Usable Verification of Object-Oriented Programs by Combining Static and Dynamic Techniques

Julian Tschannen, Carlo A. Furia, Martin Nordio, and Bertrand Meyer

Chair of Software Engineering, ETH Zurich, Switzerland
firstname.lastname@inf.ethz.ch

Abstract. With formal techniques becoming more and more powerful, the next big challenge is making software verification practical and usable. The Eve verification environment contributes to this goal by seamlessly integrating a static prover and an automatic testing tool into a development environment. The paper discusses the general principles behind the integration of heterogeneous verification tools; the peculiar challenges involved in combining static proofs and dynamic testing techniques; and how the combination, implemented in Eve through a blackboard architecture, can improve the user experience with little overhead over usual development practices. Eve is freely available for download.

1 Verification as a Matter of Course

Even long-standing skeptics must acknowledge the substantial progress of formal methods in the last decades. Established verification techniques, such as those based on axiomatic semantics or abstract interpretation, have matured from the status of merely interesting scientific ideas to being applicable in practice to realistic programs and systems. Novel approaches have extended their domain of applicability beyond their original scope, providing new angles from which to attack the hardest verification challenges; for example, model checking techniques, initially confined to digital hardware verification, are now applied to software or real-time systems. Other techniques, such as testing, have long been part of the standard development process, but only recently have they become first-class citizens of the verification realm, evolving in the case of random-based testing into rigorous, formal, and automatable approaches. Verification requires accurate specifications, and progress in this area has been no less conspicuous, with the development of understandable notations, such as those based on Design by Contract, which integrate seamlessly with the programming language and are amenable to static as well as dynamic analysis techniques. Finally, tool support has tremendously improved in terms of both reliability and performance, as a result of cutting-edge engineering of every component in the verification tool-chain as well as the increased availability of computing power.

With the consolidation of these outstanding achievements [14], the new frontier is to make verification really usable by practitioners [27]: the quest for high reliability to become a standard part of the software development process—"verification as a matter of course". The present paper is a step towards this ambitious goal with two contributions, one general and one specific.

The general contribution is a development environment that seamlessly integrates formal verification with the standard tools offered by programming environments for object-oriented development (editor, compiler, debugger, ...). The integrated environment is called Eve, built on top of EiffelStudio—the main IDE for Eiffel developers. Section 6 describes the engineering of Eve, showing how it takes into account several of the heterogeneous concerns originating from the goal of improving the usability of formal verification, such as user interaction and management of computational resources.

The implementation of Eve, freely available for download [11], continues to evolve as a result of ongoing efforts to integrate more verification techniques and new verification tools. The currently available implementation, illustrated through an example session in Section 2, focuses on the integration of two well-known techniques: static verification based on Hoare-style proofs, currently implemented in Eve through the AutoProof tool [20], and dynamic analysis based on random testing, implemented through AutoTest [19]. Section 4 describes these tools. After a review of the state of the art in Section 3, Sections 5 and 7 illustrate the specific contribution of the paper by discussing the challenges of integrating two very different verification techniques, tests and proofs, and how Eve combines them to improve each one's effectiveness and usability. Section 9 concludes with an analysis of limitations and our current work to overcome them.

2 An Example Session with Eve

Consider the perspective of a user—henceforth called Adam—who is using Eve to develop a collection of data structure implementations. Table 1 shows portions of Adam's code; the code shown is simplified for presentation purposes, but it reflects real features found in versions of EiffelBase, a standard library used in most Eiffel programs.

The ancestor class *COLLECTION* models generic containers with a well-defined interface including, in addition to other features not shown, routines (methods) *extend* that adds its argument to the collection and *is_equal* which tests for object equality. *extend* is annotated with a precondition (**require**) and postcondition (**ensure**) which refer to other features of the class (such as *has*) not shown. *extend* is *deferred* (abstract) as it lacks an implementation; *is_equal*'s body, instead, calls a pre-compiled implementation written in C through the **external** keyword. This encapsulation mechanism prevents correctness proofs of the routine's implementation (whose source is not accessible); in addition, *COLLECTION* cannot be instantiated and tested because it includes deferred routines. This seems an unfortunate situation for verification, but verification with Eve becomes effective in the two descendants of *COLLECTION* shown in Table 1: *ARRAY* and *ARRAYED_LIST*.

Class *ARRAY* redefines the attribute *extendible* to **False** because an array is a container of statically-defined size and cannot accommodate new elements *ad lib*. Correspondingly, the precondition of the inherited feature *extend* becomes unsatisfiable in *ARRAY*. This way of "deactivating" a routine is inconvenient for automatic testing tools such as AutoTest, which tries, in a vain effort, to generate instances of *ARRAY* where the precondition of *extend* holds in order to test it. AutoProof, the static proof component of

```
1  deferred class COLLECTION [G]
2
3    ...
4
5    extendible: BOOLEAN
6
7    extend (v: G)
8      -- Ensure that structure contains 'v'.
9    require
10     extendible
11     v ≠Void
12   deferred
13   ensure has (v) end
14
15   is_equal (other: COLLECTION [G]): BOOLEAN
16     -- Is 'other' attached to an object equal to 'Current'?
17   require other ≠Void
18   external built_in
19   ensure Result = other.is_equal (Current) end
20
21 end -- CONTAINER
22
23
24 class ARRAY [G]
25   inherit COLLECTION [G]
26           redefine extendible end
27
28   extendible: BOOLEAN = False
29
30   ...
31 end -- ARRAY
```

```
32 class ARRAYED_LIST [G]
33   inherit ARRAY [G]
34           redefine extendible end
35
36   extendible: BOOLEAN = True
37
38   extend (v: G) do ... end
39
40   make_default (n: INTEGER)
41     -- Allocate list with 'n' slots items
42     -- and fill it with default values.
43   require n ≥ 0
44   local l_v: G
45   do
46     Precursor (n)
47     across [1..n] as i loop extend (l_v) end
48   end
49
50   remove_left_cursor (c: CURSOR)
51     -- Remove item to left of 'c' position.
52   require
53     not is_empty
54     c ≠Void and valid (c)
55     not c.before and not c.is_first
56   do
57     remove (c.index − 1)
58   ensure
59     count = old count − 1
60     c.index = old c.index − 1
61   end
62 end -- ARRAYED_LIST
```

Table 1. Classes *CONTAINER*, *ARRAY*, and *ARRAYED_LIST*.

Eve, comes to the rescue in this case: it easily figures out that the precondition of *extend* is unsatisfiable in *ARRAY* (line 10 in Table 1), and hence that *extend* is trivially correct and requires no further analysis. Adam checks that *ARRAY.extend* receives a green light and requires no further attention (Figure 1).

Class *ARRAYED_LIST* switches *extendible* to **True** and provides a working implementation of *extend* available to clients. When Eve tries to test the class, it quickly discovers a fault in the creation procedure (constructor) *make_default*: after the instruction **Precursor** (*n*) calls the creation procedure in the ancestor of *ARRAY*, the loop (**across**...**loop**...**end**) tries to call *extend* with the local *l_v* as argument; this violates *extend*'s precondition clause *v* ≠**Void** because *l_v* is not initialized and hence equals the default value **Void** (*null* in Java or C). Adam sees there is something wrong in Eve's report (Figure 1); he expands the description of the error and understands how to fix the bug by adding an instruction **create** *l_v* before the loop on line 47.

While Adam is busy fixing the error, testing cannot proceed on the same class. Even if the creation procedure were correct, routine *remove_left_cursor* would remain arduous for automated testing techniques because its precondition is relatively complex; a random-based approach to the generation of test cases requires specialized techniques and a long running time to select objects satisfying the clauses in lines 53–55 [29]. Eve circumvents these limitations by running a static proof, which analyzes individual

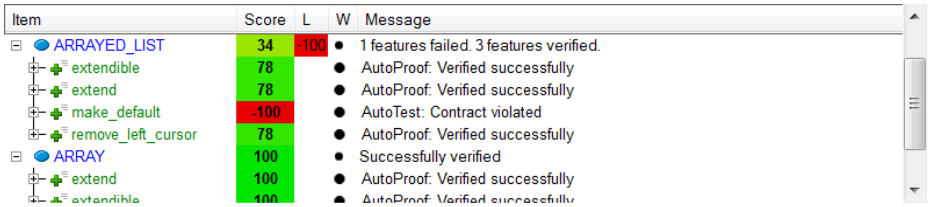

Fig. 1. Example report of Eve, showing scores of classes and routines. The third column displays the lowest negative score among the routines of each class.

routines and does not need a correct creation procedure. The proof succeeds in establishing that the invocation of *remove* (line 57) is correct and ensures the postcondition of *remove_left_cursor*: the routine is correct and no testing is needed (Figure 1).

Later, as soon as the constructor of *ARRAYED_LIST* is fixed, Eve continues its work and exhaustively tests the implementation of *is_equal* finding no postcondition violations. This is not as good as a correctness proof, but it comforts Adam's confidence in the reliability of *is_equal*, and it is the best result possible for a routine whose implementation can be analyzed only as black-box.

Although it only uses some of Eve's features, this scenario illustrates how Eve can help develop correct applications with little overhead over standard practices:

- Eve is completely automatic and integrated in a full-fledged IDE.
- It supports verification of functional correctness specifications embedded as contracts (pre and postconditions, class invariants, intermediate assertions).
- It transparently manages different verification engines to complement their strengths, supports the full programming language Eiffel, and provides fast feedback to users.
- It only displays such feedback when needed, to encourage focus on the most egregious errors, and to increase the users' confidence in the correctness of an implementation based on the available evidence.

3 Related Work

The following sections explains the Eve machinery that makes usage scenarios such as the above possible. To set these solutions in context, we first examine briefly a few state-of-the-art tools for static and dynamic verification (proofs and tests), with a summary of their distinctive features and a summary of the relatively few approaches that combine both techniques. A broader review of formal techniques is available elsewhere [14,27].

Static verification. Projects such as ESC/Java [12] and Spec# [2] have made Hoare-style correctness proofs more practical and automatic, at least for simple programs. The Spec# language extends C# with preconditions, postconditions, object invariants, and non-null types; the Spec# environment verifies Spec# programs by translating them into Boogie, also developed within the Spec# project. The success of this approach has shown the importance of using an intermediate language for verification. Spec# works

on interesting examples; however, it is still not applicable to every feature of C# (for example, exceptions and function objects). A design choice that distinguishes Spec# from AutoProof for Eiffel is the approach to deal with some delicate issues, namely the framing problem and managing class invariants. Spec# introduces specialized annotations, which make proofs easier but at the price of a significant additional annotational burden for developers. AutoProof, on the contrary, does not introduce *ad hoc* annotations and correspondingly may fail to verify programs where Spec# is successful. Some of these limitations are mitigated in Eve by supplementing AutoProof with testing.

Separation logic is an extension of Hoare logic designed to handle frame properties; verification environments based on separation logic (e.g., jStar [8] for Java and VeryFast [16] for C and Java) can verify sophisticated features such as usages of the visitor, observer, and factory patterns. Writing separation logic annotations requires considerably more expertise than using contracts embedded in the programming language; this makes separation-logic tools more challenging to use by practitioners.

Other static verification techniques, such as software model-checking [3] and abstract interpretation [6], approximate the semantics of programming languages to make their analysis scalable and to require little annotations. These techniques are currently unsupported in Eve, but they may become as part of future work.

Dynamic verification. Only recently have dynamic techniques, such as testing, become applicable fully automatically to large programs (e.g., [13,17,4]). In this line of work, DART [13] introduced the concept of dynamic symbolic execution, a combination of dynamic verification with lightweight static techniques. CUTE [23] and EXE [4] follow similar approaches but they are applicable to more complex features (such as pointers and complex data structures) and scale massively. The main high-level difference of AutoTest is that it relies on contracts to verify functional properties; the aforementioned testing tools, instead, work on languages without contracts and therefore are limited in the kinds of errors that they can detect.

In recent years, dynamic techniques have extended their domain of applicability to problems such as contract inference [9,29] and specification mining [1,7] which have traditionally been approachable only by static means. Future versions of Eve will integrate dynamic contract inference as implemented in our AutoInfer tool (sketched in Section 6).

Combined static/dynamic verification. Recently, a few authoritative researchers have pointed out the potential of combining static and dynamic techniques [22,10,24] to make verification more usable; the present paper concurs in this vision.

Some of the aforementioned testing tools [13,23,4] already leverage lightweight static analyses to boost the performance of automated testing. Pex [25] is another scalable automatic testing framework, which relies more heavily on static methods: it exploits a variant of dynamic symbolic execution where an automated theorem prover (Z3) analyzes the symbolic executions to improve code coverage. Pex uses parameterized unit tests [26] as specifications. This makes it possible to test fairly sophisticated properties, but it also requires users to produce specifications in this customized form; contract specifications, however, seem more palatable to practitioners [5].

DASH [22] combines static and dynamic verification with an approach extending the software model-checking paradigm [3]: DASH's algorithm to generate exhaustive

tests maintains a sound abstraction of the program, which can be used to construct automatically a correctness argument.

The few recent attempts at combining static and dynamic techniques tend to be specific conservative extensions of basic methods; the approach described in the present paper tries integration at a higher level to avail the complementarity of static and dynamic techniques to a larger extent.

4 The Tools of Eve

The integrated verification techniques currently available in Eve and illustrated in the preceding example session rely on two fundamental tools: AutoProof and AutoTest, which are now presented.

AutoTest. AutoTest [19]—now a standard component of commercial EiffelStudio—is a fully automatic contract-based testing tool. AutoTest generates objects by random calls to creation procedures. Preconditions select valid inputs and postconditions serve as oracles: every test case consists of the execution of a routine on objects satisfying its precondition; if executing the routine violate its postcondition or calls another routine without satisfying its precondition, the routine tested has a fault. A failing test case provides a concrete error report which is useful for debugging.

Like any dynamic technique based on execution, AutoTest handles every feature of the source language (Eiffel). Among its limitations, instead, is that random testing can take several hours to find the most subtle faults, and that complex specifications can exacerbate this problem.

AutoProof. AutoProof [20]—a more recent member of the Eiffel tool-set—is an automatic verification tool that translates Eiffel programs (with contracts) into annotated Boogie programs. AutoProof then uses the Boogie verifier [2] to check whether the Eiffel implementation satisfies its specification.

AutoProof improves on similar environments for static verification (e.g., Spec# [2]) by supporting some advanced language constructs such as function objects (agents in Eiffel terminology). Nonetheless, some features of Eiffel—most notably exceptions and floating point arithmetic—are still unsupported and routines using them are not adequately translated to Boogie. The performance of AutoProof depends on the quality of contracts available; accurate contracts improve the modularity of the analysis which can then also verify partial implementations.

5 The Advantages of Being Static and Dynamic

From a user's perspective, Eve's integration of static and dynamic tools can make verification more effective and agile in a variety of scenarios.

– Static verification is more modular and scales better to large systems made of several classes. It can also verify routines of deferred (abstract) classes which cannot be tested because they cannot be instantiated. This indirectly improves the performance of testing as well, because the testing effort can focus on routines or classes not proved correct.

Conversely, whenever testing uncovers a faulty routine, the static tool stops trying to verify that routine. This policy may be broken down to individual clauses: for example, if testing finds a run of *remove_left_cursor* (Table 1) violating the postcondition clause *count* = **old** *count* − 1, it may still be worthwhile to try to prove the other clause, *c.index* = **old** *c.index* + 1.

– Dynamic analysis provides concrete reports of errors, which make debugging easier. For example, the following trace documents the error in the creation procedure *make_default* of Table 1:

```
create {ARRAYED_LIST} l.make_default (1)
    -- Inside make_default:
    l_v := Void ; Precursor (1) ; extend (l_v) -- l_v is Void
```

– Classes that have a faulty creation procedure or are deferred cannot be instantiated; testing cannot proceed in this case unless the constructor is fixed or an implementation of every routine is available. Static techniques do not incur these limitations: as illustrated in the example of Section 2, they can verify individual implemented routines even if others in the same class are deferred.

– Many core libraries rely on routines implemented through calls to low-level external routines (typically, in the Eiffel case, C functions); an example was *is_equal* in Table 1. Such routines are inaccessible to static analysis but are still testable. The integrated results of static and dynamic analysis on classes with such external routines reinforce the confidence in the correctness of the overall system.

– The combination of static and dynamic analysis can help detect discrepancies between the runtime behavior of a program and its idealized model. Examples are overflows and out-of-memory errors, which are often not accounted for in an abstract specification assuming perfect arithmetic and infinite memory. Consider, for example, a routine that updates the balance of a bank account as a result of a deposit operation:

```
deposit (amount: INTEGER)
  require amount > 0
  do balance := balance + amount
  ensure balance > old balance
end
```

balance: *INTEGER*

AutoProof, which models the type *INTEGER* as mathematical integers, verifies that the routine is correct. AutoTest, however, still finds a bug which occurs when **old** *balance* + *amount* is greater than the largest integer value representable and *balance* overflows. It is then a matter of general policy whether one should change the postcondition or the implementation. In any case, the comparison of the results of static and dynamic analysis clearly highlights the problem and facilitates the design of the best solution. With default settings, Eve gives a null correctness score (Section 7) in such situations, which reflects the uncertainty and the need for further analysis.

– Complex contracts considerably slow down automatic testing, both because their runtime evaluation incurs a significant overhead and because random generation

takes a long time to build objects that satisfy complex preconditions. Contracts may even in some cases be non-executable because they involve predicates over infinite sets; for example, the invariant of a class modeling a hash function requires that the hash code of every possible object (an infinite set) be a nonnegative integer. Static techniques can help in all such scenarios: it may be easier to prove the correctness of a routine if the precondition is complex, and hence also stronger; complex postconditions boost modular verification.

These observations highlight the usefulness of treating proofs and tests as complementary and convergent techniques (even though they have often been pursued, in the past, by separate research sub-communities in software engineering). There is indeed no contradiction; in particular, with the purpose of tests being entirely defined as attempting to make programs fail [18], a useful (that is, failed) test is a proof that the program is not correct. The approach illustrated by Eve is then to combine tools that can prove a program correct (such as AutoProof) and tools that can prove a program incorrect (AutoTest); as soon as a user has written a new program element, the two will start in parallel, each with its own specific goal, prove or disprove; in favorable situations, one of them will reach its goal fast, providing the user with a fast response of correctness or incorrectness.

6 The Design of an Integrated Verification Environment

The Eve integrated verification environment is built on top of the EiffelStudio IDE and supplements it with functionalities for verification.

Contracts. The choice of Eiffel as programming language ensures that we rely on formal specification elements embedded in the program text as *contracts* (pre and postconditions, class invariants, and other annotations). Since correctness is a relative notion (being dependent on a specification), every verification activity requires *some form* of specification. Empirical evidence suggests that formal specifications in the form of contracts are a good compromise between the rigor required by formal techniques and the kind of effort that practitioners are able, or willing, to provide [5,21].

Not all contracts must be written by programmers: the architecture of Eve can accommodate components for *specification inference* (see Section 3) to help users write better and stronger contracts. This particular property, however, is not emphasized in the present paper, which focuses on the integration of static and dynamic analysis assuming some contracts are available.

Automation. A defining characteristic of the verification tools in Eve is that they are automatic and can do most of the work without any explicit input from the user, assuming the presence of contracts which Eiffel programmers are already used to provide [5]. In order to decouple the machinery of the individual verification tools and to filter out their output, Eve relies on a blackboard architecture [15] shown in Figure 2.

A *controller* is responsible for triggering the various tools when appropriate, invisibly to the users. The controller bases its decisions on what the user is currently doing, which resources are available, and the results of previous verification attempts. The latter are collected in a *data pool* where every verification tool stores the results of its

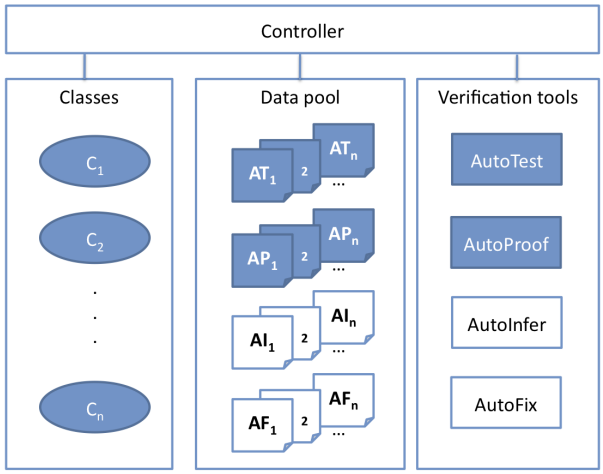

Fig. 2. Eve's blackboard architecture (shadeless boxes currently not implemented).

runs. Users do not directly read the output of individual tools in the data pool. Instead, the controller summarizes the output data and displays it only when appropriate, that is when the cumulative data collected is significant.

A major design decision of Eve was to make the verification mechanisms as unobtrusive as possible. Users can continue using the IDE and their preferred software development process as before; the verification techniques are an additional benefit, available on demand and compatible with the rest of the IDE's tools. In the same way that type checking adds a new level of help on top of the more elementary mechanisms of syntax error checking, Eve provides reports from proofs and tests on top of the simple verification techniques provided by type checking.

Interaction with the user. Users only have a coarse-grained, binary form of control over the verification: enable or disable. Typically, they will enable verification as soon as some code and a few contracts have been written. Even when enabled, verification never interferes with the more traditional development activities: Eve works in the background on the latest compiled version of the system, and displays a summary of the verification results through an interface similar to that used to signal syntax or type errors in standard IDEs (Figure 1). At any time, the user can browse through the *result list*, which links back to the parts of the program relevant for each message, and decide to revise parts of the implementation or specification according to the suggestions provided by Eve.

Every entry in the result list has a *score*: a quantitative estimate of the correctness or incorrectness of the associated entry, based on the evidence gathered so far by running the various tools. The score varies over the real interval $[-1, 1]$ (In the user interface the scale is, for more readability, -100 to $+100$, with rounding to the closest integer). A positive score indicates that the evidence in favor of correctness prevails, whereas a negative score characterizes evidence against correctness. The absolute value of the

score indicates the level confidence: 1 is conclusive evidence of correctness (for example a successful correctness proof), -1 is conclusive evidence of incorrectness (for example a failing test case), and 0 denotes lack of evidence either way. Figure 1 shows an example of report with scores and stripes colored accordingly. Section 7 discusses how the score is computed for the verification tools currently integrated in Eve.

Modularity and granularity. Object-oriented design emphasizes modularity, from which verification can also benefit. While the level of granularity achievable by an integrated verification environment ultimately depends on the level of granularity provided by the tools it integrates, Eve orients verification at the two basic levels of encapsulation provided by the object-oriented model: *classes* and *routines* within a class. Eve associates *correctness scores* with items at both levels. Additional information may be attached to a correctness score, such as the line where a contract violation occurs in a test run, or the abstract domain used in an abstract interpretation analysis. For large systems, it is also useful to have scores for highest levels of abstraction, such as groups of classes or entire libraries, but in the present discussion we limit ourselves to routine and class levels.

The scores from multiple sources of data at the same level are combined with weighted averages, and define the correctness scores at coarser levels. For example, every tool t tries to verify a routine r in class C and reports a correctness score $s_r^C(t) \in [-1, 1]$. The cumulative score for the routine r is then computed as the normalized weighted average:

$$s_r^C \;=\; \frac{1}{\sum_{t \in T} w_r^C(t)} \cdot \sum_{t \in T} w_r^C(t) s_r^C(t) \tag{1}$$

where $w_r^C(t) \in \mathbb{R}_{\geq 0}$ denotes the weight of the tool t on routine r. A similar expression computes the cumulative score s^C for a class C from the scores s_r^C of its routines and their weights w_r^C:

$$s^C \;=\; \frac{1}{\sum_{r \in R} w_r^C} \cdot \sum_{r \in R} w_r^C s_r^C \tag{2}$$

The weights take various peculiarities into account:

- A tool may not be able to handle certain constructs: its confidence should be scaled accordingly. For example, a tool unable to handle exceptions appropriately has its score reduced whenever it analyzes a routine which may raise exceptions.
- The results of a tool may be critical for the correctness of a certain routine. For example, a quality standard may require that every public routine be tested for at least one hour without hitting a bug; correspondingly, the weight $w_r^C(t)$ for public routines r would be high for testing tools and low (possibly even zero) for every other tool.
- The correctness of a routine may be critical for a class; then the routine score should have a higher weight in determining the class cumulative score.
- More generally, the weight may reflect suitable *metrics* that estimate the criticality of a routine according to factors such as the complexity of its implementation or specification, whether it is part of the interface of public, and the number of references to it in clients or within the containing class.

– Similar metrics are applicable at other levels of granularity, for example to weigh the criticality of a class within the system.

Eve provides default values for all the weights (Section 7), but users can override them to take relevant domain knowledge into account.

Resource usage. The verification layer must not impede more traditional development activities. This requires in particular a careful usage of the resources to guarantee that the responsiveness of the IDE when verification is enabled does not cause a significant overhead. In Eve, the controller activates the various verification tools only when there are resources to spare. It also takes into account the peculiarities of the various tools: those with short running times are usually run first and re-run often, while those requiring longer sessions are activated later, only if the faster tools did not give conclusive results, and when enough resources are available.

When verification is first activated for a project, all the scores are null, as the system does not have any evidence to assess correctness or incorrectness. Then, the controller runs the least demanding tools first, to provide the user with *some* feedback *as soon as possible*. The same approach is applied modularly, whenever some part of the system changes (and is recompiled). More detailed analysis is postponed to when more time is available, the system is sufficiently stable, or conclusive evidence is still lacking about the correctness of some routines or classes.

Extensibility. The architecture of Eve is *extensible* to include more tools of heterogeneous nature. The user interface will stay the same, with the blackboard controller being responsible for managing the tools optimally and only reporting the results through the summary scores described above. It is our plan to integrate more verification tools, currently available only through explicit invocation.

The architecture can also accommodate tools that, while not targeted to verification in a strict sense, enhance the user experience. For example, tools for assertion inference—such as our own AutoInfer [28]—can complement user-provided contracts and improve the performance of approaches that depend on contracts. The controller can activate assertion inference when the verification machinery performs poorly and when metrics suggest that the code is lacking sufficient specifications. The assertion inference tools themselves may sometimes re-use the results of other tools; for example AutoInfer relies on the test cases generated by AutoTest. Finally, Eve can show the inferred assertions in the form of suggestions, in connection with the results of other verification activities. For example, it could display an inferred loop invariant with the report of a failed proof attempt, and suggest that the invariant can make the correctness proof succeed if added to the specification. The current implementation of Eve does not integrate such suggestions mechanisms yet, but the architecture is designed with these extensions in mind.

7 Correctness Scores for Proofs and Testing

Equation 1 on page 10 gives the correctness score for a routine r of class C; now, consider a set of tools $T = \{p, t\}$, where p denotes AutoProof and t denotes AutoTest.

General principles for scores. We noted earlier that an interesting test, that is to say a failed test, is a proof of incorrectness. This is of course another form of Dijkstra's

famous observation about testing—but restated as an argument *for* tests rather than a criticism of the notion of testing. This observation has two direct consequences on the principles for computing correctness scores.

First, it is relatively straightforward to assign scores to the result of a testing tool when it reports errors: assign a score of -1, denoting certain incorrectness, to every routine where testing found an error. In certain special circumstances, the score might be differentiated according to the severity of the fault; for example a bug that occurs only if the program runs for several hours may be less critical than one that occurs earlier, if the system is such that it is reset every 45 minutes. In most circumstances, however, it is better to include such domain information in the specification itself and to treat every reported fault as a routine error. Then, different routines may still receive a different weight in the computation of the score of a class (Equation 2 on page 10)—for example, a higher weight to public routines with many clients.

The second consequence is that it is harder to assign a positive score sensibly to routines passing tests without errors. It is customary to assume that many successful tests increase the confidence of correctness; hence, this could determine a positive correctness score, which increases with the number of tests passed, the diversity of input values selected, or the coverage achieved according to some coverage criteria such as branch or instruction coverage. In any case, the positive score should be normalized so that it never exceeds an upper limit strictly less than 1, which denotes certain correctness and is hence unattainable by testing.

Since static verification tools are typically sound, a successful proof should generally give a score of 1. Certain aspects of the runtime behavior, such as arithmetic and memory overflows as discussed above, may still leak in some unsoundness if the static verifier does not model them explicitly; in such cases the score for a successful proof may be scaled down in routines with a specification that depends on such aspects.

Which score to assign to a static verifier reporting a failed proof attempt depends on the technique's associated guarantee of *completeness*. For a complete tool, a failed proof denotes a certain fault, hence a score of -1. If the tool is incomplete, a failed proof simply means "I don't know"; whether and how this should affect the score depends on the details of the technique. For example, partial proofs may still increase the evidence for correctness and yield a positive score.

Score and weight for AutoTest. If AutoTest reports a fault in a routine r of class C, the correctness score $s_r^C(t)$ becomes -1. This score receives a high weight $w_r^C(t) = 100$ by default; the user can adjust this value to reflect specific knowledge about the criticality of certain routines over others with respect to testing.

When AutoTest tests a routine r of class C without uncovering any fault, the score $s_r^C(t)$ increases proportionally to the length of the testing session and the number of test cases executed, but with an upper limit of 0.9. With the default settings, this maximum is reached after 24 hours of testing and 10^4 test cases executed without revealing any error in r. Users can change these parameters; the default settings try to reflect the specificities of random testing shown in repeated experiments [30]. We decided against using specific coverage criteria such as branch coverage in the calculation of the routine score, as the experiments suggest that for example the correlation between branch coverage and the number of uncovered faults is weak.

Score and weight for AutoProof. AutoProof implements a sound but incomplete proof technique. The score $s_r^C(p)$ for a routine r of class C is set accordingly: a successful proof yields a score of 1; an out-of-memory error or a timeout are inconclusive and yield a 0; a failed proof with abstract trace may be a faint indication of incorrectness: the abstract trace may not correspond to any concrete trace (showing an actual fault), but it often suggests that a proof might be possible with more accurate assertions. The score is then -0.2 to reflect this heuristic observation.

The weight $w_r^C(p)$ takes into account the few language features that are currently unsupported (floating point numbers and exceptions, see Section 3): if r's body contains such features, $w_r^C(p)$ is conservatively set to zero. In all other cases, $w_r^C(p)$ is 1 by default, but the user can adjust this value.

Routine weights. Equation 2 (page 10) combines the scores s_r^C of every routine r of class C with weights w_r^C to determine the cumulative score of C. The weights w_r^C should quantify the relevance of routine r for the correctness of class C. This depends in general on the overall system design, which only developers can express appropriately, but which often depends on the visibility of a routine.

Eve supports a simple way to enter this piece of information: every routine has an optional *importance* flag which takes the values *low* and *high*. w_r^C is then

$$w_r^C \;=\; v_r^C \cdot i_r^C$$

The visibility of r determines v_r^C, which is 2 if r is public and 1 otherwise. The importance of r determines i_r^C, which is 2 if r has high importance, $1/2$ if it has low importance, and 1 it the developer did not set the importance flag.

8 Usage Scenarios

How serviceable is Eve's score which combines the results of different verification tools, as opposed to considering the tools' outputs individually? This section outlines a few straightforward scenarios that compare the output given by AutoProof or AutoTest in isolation against Eve's combined output; they show the greater confidence supplied by Eve, and the straightforward interpretability of its output. The example[1] models attributes of an individual with a class *PERSON*. Table 2 lists 5 routines of the class to be verified; for each routine, the table reports the score and weight of AutoProof and AutoTest within Eve, and the corresponding combined score.

Routine *set_age* demonstrates a favorable scenario, where each tool can provide strong positive evidence indicating correctness. The overall score is, correspondingly, quite high, but it still falls short of the maximum because testing can never prove the absence of errors with 100% confidence.

Routine *increase_age* includes integer arithmetic, which might produce overflow. AutoProof can verify the routine, but Eve is aware that the proof scheme models integers as mathematical integers, hence it weights down the value of the successful proof

[1] The complete source code of the example is available at:
`http://se.ethz.ch/people/tschannen/sefm2011_example.zip`.

Item	Tool	Result	Weight	Score
set_age	AutoProof	Verified successfully	1.0	1.00
	AutoTest	No errors found	1.0	0.90
	Routine score			**0.95**
increase_age	AutoProof	Verified successfully	0.5	1.00
	AutoTest	Overflow detected	100.0	-1.00
	Routine score			**-0.99**
age_difference	AutoProof	Verified successfully	0.5	1.00
	AutoTest	No errors found	1.0	0.90
	Routine score			**0.93**
set_name	AutoProof	Proof failed	0.5	-0.10
	AutoTest	No errors found	1.0	0.90
	Routine score			**0.57**
body_mass_index	AutoProof	Inapplicable	0.0	0.00
	AutoTest	No errors found	1.0	0.90
	Routine score			**0.90**
apply_command	AutoProof	Verified successfully	1.0	1.00
	AutoTest	Inapplicable	0.0	0.00
	Routine score			**1.00**
PERSON	**Class score**			**0.56**

Table 2. Individual and combined results for class *PERSON*.

because the abstraction may overlook overflow errors. Indeed, AutoTest reveals an overflow when executing the routine with the maximum integer value. The combined score indicates that there is an error, which AutoTest discovered beyond the limitations of AutoProof. Another routine *age_difference* also uses integer arithmetic but it is correct. Eve still scales down AutoProof's score accordingly; in this case, however, AutoTest does not find any error, hence the overall score grows high: the uncertainties of the two tools compensate each other and the cumulative score indicates confidence.

Routine *set_name* relies on the object comparison semantics, which AutoProof overapproximates. In this case, a failed proof does not necessarily indicate an error in the routine, hence it only accounts for a mildly negative score. When AutoTest does not find any error after thorough testing, the combined score becomes visibly positive, while still leaving a margin of uncertainty given the lack of conclusive evidence either way.

Routines *body_mass_index* and *apply_command* demonstrate how Eve's combination of tools expands the applicability of verification: *body_mass_index* uses floating point arithmetic, unsupported by AutoProof, whereas *apply_command* uses agents, unsupported by AutoTest. Eve relies entirely on the only applicable tool in each case.

The overall class score (last line of Table 2) uses a uniform weight for the routines; the score concisely indicates that considerable effort has been successfully invested in the class's verification, but some non-trivial issues are open.

9 Conclusions

Eve improves the usability of the individual verification tools by integrating them into an environment which features: automation and minimal direct user interaction; modularity at class and routine level; and extensibility with new tools. The current implementation of Eve [11] combines a static verifier for Hoare-style proofs and a dynamic contract-based testing framework. The present paper has shown how these two techniques can be used in combination to improve the overall productiveness of verification.

Limitations and future work. In some situations, the integration of proofs and tests is still ineffective and provides an unsatisfactory user experience:

- When testing is the only technique applicable, it may be difficult to provide users with fast feedback. Automated random testing is very effective at finding delicate and unexpected bugs, but may require long sessions.
- The analysis of correct routines that use certain sophisticated language features—beyond those currently supported by AutoProof—may be inconclusive: testing does not find any error, but this is no substitute for a correctness proof.
- The completeness of contracts strongly affects the performance of verification. Weak contracts are easier to write and to reason about; strong contracts boost modular verification and expose subtler defects.
- Eve integrates multiple verification tools to complement their strengths and weaknesses. Different tools, however, may introduce discrepant models of the same implementation, such as for the class *INTEGER* discussed above. As the number of integrated tools grows, reconciling several contradictory semantics may become a delicate issue.

Future work will address these limitations to perfect the integration of testing and proofs; in particular, the following directions deserve further investigation.

- If AutoProof successfully verifies an assertion clause, the runtime checking of that specific clause can be disabled; this would contribute to speeding up the testing process. This improvement is currently unsupported because it requires a change in the Eiffel runtime to enable and disable the checking of individual assertion clauses.
- Integrating contract inference tools, such as our own AutoInfer [28], will assuage the problem of weak contracts that hold back the full potential of static provers. Another related synergy between static and dynamic techniques is the static verification of dynamically *guessed* contracts.
- A failed proof attempt usually comes with an *abstract* counterexample trace, which is, in general, not directly executable. The abstract trace may, however, provide enough information to suggest a *concrete* trace that is executable and show a real bug, or to conclude that the abstract trace is spurious. A spurious trace can help refine the proof model and sharpen the proof attempt, in a way similar to what done in the CEGAR (Counter-Example Guided Abstraction Refinement) paradigm [3].

The integration of more tools into Eve will improve the overall effectiveness of the various techniques and advance the quest towards the goal of *Verification As A Matter Of Course*.

Acknowledgements. The authors thank Nadia Polikarpova and Yi Wei for suggesting examples discussed in the paper.

References

1. G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL*, pages 4–16, 2002.
2. M. Barnett, R. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *CASSIS*, volume 3362 of *LNCS*, pages 49–69. Springer, 2004.

3. D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast. *STTT*, 9(5-6):505–525, 2007.
4. C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12, 2008.
5. P. Chalin. Are practitioners writing contracts? In *The RODIN Book*, volume 4157 of *LNCS*, page 100, 2006.
6. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.
7. V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. In *ISSTA*, July 2010.
8. D. Distefano and M. J. Parkinson. jStar: Towards Practical Verification for Java. In *Proceedings of OOPSLA*, pages 213–226, 2008.
9. M. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington, US, 2000.
10. M. D. Ernst. How tests and proofs impede one another: The need for always-on static and dynamic feedback. In *TAP*, volume 6143 of *LNCS*. Springer, 2010.
11. Eve: Eiffel verification environment. `http://se.inf.ethz.ch/research/eve/`.
12. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245. ACM, 2002.
13. P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proceedings of PLDI*, pages 213–223. ACM, 2005.
14. C. A. R. Hoare and J. Misra. Preface to special issue on software verification. *ACM Comput. Surv.*, 41(4), 2009.
15. J. Hunt. Blackboard architectures, 2002. JayDee Technology Ltd 27.
16. B. Jacobs, J. Smans, and F. Piessens. A quick tour of the VeriFast program verifier. In *Proceedings of APLAS 2010*, 2010.
17. B. Korel. Automated test data generation for programs with procedures. In *Proceedings of ISSTA*, pages 209–215. ACM, 1996.
18. B. Meyer. Seven principles of software testing. *Computer*, 41:99–101, 2008.
19. B. Meyer, A. Fiva, I. Ciupa, A. Leitner, Y. Wei, and E. Stapf. Programs that test themselves. *IEEE Software*, 42:46–55, 2009.
20. M. Nordio, C. Calcagno, B. Meyer, P. Müller, and J. Tschannen. Reasoning about Function Objects. In *Proceedings of TOOLS-EUROPE*, LNCS. Springer, 2010.
21. N. Polikarpova, I. Ciupa, and B. Meyer. A comparative study of programmer-written and automatically inferred contracts. In *ISSTA*, pages 93–104, 2009.
22. S. K. Rajamani. Verification, testing and statistics. In *FM 2009: 2nd World Congress on Formal Methods*, volume 5850 of *LNCS*, pages 33–40. Springer, 2009.
23. K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of ESEC/FSE*, pages 263–272. ACM, 2005.
24. International conference on tests and proofs. Springer LNCS, 2007–2010.
25. N. Tillmann and J. de Halleux. Pex–White Box Test Generation for .NET. In *Proceedings of TAP*, pages 134–153, 2008.
26. N. Tillmann and W. Schulte. Parameterized unit tests. In *Proceedings of ESEC/FSE*, pages 253–262. ACM, 2005.
27. Usable verification workshop. `http://fm.csl.sri.com/UV10/`, November 2010.
28. Y. Wei, C. A. Furia, N. Kazmin, and B. Meyer. Inferring better contracts. In *Proceedings of ICSE'11*, 2011. To appear.
29. Y. Wei, S. Gebhardt, M. Oriol, and B. Meyer. Satisfying test preconditions through guided object selection. In *Proceedings of ICST'10*, pages 303–312, 2010.
30. Y. Wei, M. Oriol, and B. Meyer. Is coverage a good measure of testing effectiveness? Technical Report 674, ETH Zurich, 2010.