Static versus Dynamic Verification in Why3, Frama-C and SPARK 2014

Nikolai Kosmatov, Claude Marché, Yannick Moy, Julien Signoles

Static versus Dynamic Verification in Why3, Frama-C and SPARK 2014[⋆]

Nikolai Kosmatov[1], Claude Marché[2,3], Yannick Moy[4], and Julien Signoles[1]

[1] CEA, LIST, Software Reliability Laboratory, PC 174, F-91191 Gif-sur-Yvette
[2] Inria, Université Paris-Saclay, F-91893 Palaiseau
[3] LRI, CNRS & Univ. Paris-Sud, F-91405 Orsay
[4] AdaCore, F-75009 Paris

Abstract. Why3 is an environment for static verification, generic in the sense that it is used as an intermediate tool by different front-ends for the verification of Java, C or Ada programs. Yet, the choices made when designing the specification languages provided by those front-ends differ significantly, in particular with respect to the executability of specifications. We review these differences and the issues that result from these choices. We emphasize the specific feature of *ghost code* which turns out to be extremely useful for both static and dynamic verification. We also present techniques, combining static and dynamic features, that help users understand why static verification fails.

1 Introduction

Why3 (http://why3.lri.fr) is an environment for deductive program verification, providing a rich language for specification and programming, called WhyML. The specification part of WhyML serves as a common format for theorem proving problems, suitable for multiple provers. The Why3 tool generates proof obligations from purely logic lemmas and from programs annotated with specifications, then dispatches them to multiple provers, including SMT solvers Alt-Ergo, CVC4, Z3; TPTP first-order provers E, SPASS, Vampire; interactive theorem provers Coq, Isabelle and PVS.

Frama-C (http://frama-c.com) is an extensible platform for source-code analysis of C software. It features a plug-in architecture [42]: the Frama-C kernel performs syntactic analysis and typing of C code, and then allows the user to continue with different kinds of analyses, both static ones, e.g. based on theorem proving or abstract interpretation, or dynamic ones. The Frama-C kernel provides the formal specification language ACSL [3] for specifying contracts on C functions. Contracts can be written by users, or generated by plug-ins. Two plug-ins (Jessie and WP) permit deductive verification, that is, they can check that a given C function respects its ACSL specification, using theorem proving. Both plug-ins make use of Why3 as intermediate tool.

The SPARK language is a subset of Ada dedicated to real-time embedded software that requires a high level of safety, security, and reliability. It has been applied for many

⋆ Work partly supported by the Joint Laboratory ProofInUse (ANR-13-LAB3-0007, http://www.spark-2014.org/proofinuse) of the French national research organization

years in on-board aircraft systems, control systems, cryptographic systems, and rail systems [9]. Ada 2012 is the latest version of the Ada language [1], adding new features for specifying the behavior of programs, such as subprogram contracts and type invariants. SPARK 2014 (`http://www.spark-2014.org/`) is the last major version of SPARK, designed to interpret Ada 2012 contracts [39]. To formally prove a SPARK program correct, the SPARK 2014 toolset also uses WhyML as an intermediate language, and relies on Why3's interface to provers to discharge proof obligations.

Although deductive verification with both SPARK 2014 and Frama-C proceeds through Why3, the design of their specification languages differ significantly, and they are also different from Why3's own specification language. One of the reasons is that specification languages in Frama-C or SPARK aim at being used for other purposes than purely deductive verification, in particular they can be used for *dynamic* verification. *Run-time assertion checking* is the dynamic verification approach originating from the concept of design-by-contract, as it was implemented first in the Eiffel language [40] and later in the Java Modeling Language (JML) [35]. In those settings, annotations or contracts are clauses (pre- and postconditions, loop invariants, assertions) associated with boolean expressions. The run-time assertion checker inserts some extra code into the regular program code, that will throw an exception if any of these clauses is violated during execution. Statically checking the validity of contracts came a bit later in particular with the ESC-Java [5] tool, and later tools like Spec# [2], Dafny [37], OpenJML [14]. The issues that arise when trying to combine static and dynamic verification are quite well known [35] (mainly, how to make them agree on a common semantics), and studied, in particular regarding the impact on end-users of formal methods [7,8].

In Section 2, we review the different choices made in the design of the specification languages of Why3, Frama-C and SPARK 2014, and investigate the consequences and issues arising for the various kinds of analyses. One specific feature that is present in all of SPARK 2014, ACSL and WhyML is the notion of *ghost variables* and *ghost code*. Ghost code is a versatile way for the user to instrument code, and to exploit this instrumentation both for static and dynamic verification. Yet, ghost code features in the three languages above also differ significantly, and we investigate these differences in Section 3. In the activity of deductive verification, a major issue is to understand why a proof fails. Frama-C and SPARK 2014 implement different techniques to provide the user with hints about such a failure, using static or dynamic analysis in various ways. This aspect is reviewed in Section 4.

2 Design Choices in Specification Languages for Why3 and its Front-ends

Why3 is a versatile environment for deductive program verification. The WhyML language dedicated for specification and programming is mostly a purely functional programming language augmented with a notion of mutable variables [23]. Non-aliasing of mutable data is mandatory and is checked statically. Programs in WhyML are formally specified by contracts (mainly pre- and postconditions) written in an extended first-order logic partly detailed below. Verification proceeds by generating Verification Conditions (VCs) with a weakest precondition calculus. Why3 relies on external provers,

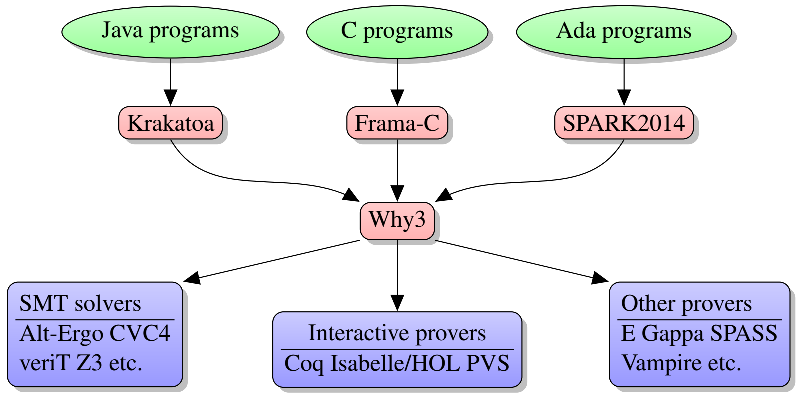

Fig. 1. Why3 front-ends and back-ends

both automated and interactive, in order to discharge these VCs. WhyML is used as an intermediate language for verification of SPARK programs as well as C and Java programs [22] (see Figure 1), and can also be used as a primary programming language (it can be compiled to OCaml).

2.1 Why3's Specification Language

Why3's core logic is a typed first-order logic with equality and built-in integer and real arithmetic. The user can enrich the logical context of a program's specification by designing extra theories defining new types, function symbols and predicates. New types can be defined e.g. by algebraic data type specification, while function and predicate symbols can be defined, possibly recursively, using pattern-matching on algebraic arguments. Types and logic symbols can also be declared axiomatically by giving symbol signatures and arbitrary axioms. Why3's core logic also provides extended features such as type polymorphism, inductive predicates and some form of higher-order functions [11]. Why3 comes which a pretty rich standard library of theories.

A logic of total functions. The choice of basing Why3's logic on standard first-order logic implies that it is a 2-valued logic with only total functions. When new function symbols are defined recursively, Why3's kernel statically checks that this recursion is well-founded, so as to be sure that a total function is defined. However, this raises issues when a function is defined axiomatically. Consider for example the typical case of division on real numbers, axiomatized by:

```
function div real real : real
axiom div_spec: forall x y:real. y ≠ 0 → y ∗ div x y = x
```

Notice first that if we omit the premise `y≠0` then the axiomatization would be inconsistent: `0 ∗ div 1 0 = 1` hence `0 = 1`. A first issue is thus that nothing prevents the user from writing inconsistent axioms. A mean to avoid such issues is to ask a prover to try to derive `false` from the specification: if it succeeds then for sure there is an inconsistency, but this cannot be a complete check because of undecidability of first-order logic. Such a *smoke detection* check can be done on demand in Why3 similarly to other tools like Dafny. The second issue is *underspecification*: division by zero is not specified, but it is some value because all functions are total. It is thus perfectly correct to state the proposition `div 1 0 = div 1 0` and indeed it is a tautology.

In the programming part of WhyML, it is not hard to check for division by zero: one can specify the division operator in programs as a declared, but not implemented, procedure as follows:

```
val division (x y:real) : real
  requires { y ≠ 0 }                (* precondition *)
  ensures { result = div x y }     (* postcondition *)
```

In other words, when a division is done in a WhyML program, a check is generated to ensure that divisor is not null, and if this check succeeds then it is sure that the result of the procedure is identical to the division specified in the logic.

This choice of a logic with total functions is very good for calling back-end provers such as SMT solvers or first-order provers, because they implement the very same choice. However, the issues discussed above are traps that a user more used to executable semantics of programs can fall into. These issues are extensively discussed by Chalin [7] based on experimental studies with practitioners.

Executable features of Why3. Why3's primary goal is static verification of contracts on WhyML procedures. Why3 also offers two features for executing programs: it implements a basic interpreter, and a compiler to OCaml. In both cases, the specifications are just discarded: there is no way to perform run-time assertion checking in Why3. As a consequence, the non-executability of logic specifications is not an issue. We discuss future work related to executability in Section 5.

2.2 The Krakatoa Specification Language

Krakatoa was historically the first front-end added to Why3 (actually to Why [22], the ancestor of Why3). It was designed in the context of the VerifiCard European project, aiming to statically check properties of JavaCard source code [27]. The initial goal was to interpret contracts, added to the Java source code and written in JML [5].

A major feature of JML is that an existing Java method can be called in the clauses of a contract, provided it has no side-effects. This feature allows the user to design specific Java code for specifying the rest of the code, and indeed JML comes with a library of side-effect free Java classes implementing general-purpose structures like sets. This design choice implies that the specifications are executable, which is natural since JML was initially designed for run-time assertion checking. The primary tool for static checking of JML annotations was ESC-Java [5] now superseded by OpenJML [14]. A consequence of the choice of using pure Java methods in specifications is that ESC-Java must automatically turn them into logic symbols, which is a highly non-trivial task. Indeed, if such a Java method has itself a precondition, when should it be checked? Should the code or the contract of this method be used for specification? What if the method

```
/*@ predicate Swap{L1,L2}(int a[],
  @                 integer i, integer j) =
  @  \at(a[i],L1) == \at(a[j],L2) &&
  @  \at(a[j],L1) == \at(a[i],L2) &&
  @  \forall integer k; k != i && k != j ==>
  @     \at(a[k],L1) == \at(a[k],L2);
  @*/

/*@ inductive Permut{L1,L2}(int a[],
  @                 integer l, integer h) {
  @ case Permut_refl{L}:
  @  \forall int a[], integer l h;
  @   Permut{L,L}(a, l, h) ;
  @ case Permut_sym{L1,L2}:
  @  \forall int a[], integer l h;
  @   Permut{L1,L2}(a, l, h) ==>
  @    Permut{L2,L1}(a, l, h) ;
  @ case Permut_trans{L1,L2,L3}:
  @  \forall int a[], integer l h;
  @   Permut{L1,L2}(a, l, h) &&
  @    Permut{L2,L3}(a, l, h) ==>
  @     Permut{L1,L3}(a, l, h);
  @ case Permut_swap{L1,L2}:
  @  \forall int a[], integer l h i j;
  @    l <= i < h && l <= j < h &&
  @     Swap{L1,L2}(a, i, j) ==>
  @      Permut{L1,L2}(a, l, h) ;
  @ }
  @*/
```

```
/*@ predicate Sorted{L}(int a[],
  @                 integer l, integer h) =
  @  \forall integer i j;
  @   l <= i <= j < h ==>
  @    \at(a[i] <= a[j],L) ;
  @*/

 /*@ requires t != null &&
   @  0 <= i < t.length &&
   @  0 <= j < t.length;
   @ assigns t[i],t[j];
   @ ensures Swap{Old,Here}(t,i,j);
   @*/
 void swap(int t[], int i, int j) {
  int tmp = t[i]; t[i] = t[j]; t[j] = tmp;
 }

 /*@ requires t != null;
   @ assigns  t[..];
   @ behavior sorted:
   @  ensures Sorted{Here}(t,0,t.length);
   @ behavior permutation:
   @  ensures
   @   Permut{Old,Here}(t,0,t.length);
   @*/
 void selection_sort(int t[]) {
   ...

 }
```

Fig. 2. Krakatoa annotation language: illustration of hybrid symbols

is not guaranteed to terminate? When should class invariants be checked? Leavens *et al.* [35] extensively discuss static versus dynamic verification for JML.

To avoid the issues of turning Java methods into logic symbols, it was decided in Krakatoa to forbid the use of Java method calls in specifications and provide access to Why's core logic instead [38]. This decision facilitates static verification, but removes the ability to execute specifications. It is to be noted however that when designing Krakatoa's specification language, we introduced the notion of so-called *hybrid* logic symbols: these are symbols whose definitions are not purely in the logic world but depend on the memory heap. Indeed they can even depend on several memory states, and this facility is made available to users using labels and JML's `\at` construct to refer to labels. This is exemplified by the annotated code for sorting an array of integers shown in Figure 2. The type `integer` denotes unbounded mathematical integers. The predicate `Permut` is defined inductively by four clauses introduced by the keyword `case`, and depends on two memory states: `Permut{L1,L2}(a,l,h)` means that the elements of array `a` in memory states `L1` and `L2` can differ between indices `l` and `h`, by a permutation of elements, and are the same elsewhere. In the postconditions of the contracts, `Old` is used for `L1` to refer to the pre-state of the method while `Here` is used for `L2` to denote the post-state. Although the specifications seem quite involved, such a code is statically checked automatically using SMT solvers.

2.3 ACSL: the ANSI C Specification Language

Historically, the first C front-end of Why came quickly after the Java front-end and was implemented in the Caduceus tool [21]. There was no largely adopted specification language for C like JML, so a home-made specification language was designed. It is mostly reusing the same design choice as Krakatoa's variation on JML: specifications may use pure function symbols and not the C functions themselves.

The design of the Frama-C framework [30] started in 2006, aiming at analysis of C source code using various techniques. An open plug-in architecture was designed so that a user can choose among different kinds of analyses. Originally, plug-ins were provided for deductive verification and static verification using abstract interpretation. The language ACSL [3] was designed for attaching formal contracts to C functions. This language is also a way for plug-ins to communicate information.

Since the main initial objective of Frama-C was static verification, the ACSL language was largely inspired by Caduceus and Krakatoa. In particular, the same choice of using a first-order logic with total functions was made, the use of unbounded integer arithmetic was encouraged, and the notion of hybrid predicates showed up too.

Later on, the number of plug-ins available in Frama-C increased a lot, aiming at many different kinds of analyses and not just static verification (`http://frama-c.com/plugins.html`). With the increasing use of Frama-C in industrial applications [30], the need for dynamic verification approaches showed up. In the following we detail the design of the E-ACSL variation of ACSL, aiming at run-time verification.

2.4 E-ACSL: Run-time Verification of ACSL Specifications

The E-ACSL plug-in [33] automatically translates a C program with ACSL annotations into another C program that reports a failure whenever an annotation is violated at run time. If no annotation is violated, the functional behavior of the new program is exactly the same as that of the original one. This plug-in thus provides a run-time assertion checker in the same vein as the one for JML. As such, it provides to the Frama-C environment the possibility to detect wrong annotations using concrete execution of the program, that is one way to "debug" specifications. Moreover, an executable specification makes it possible to check assertions that cannot be verified statically, and thus to establish a link between monitoring tools and static analysis tools [34]. An additional benefit of this plug-in is that it helps in combining some Frama-C analyzers with other ones that do not natively understand the ACSL specification language.

A major issue is that the initial design of ACSL did not take into account the possibility of executing specifications. This is why the E-ACSL plug-in only supports a subset of ACSL called E-ACSL [16,33]. The main features that are excluded are: unbounded quantifications, that is quantification on sets that cannot be statically seen as finite; and logic symbols that are axiomatized. Support of some ACSL clauses is not yet implemented (namely `assigns` clauses for frame properties and `decreases` clauses for termination properties). However, a significant effort was made to support the following important features.

Unbounded mathematical integers. These are compiled into C code, using GNU Multi-Precision library [25] if needed. Moreover, a careful static analysis is performed to avoid use of GMP's unbounded integers in many cases, for instance when the result of an arithmetical operation can still be represented by a machine integer (of the same, or a longer C type). It was noted that in practice only few uses of GMP integers are needed in the resulting code [16,28], meaning that supporting unbounded integers does not induce a significant overhead.

Support for memory-related ACSL constructs. ACSL provides built-in predicates that allow the user to express properties about the memory, for example that a pointer refers to a valid memory location [3]. This is supported in E-ACSL thanks to a custom C memory monitoring library, that tracks memory-related C constructs (`malloc` and `free` functions, initialization of variables, etc.) so that the generated instrumented code calls the monitoring library primitives to store validity and initialization information (whenever a memory location is allocated, deallocated and assigned), and to extract this information when evaluating memory-related ACSL constructs. To optimize the performance of the resulting code and avoid monitoring of irrelevant variables, a preliminary backward dataflow analysis has been implemented to determine a correct over-approximation of the set of memory locations that have to be monitored for a given annotated program [28,32].

Coping with underspecified logic functions. An annotation may contain underspecified functions (division by zero, but also access to an invalid pointer, etc.). A design choice is not to model this kind of undefined behavior, but to report an error instead. Technically, this is done by relying on the pre-existing Frama-C plug-in RTE dedicated to generate assertions from potential run-time errors. E-ACSL then translates the generated assertions as well. In practice, it means that an assertion such as

```
//@ assert (*p == *p);
```

although valid in any case in ACSL, will be reported as an error by E-ACSL when `p` is not valid, because RTE generates an assertion

```
//@ assert \valid(p);
```

This choice to have a different semantics in ACSL and E-ACSL with respect to underspecified functions follows the general observation by Chalin [7] that an end-user who writes ACSL annotations typically expects that an error is reported when dereferencing an invalid pointer in specifications.

2.5 SPARK 2014: Static Verification of Ada 2012 Contracts

Historically, the SPARK toolset, up to version 2005, was using its own specification language, for static verification only. The new version of Ada in 2012 added a notion of contracts in the Ada language itself, in a similar fashion as Eiffel contracts: they can be checked dynamically, as the compiler turns these contracts into executable code. Then the new version SPARK 2014 was redesigned, in order to use Ada2012 contracts as specification language, and a new static verification tool GNATprove was designed using Why3 as intermediate tool for VC generation.

The path followed by SPARK 2014 is thus similar to JML, and the reverse of the path from ACSL to E-ACSL: a language initially designed for run-time checking had to be used in static verification. A major objective was to guarantee that the semantics of the contracts must be the same for both run-time checking and static checking. The main issues to achieve this objective are as follows.

Capture undefinedness in assertions. Any expression in contracts that may generate an error at run time (e.g. division by zero) should induce the generation of a verification condition that proves it is defined. This means that the GNATprove tool must analyze each expression to collect all possible run-time errors, and generate additional assertions for them. It is somehow very similar to the RTE plug-in of Frama-C.

Promote program procedures into logic functions. This is the same issue ESC-Java had to solve in order to handle Java methods in specifications. The solution adopted by SPARK is to completely forbid side-effects in functions used in specifications. Such a check is quite easy to perform in the context of SPARK because there are strict coding rules for an Ada program to be in the SPARK fragment: pointers are forbidden, aliasing is forbidden, and a dataflow analysis is performed to collect read and write effects. Also, Ada 2012 has the notion of expression-functions, whose translation into logic is immediate. In fact, the work on SPARK 2014 was influential in getting expression-functions into Ada 2012, so that they can be used in static verification.

Providing access to non-executable datatypes. It turns out that for complex specifications it is important to provide extra datatypes. Datatypes that are often needed are collections. In SPARK, there is a library of collections that are specifically designed for simultaneous use in dynamic and static verification [17]. The user can even design her own library of non-executable datatypes, using the so-called *external axiomatizations*, for example to support unbounded integers in proof. Partial support for unbounded integers is also available by selecting a compilation switch, which ensures that intermediate computations are performed in arbitrary precision: in SPARK, there is a library for unbounded arithmetic that is used for this purpose. When the switch is selected, contracts with arithmetic computations can be both dynamically checked with this library and also interpreted as mathematical integers in static verification.

Type invariants. In Ada, dynamic verification of type invariants is partial, for efficiency reason. It is only done at exit of public procedures of a package, and only for types that are defined in the same package. the JML run-time assertion checker has similar restrictions when checking class invariants. In the SPARK subset of Ada, appropriate restrictions on the expressions used in invariants were chosen so that verification of invariants can be done statically. It is important to notice that non-aliasing restrictions of SPARK are crucial to be able to check invariants in a sound way. Why3 has a similar notion of type invariants, and their sound static verification is also possible thanks to non-aliasing restrictions. On the contrary, there is no Frama-C plug-in today that can statically check ACSL's type invariants because of the potential aliasing in C data structures.

2.6 Mixed Static-Dynamic Verification in Frama-C and SPARK 2014

As seen above, both Frama-C and SPARK 2014 have different techniques and tools to check specifications either statically or dynamically. A natural question that arises is whether it is safe to mix these various kinds of verification techniques on the same program. Both Frama-C and SPARK 2014 have tool support to ensure consistency of verification activities.

The Frama-C kernel is the central core that communicates with all the plug-ins. When a given plug-in is able to verify that some annotation is valid, it is usually under the assumption that some other annotations are valid. For example, when a static verification plug-in can prove that a postcondition for a procedure is valid, it is under the hypothesis that the pre-condition holds. To ensure consistency, the Frama-C kernel attaches to each annotation some information status: it tells which plug-in validates it, together with the set of other annotations that are assumed by this plug-in [15]. The graph of dependencies between annotations that are assumed or proved can be displayed graphically, and it is checked automatically whether every specification has been proved by at least one plug-in.

Combining static and dynamic verification in SPARK is possible and indeed expected, including when the program also contains non-SPARK Ada code. SPARK reconciles the logic semantics and executable semantics of contracts, so users can execute contracts, debug them like code, and test them when formal verification is too difficult to achieve. Furthermore, by keeping the annotation language the same as the programming language, users don't have to learn another language. Like Ada has been designed to integrate smoothly with parts of the application written in C, SPARK has been designed to integrate smoothly with parts of the application written in Ada outside of the SPARK subset. Hence, a SPARK application may consist of functions in SPARK, Ada and C being linked together. While formal verification can be applied to the SPARK part of the application, this is not the case for the Ada part or (unless the user also uses Frama-C) the C part. Those parts should be verified using traditional verification techniques based on testing and reviews. The overall verification argument may be composed from individual verification arguments on the SPARK subprograms (using formal verification) and Ada or C subprograms (using other techniques), based on the subprogram contracts used in formal verification. Indeed, the assumptions made during formal verification of a subprogram can be verified during testing of another function called by or calling the first one: preconditions and postconditions can be executed with the very same semantics that they have in proofs. SPARK 2014 offers a similar mechanism as Frama-C [29] to check what is proved, by which technique, under which assumptions.

3 Ghost Variables and Ghost Code

A *ghost variable* is a variable that is added to a given program only for the purpose of formal specification. This notion is reminiscent from the notion of auxiliary variables in

```
/*@ requires x >= 0 && y >= 0;
  @ ensures \exists integer a,b; a*x+b*y == \result;
  @*/
int gcd(int x, int y) {
  //@ ghost integer a = 1, b = 0, c = 0, d = 1;
  /*@ loop invariant x >= 0 && y >= 0 ;
    @ loop invariant a*\at(x,Pre)+b*\at(y,Pre) == x ;
    @ loop invariant c*\at(x,Pre)+d*\at(y,Pre) == y ;
    @ loop variant y;
    @*/
  while (y > 0) {
    int r = x % y;
    //@ ghost integer q = x / y;
    x = y; y = r;
    //@ ghost integer ta = a, tb = b;
    //@ ghost a = c, b = d, c = ta - c * q, d = tb - d * q;
  }
  return x;
}
```

Fig. 3. Ghost code for computing Bézout coefficients

Hoare logic. These variables typically need to be assigned, during the normal execution of the program: this is done by adding *ghost code*.

As an illustrative example, consider Euclide's algorithm to compute the greatest common divisor d of two integers x and y. One may want to state as a postcondition the Bézout property: there exist integers a and b such that $d = ax + by$. A postcondition with an existential quantification is typically hard to prove by automatic provers. Moreover, in this particular example, the property itself is a non-trivial mathematical one. It is a typical example where ghost code can help: in that example, the values of a and b can be computed during execution of the algorithm itself. A program in C annotated in ACSL is shown in Figure 3. The ghost variables a, b, c and d store coefficients, modified in the ghost code of the loop body, so that they keep satisfying Bézout-like properties as described by the loop invariants.

Another example of ghost code is an alternative way to specify the permutation property of a sorting algorithm: instead of an inductive predicate as in Figure 2, a sorting algorithm may return a ghost array, mapping the interval of indexes $[0..n-1]$ to itself in a bijective way, expressing the permutation of elements before and after sorting. The content of this ghost array can be updated with ghost code during the sorting algorithm, so that it keeps representing the permutation of elements from the initial array to its current state.

3.1 Ghost Code in Why3

The ability to set a ghost attribute to variable declarations and to arbitrary code is natively part of the WhyML language. The Why3 type system ensures that ghost code must not interfere with regular code, in the sense that it can be erased without observable difference in the program outcome. In particular, ghost data is forbidden to participate in regular computations and ghost code can neither mutate regular data nor diverge [20]. There are numerous and various examples of code that naturally need ghost code for their formal specification and their proof (`http://toccata.lri.fr/gallery/ghost.en.html`).

Lemma functions. Beyond instrumenting the regular code, ghost code is an effective way to guide the automatic provers in static verification. Ghost code can be used to prove properties: if one writes a ghost function with a contract of the form

```
let f(x₁ : τ₁,...,xₙ : τₙ) : τ
  requires Pre
  variant var
  ensures Post
```

and if this function has no side-effect and is proved terminating (with the decreasing measure *var* given by the `variant` clause), then it is a constructive proof of

$$\forall x_1,\ldots,x_n, \exists result, Pre \Rightarrow Post$$

In particular, if f is defined recursively, it simulates a proof by induction: the VC generator effectively generates the cases of an induction scheme. This technique of using programs to make proofs is nowadays called "auto-active verification" and is available in several other verification environments [36,44].

Lemma functions are often used in complex programs proved in Why3, for example to deal with recursive data structures [11] or to reason on semantics [12]. The most complex case study of static verification using Why3 up to now, a verified first-order prover [13], makes extensive use of ghost code and lemma functions.

3.2 Static and Dynamic Verification of Ghost Code

Ghost code is not only useful for static verification. It may be executed under some conditions, and thus is equally helpful for run-time verification: it can monitor properties dynamically. Environments like JML and Spec# have ghost variables and ghost code, primarily for run-time execution. Dafny also has a notion of lemma functions.

Ghost code in Frama-C. In Frama-C, ghost code is just regular C code located in ACSL comments. As such, it is naturally possible to use it for static verification [6], and to execute it with E-ACSL. However, Frama-C has currently some limitations with respect to ghost code: unlike what is specified in the ACSL design [3], the current implementation of Frama-C only allows ghost variables to have a C type, not a logic type like unbounded integers. This, of course, simplifies execution of ghost code, but limits the ghost capabilities for static verification. As such, the example of Figure 3 is not accepted because the ghost variables are declared as `integer` (unbounded mathematical integers of ACSL), so to statically check this code one currently needs to turn them into `int` and ignore overflow checks. Another current limitation is that the kernel does not check that ghost code does not interfere with regular code like in Why3. Statically checking this property is much more difficult in C than in Why3, because C allows arbitrary aliasing whereas Why3 controls aliasing statically [20].

Ghost code in SPARK 2014. In SPARK, ghost code is declared using an Ada aspect `Ghost` on the declaration. In the design of ghost code in SPARK, it was mandatory to be able to check non-interference of ghost code with regular code. In particular the compiler must be able to eliminate ghost code if the user wants to compile a program without ghost. Unlike the case of Frama-C, SPARK 2014 can statically check, like Why3, the non-interference of ghost code. This is because SPARK code must follow strong non-aliasing properties (checked by data-flow analysis) and coding rules (`http://docs.adacore.com/spark2014-docs/html/lrm/subprograms.html#ghost-entities`).

As in other systems, ghost variables in SPARK are typically used for keeping intermediate values, keeping memory of previous states, or logging previous events (`http://docs.adacore.com/spark2014-docs/html/ug/spark_2014.html#ghost-code`). Various uses of ghost are presented in examples of the SPARK manual (`http://www.spark-2014.org/entries/detail/manual-proof-in-spark-2014`): ghost code can be used to encode a state machine (functional properties of the Tetris game `http://blog.adacore.com/tetris-in-spark-on-arm-cortex-m4`) or to model a file system (proving standard Ada `Get_Line` function `http://blog.adacore.com/formal-verification-of-legacy-code`). Ghost code was extensively used in high-level specifications of memory allocators [18].

A limitation with respect to ghost code, similar to Frama-C, is the executability of ghost code in case of use of external axiomatizations: in that case the compiler would refuse to compile ghost code into run-time checks.

4 Understanding Proof Failures

In static verification, a major issue is understanding the reason why some proof fails. There are various reasons why it may fail:

1. The property to prove is indeed invalid: the code is not correct with respect to the given specification.
2. The property is in fact valid, but is not proved, for two possible reasons:
   a. The prover is not able to obtain a proof (in the given time and memory limits): this is the incompleteness of the proof search;
   b. The proof may need extra (or stronger) intermediate annotations, such as loop invariants, or more complete contracts of the subprograms.

For the user to be able to fix the code or the specification of her program, it is essential to understand into which of the above cases any undischarged VC falls. A general solution is to generate *counterexamples* in order to illustrate the issue on concrete values. This capability exists in different forms in Why3, SPARK 2014 and Frama-C.

4.1 Counterexamples from SMT models

A first solution is to exploit the SMT solvers' capability of generating *models*. Indeed, to discharge a given VC, the SMT solver is given the hypotheses and the negation of the goal, and it is asked to prove unsatisfiability. In case of failure, the SMT solver provides a model that can be turned into a counterexample for the initial program. This is how it is implemented in Why3 and SPARK 2014 [26]. In Frama-C, the Counter-Example plug-in implements the very same idea [30], but it is still a not-yet-released research prototype which only supports a few constructs.

There are actually some issues with this approach, which limit its applicability. A first issue is related to the *incompleteness* of the solver: in presence of non-linear integer arithmetic, or arbitrary quantification, the logic is not decidable so the solver may time out. Second, when a model is generated, it can only lead to a *potential* counterexample, and the user still has to understand what should be fixed if it is not a true one. That is due to the solver's vision of the program, in which the code of a called function and the body of a loop are replaced by the corresponding *subcontracts*: the contract of the callee and the loop invariant, respectively. Thus, such a counterexample can illustrate either Reason 1 above (non-compliance between the code and the specification) or Reason 2b above (the code is in fact compliant to the specification, but the contracts of some callees or loops are too weak to complete the proof). Run-time checking of the program for such a counterexample candidate can be used to distinguish these cases.

4.2 Counterexamples from testing

In Frama-C, the StaDy plug-in has been designed to generate counterexamples [41]. Unlike above, the technique does not rely on a counter-model generated by the prover, it is based on test generation instead. The annotations of the input program are first transformed into C code similarly to the E-ACSL plug-in. The instrumented code is then passed to a Dynamic Symbolic Execution (DSE) testing tool that tries to find tests producing annotation failures. An interesting aspect is that with this approach it is possible (using two different instrumentation techniques) to distinguish between a non-compliance (Reason 1 above) and a subcontract weakness (Reason 2b above). Other potential benefits come from the capacity of DSE to focus on one path at a time, and to use concrete values when the constraints are too complex for a solver. The main limitation of this approach is related to the combinatorial explosion of the path space to be explored by the test generation tool. Other related approaches have been reported in the context of Eiffel [43] and Dafny [10].

All these techniques being relatively recent, more research is required to better evaluate and understand their benefits and limitations in practice.

5 Conclusions and Future Work

We have surveyed, in the context of Why3, Frama-C and SPARK 2014, various cases where dynamic verification supplement static verification. The first one is related to verification by testing those parts of the program that are too complex to prove formally,

	Why3	Krakatoa	Frama-C		SPARK
			ACSL	E-ACSL	
Executable contracts	no	no	no	yes	yes
Only total functions in logic	yes	yes	yes	no[1]	no[2]
Unbounded integers in logic	yes	yes	yes	yes	no[3]
Unbounded quantification	yes	yes	yes	no	no
Ghost code	yes	partial[4]	partial[5]	partial[5]	yes
Counterexamples from solvers	yes	no	partial[6]	partial[6]	yes
Counterexamples from testing	no	no	no	yes[7]	no

1 Run-time checks for well-definedness are generated.
2 Run-time checks and VCs for well-definedness are generated.
3 See discussion in Section 2.5.
4 Non-interference with regular code is not checked.
5 Only executable C code, and non-interference with regular code is not checked.
6 The dedicated plug-in Counter-Example is not yet publicly available.
7 The test generation tool PathCrawler, underlying StaDy, is currently not publicly available.

Fig. 4. Comparison of features supported by specification languages.

and a safe combination of tests and proofs. We have also discussed the use of ghost code, essential for formally specifying complex functional behaviors and exploitable by both static and dynamic approaches. The third case — understanding the reason why a proof fails — can rely again either on a static method (exploiting the counter-model returned by an SMT solver) or a dynamic one (applying test generation on a code instrumented with executable annotations). Figure 4 summarizes the various aspects supported or not by the considered tools.

We emphasized the role of non-aliasing restrictions in Why3 and SPARK 2014, which permits to check type invariants in a sound way, and also to statically check the non-interference of ghost code with regular code. We conclude with a few issues that are worth investigating further.

Need for executability of pure logic types. We have seen that using unbounded mathematical integers in specifications is natural in static verification, and can be supported in dynamic verification thanks to the use of libraries implementing unbounded integers. There are many other logic theories used in static verification (as present e.g. in Why3's standard library) and each of them should come with an executable counterpart to be able to use it dynamically. It should be done in a systematic way, that is, by synthesizing executable code from axiomatization [31]. A particular hard case is that of real numbers: it is a theory that is quite well supported in automatic provers, but there is no obvious solution how to provide an executable version of real numbers. Some authors propose approximation methods for that purpose [19,24].

Need for unbounded quantification. To be executable, quantification in formulas must necessarily range over finite sets. However, there are examples where specification requires quantification over infinitely many data, for instance, the solution of "patience game" from the VScomp competition in 2014 (http://toccata.lri.fr/gallery/patience.en.html) needs quantification over infinitely many sequences. JML and SPARK languages syntactically impose finite ranges of quantification so that the specification of this example can simply not be written. On that specific matter, there is still a gap between static and dynamic verification that needs to be filled.

References

1. Barnes, J.: Programming in Ada 2012. Cambridge University Press (2014)
2. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview. In: CASSIS. LNCS, vol. 3362, pp. 49–69. Springer (2004)
3. Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, version 1.10 (2013), http://frama-c.cea.fr/acsl.html
4. Bulwahn, L.: The new quickcheck for isabelle. In: CPP. pp. 92–108. Springer (2012)
5. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer 7(3), 212–232 (2005)
6. Burghardt, J., Gerlach, J., Lapawczyk, T., Carben, A., Gu, L., Hartig, K., Pohl, H., Soto, J., Völlinger, K.: ACSL by example, towards a verified C standard library. version 11.11 for Frama-C Sodium. Tech. rep., Fraunhofer FOKUS (2015), http://publica.fraunhofer.de/dokumente/N-364387.html
7. Chalin, P.: Logical foundations of program assertions: What do practitioners want? In: SEFM. pp. 383–393. IEEE Computer Society (2005)
8. Chalin, P.: Reassessing JML's logical foundation. In: Proceedings of the 7th Workshop on Formal Techniques for Java-like Programs (FTfJP'05). Glasgow, Scotland (2005)
9. Chapman, R., Schanda, F.: Are we there yet? 20 years of industrial theorem proving with SPARK. In: ITP. LNCS, vol. 8558, pp. 17–26. Springer (2014)
10. Christakis, M., Leino, K.R.M., Müller, P., Wüstholz, V.: Integrated environment for diagnosing verification errors. In: TACAS. Springer (2016)
11. Clochard, M.: Automatically verified implementation of data structures based on AVL trees. In: VSTTE. LNCS, vol. 8471, pp. 167–180. Springer (2014)
12. Clochard, M., Filliâtre, J.C., Marché, C., Paskevich, A.: Formalizing semantics with an automatic program verifier. In: VSTTE. LNCS, vol. 8471, pp. 37–51. Springer (2014)
13. Clochard, M., Marché, C., Paskevich, A.: Verified programs with binders. In: Programming Languages meets Program Verification (PLPV). ACM Press (2014)
14. Cok, D.R.: OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse. In: F-IDE. EPTCS, vol. 149, pp. 79–92 (2014)
15. Correnson, L., Signoles, J.: Combining Analyses for C Program Verification. In: FMICS. LNCS, vol. 7437, pp. 108–130. Springer (2012)
16. Delahaye, M., Kosmatov, N., Signoles, J.: Common specification language for static and dynamic analysis of C programs. In: SAC. pp. 1230–1235. ACM (2013)
17. Dross, C., Filliâtre, J.C., Moy, Y.: Correct Code Containing Containers. In: TAP. LNCS, vol. 6706, pp. 102–118. Springer (2011)
18. Dross, C., Moy, Y.: Abstract software specifications and automatic proof of refinement. In: RSSR (2016), http://www.spark-2014.org/entries/detail/spark-prez-at-new-conference-on-railway-systems
19. Dufour, J.L.: Formal Methods Applied to Complex Systems, chap. B Extended to Floating-Point Numbers: Is it Sufficient for Proving Avionics Software? John Wiley & Sons, Inc. (2014)

20. Filliâtre, J.C., Gondelman, L., Paskevich, A.: The spirit of ghost code. Formal Methods in System Design (2016), to appear.
21. Filliâtre, J.C., Marché, C.: Multi-Prover Verification of C Programs. In: ICFEM. pp. 15–29. Springer (2004)
22. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: CAV. LNCS, vol. 4590, pp. 173–177. Springer (2007)
23. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: ESOP. LNCS, vol. 7792, pp. 125–128. Springer (2013)
24. Gao, S., Avigad, J., Clarke, E.M.: Delta-complete decision procedures for satisfiability over the reals. CoRR abs/1204.3513 (2012), http://arxiv.org/abs/1204.3513
25. GMP: Gnu multiple precision arithmetic library. https://gmplib.org/
26. Hauzar, D., Marché, C., Moy, Y.: Counterexamples from proof failures in SPARK. In: SEFM. LNCS, vol. 9763. Springer (2016)
27. Jacobs, B., Marché, C., Rauch, N.: Formal verification of a commercial smart card applet with multiple tools. In: AMAST. LNCS, vol. 3116. Springer (2004)
28. Jakobsson, A., Kosmatov, N., Signoles, J.: Rester statique pour devenir plus rapide, plus précis et plus mince. In: JFLA (2015)
29. Kanig, J., Chapman, R., Comar, C., Guitton, J., Moy, Y., Rees, E.: Explicit assumptions - a prenup for marrying static and dynamic program verification. In: TAP. LNCS, vol. 8570, pp. 142–157. Springer (2014)
30. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Aspects of Computing pp. 1–37 (2015)
31. Kneuss, E., Kuraj, I., Kuncak, V., Suter, P.: Synthesis modulo recursive functions. In: OOPSLA. pp. 407–426. ACM (2013)
32. Kosmatov, N., Petiot, G., Signoles, J.: An optimized memory monitoring for runtime assertion checking of C programs. In: RV. LNCS, vol. 8174, pp. 167–182. Springer (2013)
33. Kosmatov, N., Signoles, J.: A lesson on runtime assertion checking with Frama-C. In: RV. LNCS, vol. 8174, pp. 386–399. Springer (2013)
34. Kosmatov, N., Signoles, J.: Runtime assertion checking and its combinations with static and dynamic analyses - tutorial synopsis. In: TAP. pp. 165–168 (2014)
35. Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C., Cok, D.R.: How the design of JML accomodates both runtime assertion checking and formal verification. Tech. Report 03-04, Iowa State University (2003)
36. Leino, K.R.M.: Automating induction with an smt solver. In: VMCAI. pp. 315–331. Springer (2012)
37. Leino, K.R.M., Wüstholz, V.: The Dafny integrated development environment. In: F-IDE. Electronic Proceedings in Theoretical Computer Science, vol. 149, pp. 3–15 (2014)
38. Marché, C., Paulin-Mohring, C., Urbain, X.: The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. Journal of Logic and Algebraic Programming 58(1–2), 89–106 (2004)
39. McCormick, J.W., Chapin, P.C.: Building High Integrity Applications with SPARK. Cambridge University Press (2015)
40. Meyer, B.: Object-Oriented Software Construction. Prentice-Hall, Inc., 1st edn. (1988)
41. Petiot, G., Kosmatov, N., Botella, B., Giorgetti, A., Julliand, J.: Your proof fails? testing helps to find the reason. In: TAP. LNCS, vol. 9762. Springer (2016)
42. Signoles, J.: Software Architecture of Code Analysis Frameworks Matters: The Frama-C Example. In: F-IDE. pp. 86–96 (2015)
43. Tschannen, J., Furia, C., Nordio, M., Meyer, B.: Program checking with less hassle. In: VSTTE 2013, Revised Selected Papers. pp. 149–169. Springer (2014)
44. Tschannen, J., Furia, C.A., Nordio, M., Polikarpova, N.: Autoproof: Auto-active functional verification of object-oriented programs. In: TACAS. pp. 566–580. Springer (2015)