context that respects its precondition.
- $Pre(P@)$ - in this specific calling context of P, its precondition is respected.
- $Term(P)$ - the subprogram P terminates.

The output of the SPARK tools can then be described as follows. Given a subprogram P which contains the calls $R_i@$, if flow analysis is applied without errors, then the following set of Horn clauses holds:

$$\begin{aligned} Effects(R_i) \wedge Init(P) \wedge Nonaliasing(P) \longrightarrow & Effects(P) \wedge \\ & Init(R_i@) \wedge \\ & Nonaliasing(R_i@) \end{aligned} \tag{1}$$

and if proof is applied without unproved properties being reported, then the following set of Horn clauses holds:

$$\begin{aligned} & Effects(R_i) \wedge Init(P) \wedge Nonaliasing(P) \wedge \\ & \quad AoRTE(R_i) \wedge Contract(R_i) \wedge Pre(P) \longrightarrow AoRTE(P) \wedge \\ & \qquad\qquad Pre(R_i@) \end{aligned} \tag{2}$$

$$\begin{aligned} & Effects(R_i) \wedge Init(P) \wedge Nonaliasing(P) \wedge \\ & \qquad Contract(R_i) \longrightarrow Contract(P) \end{aligned} \tag{3}$$

For the sake of succinctness, we have taken the liberty to merge Horn clauses with different conclusions, but identical premises - this is only a shortcut for the equivalent expansion using only Horn clauses. The result of successful flow analysis of subprogram P, as expressed in Formula 1, is that, assuming P's callees respect their data flow contract, and assuming P is always called on initialized inputs and non-aliased inputs/outputs, then P respects its data flow contract, and calls inside P are done in a context which does not introduce uninitialized inputs or aliasing for the callee. For proof, there are in fact two different sets of results and assumptions. The first one, expressed in Formula 2, is that, assuming P's callees respect their data flow contract and their pre/post contract, and they do not raise run-time errors, and assuming P is always called on initialized inputs and non-aliased inputs/outputs, in a context where its precondition holds, then P does not raise run-time errors, and calls inside P are done in a context where their precondition holds. The second one, expressed in Formula 3, is that assuming P's callees respect their data flow contract and their pre/post contract, and assuming P is always called on initialized inputs and non-aliased inputs/outputs, then P also respects its pre/post contract.

Note that the precondition of P is *not* an assumption of Formula 3, because the tag *Contract* already includes the precondition. In this manner, as can be easily seen, Formula 3 propagates assumptions about contracts *down* the call graph, while Formula 2 propagates the assumptions on preconditions *up* the call graph.

Note that *Pre*, *Init* and *Nonaliasing* applied to a subprogram are a bit special: they only appear as assumptions and not as claims. They are in fact assumptions on the calling context, and the only way to discharge them is to verify that they hold for all calling contexts. As a consequence, a non-modular analysis is needed here to identify all calls to a subprogram, so we add Horn clauses of the form:

$$tag(\overline{P@}) \longrightarrow tag(P) \tag{4}$$

where *tag* is any of *Pre*, *Init* and *Nonaliasing* and $\overline{P@}$ are all calls to a given subprogram P. An important special case is that, for main subprograms (which are not called by any other subprogram), we obtain the immediate guarantees of the form:

$$tag(P) \tag{5}$$

By combining Formulas 1, 4, and 5, it can be checked easily that, if formal verification is applied to the entire program[6], then *Effects*(P), *Init*(P) and *Nonaliasing*(P) hold for every subprogram P. Similarly, by combining Formulas 2 to 5 together with the guarantees just obtained, it can be checked easily that, if formal verification is applied to the entire program, then $AoRTE(P)$, *Pre*(P) and *Contract*(P) hold for every subprogram P. As we did not check termination here, this corresponds exactly to partial correctness of the program.

But, as we argued earlier, it is almost never the case that formal verification is applied to a complete program. In that very common case, it is not immediately clear what guarantees the application of formal verification gives. In particular, a user is probably interested in knowing that no run-time errors can be raised, which corresponds in our formalization to $AoRTE(P)$ and *Pre*$(P@)$ and that the subprogram contracts are respected, which corresponds in our formalization to *Effects*(P) and *Contract*(P). With our formulation of formal verification results as Horn clauses, we can precisely compute on which unverified assumptions these claims depend.

Termination. We have not discussed termination (represented by the tag *Term*) yet. In fact, termination is *not* an assumption of Formulas 2 and 3, because the properties claimed there are formulated in terms of partial correctness. For example, the most precise formalization of *Contract* is: if the precondition holds, and *if* the control flow of the program reaches the end of the subprogram, then the postcondition holds. Assuming absence of recursion, the SPARK tools can in fact establish termination by adding the following set of Horn clauses for each subprogram:

$$Term(R_i) \wedge Term(L_k) \longrightarrow Term(P)$$

[6] We are assuming absence of recursion in the program. Recursion requires a more advanced treatment.

assumption	verification strategy
assumption on call	
$Init(P@)$	coding standard, run-time initialization checking
$Nonaliasing(P@)$	static analysis, run-time non-aliasing checking, review
$Pre(P@)$	unit testing with assertions enabled
assumption on subprogram	
$Effects(P)$	static analysis, review, coding standard
$AoRTE(P)$	unit testing with run-time checks enabled
$Contract(P)$	unit testing with assertions enabled
$Term(P)$	unit testing, review

Table 2: Assumptions and possible verification strategies

where the R_i are the subprograms called and the L_k are the loops occurring in the subprogram P. Termination of loops can be established in SPARK by two means: `for`-loops terminate by construction in Ada, and more general loops can be annotated with a variant, wich allows to prove termination of the loop.

5.2 Discharging Assumptions

As visible from Formulas 1 to 3, the SPARK tools provide claims for the formal verification of one subprogram that discharge assumptions for the formal verification of another subprogram. It remains to see how to discharge assumptions at the boundary between formally verified and non-formally verified code.

Table 2 summarizes the assumptions of SPARK and presents possible verification strategies when the SPARK tools cannot be applied to the code on which the assumption is issued. The possibility in Ada to perform exhaustive run-time checking allows applying unit testing for verifying the absence of run-time errors. The possibility to also execute functional contracts is only available with SPARK 2014, not SPARK 2005, and it allows applying unit testing for verifying functional contracts.

Assumptions on the calling context are a bit more difficult to verify. Table 2 does not contain entries for assumptions on the calling context, so the first step is to find out all callers. Once all call points are identified, one needs to verify that each call verifies the assumptions that have been made for the verification of the called subprogram. This poses another interesting challenge: How to verify by testing that, *e.g.,* the precondition of a call deep inside the tested subprogram holds? How can one be sure that enough testing was applied? We are not answering these questions in this paper, but raising the issue.

Finally, SPARK lets the user insert assumptions inside the program for both flow analysis and proof. A typical example is a counter whose incrementation could overflow in theory, but never does in practice because it would require that the system runs for longer that its longest foreseen running time. In that case, the user can insert a suitable code assumption before the counter is incremented:

```
pragma Assume (Cnt < Integer'Last, "system is rebooted every day");
Cnt := Cnt + 1;
```

Such assumptions can also be part of the output of the tools, so that a review of all remaining assumptions can assess their validity.

5.3 A Concrete Example

In this section, we exercise the assumptions mechanism on the example provided in Section 2. Let us assume that we apply the SPARK tools only to `Set_Off_Button`, and in the remainder of this section we assume that flow analysis and proof have been applied successfully. We therefore obtain the following verification results:

$$\begin{aligned} \mathit{Effects}(R_i) \wedge \mathit{Init}(P) &\longrightarrow \mathit{Effects}(P) \wedge \mathit{Init}(R_i@) \\ \mathit{Effects}(R_i) \wedge \mathit{Init}(P) \wedge \mathit{AoRTE}(R_i) &\longrightarrow \mathit{AoRTE}(P) \end{aligned}$$

where P is `Set_Off_Button` and the R_i are the subprograms called by `Set_Off_Button`: `To_C_Bool`, `Background_Color_T`, `TB_Set_Off_Button`, and `Log_And_Handle_If_Error`.

Note that the above statement is somewhat shorter than the general one because the tags *Pre* and *Contract* do not apply (no preconditions or postconditions appear in the example), just as the tag *Nonaliasing*. In fact, it is impossible for aliasing to occur in the example, partly due to the types of parameters that cannot alias in some calls, and partly because scalar input parameters are passed by copy in Ada, and thus cannot alias with anything. The other claims that SPARK could provide do not apply, because `Set_Off_Button` doesn't have a postcondition, and the called subprograms do not have preconditions.

There are three assumptions in the above Horn clauses, for which we can find both which verification was actually performed in Project X, in Table 1, and to which general verification strategy this corresponds, in Table 2, as summarized in Table 3:

Assumption	How verified in Project X	Verification strategy applied
$\mathit{Effects}(R_i)$	MISRA	coding standard
$\mathit{Init}(P)$	MISRA	coding standard
$\mathit{AoRTE}(R_i)$	TEST	unit testing

Table 3: Discharging assumptions by other methods in a concrete example

In fact, most assumptions from Table 1 also appear in Table 2. Those that do not appear are project-specific. For example, the use of SPARK tools does not prevent the use of dynamic allocation in other parts of the program, in general. It happens to be a requirement of the project described in this paper.

6 Conclusion

We have presented the current state of the art in industrial software when applying formal verification on part of the code only. We reused the notion of explicit assumptions, which has already been present in other works, but used differently and for different purposes, to show how to render formal verification truly modular by proper tool support. We have experimented with a coarse-grain variant of explicit assumptions to realize the combination of proof and test, and have presented a more fine-grain model.

Future Work. Our immediate plan is to implement explicit assumptions in the SPARK technology, using the evidential tool bus as the back-end for assumptions management. More work is required to make the application of the framework truly usable. For example, all the presented tags simply have a subprogram name or call as argument. To increase precision, it would be better to also include tags with variable arguments, *e.g.,* a tag such as $Nonaliasing(x, y)$. Such support is not very different from what we describe here, but much more complex to write down, and the non-modular analysis to match call guarantees with calling context assumptions requires more work.

Our ultimate goal is to provide support for assumptions management and a smooth combination of test and proof in a future version of the commercial SPARK tools.

References

1. W. Ahrendt, G. J. Pace, and G. Schneider. A unified approach for static and runtime verification: framework and applications. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, pages 312–326. Springer, 2012.
2. M. I. S. R. Association. *MISRA C:2004 - Guidelines for the use of the C language in critical systems.* 2004.
3. J. Barnes. *SPARK: The Proven Approach to High Integrity Software.* Altran Praxis, 2012.
4. J. J. Chilenski. An Investigation of Three Forms of the Modified Condition/Decision Coverage (MCDC) Criterion. Technical Report DOT/FAA/AR-01/18, Apr. 2001.
5. M. Christakis, P. Müller, and V. Wüstholz. Collaborative verification and testing with explicit assumptions. In D. Giannakopoulou and D. Méry, editors, *FM 2012: Formal Methods*, volume 7436 of *Lecture Notes in Computer Science*, pages 132–146. Springer Berlin Heidelberg, 2012.
6. C. Comar, J. Kanig, and Y. Moy. Integrating formal program verification with testing. In *Proc. ERTS*, 2012.
7. L. Correnson and J. Signoles. Combining Analyses for C Program Verification. In M. Stoelinga and R. Pinger, editors, *Formal Methods for Industrial Case Studies (FMICS'12)*, volume 7437 of *Lecture Notes in Computer Science*, pages 108–130. Springer, Aug. 2012.

8. S. Cruanes, G. Hamon, S. Owre, and N. Shankar. Tool integration with the evidential tool bus. In *Verification, Model Checking, and Abstract Interpretation*, pages 275–294. Springer, 2013.
9. P. Cuoq, J. Signoles, P. Baudin, R. Bonichon, G. Canet, L. Correnson, B. Monate, V. Prevosto, and A. Puccetti. Experience report: OCaml for an industrial-strength static analysis framework. *SIGPLAN Not.*, 44(9):281–286, Aug. 2009.
10. C. Dross, P. Efstathopoulos, D. Lesens, D. Mentré, and Y. Moy. Rail, space, security: Three case studies for spark 2014. In *Proc. ERTS*, 2014.
11. I. O'Neill. SPARK – a language and tool-set for high-integrity software development. In J.-L. Boulanger, editor, *Industrial Use of Formal Methods: Formal Verification*. Wiley, 2012.
12. RTCA. DO-178C: Software considerations in airborne systems and equipment certification, 2011.
13. J. Tschannen, C. A. Furia, M. Nordio, and B. Meyer. Usable verification of object-oriented programs by combining static and dynamic techniques. In *Software Engineering and Formal Methods*, pages 382–398. Springer, 2011.