Eiffel as a Framework for Verification

Bertrand Meyer

ETH Zurich
http://se.inf.ethz.ch
Eiffel Software
www.eiffel.com

Abstract. The Eiffel method and language integrate a number of ideas originating from work on program verification. This position paper describes the goals of the Eiffel approach, presents current Eiffel-based verification techniques using contracts for run-time checks for testing and debugging, and outlines ongoing work on static verification.

1 Application Areas

Advances in programming languages, especially object-oriented principles and techniques, have profoundly influenced the production of software. Most development teams are not ready to renounce these gains in expressive power, including both:

- Static mechanisms: classes, information hiding, genericity, inheritance (including multiple and repeated), polymorphism, smart type systems, conversions, once features, contracts.
- Dynamic mechanisms: objects, references (type-safe pointers), exception handling, dynamic object creation, automatic garbage collection.

Much of the work towards verified software explicitly gives up on these facilities, preferring to use minimal programming languages; the goal is to remove as many obstacles as possible in the quest for full verification, in particular proofs.

While this approach has been successfully applied to certain areas, in particular life-critical systems for which the goal of verification overrides all other concerns, it is not realistic for the vast majority of industry applications. This is not just a matter of programmer comfort. Modern programming constructs are there for a reason: they make it possible to turn out systems faster and at lesser cost; they favor extendibility (ease of change) and reuse, both essential in industry practice; they also provide a decisive boost to software quality, by eliminating whole classes of errors — type errors in the case of static typing, memory errors in the case of garbage collection — hence easing the task of verification.

Such software engineering concerns are crucial to many industries. A typical example is financial software. A development model that freezes the specification and then devotes two years to producing a program guaranteed to meet that specification, using a low-level language and extensive verification techniques, would just not work in the world of financial systems: specifications change frequently and new ideas for products or strategies, critical to a company's survival, need to be implemented in a

B. Meyer and J. Woodcock (Eds.): Verified Software, LNCS 4171, pp. 301--307, 2008.

matter of weeks or days. It is for this kind of environment, overwhelmingly dominant in industry, that modern development techniques and languages have been devised; managers and developers understand their benefits and will not forgo them.

This does not mean that they ignore other aspects of verification. In fact verification is almost as desirable for mainstream applications as it is — for example — for in-flight software. A software error in a financial system can cause loss of enormous amounts of money; bad investments, whether or not on the advice of a computer program, can land the company's CEO in jail for breach of fiduciary duty.

Our Eiffel-related work attempts to address the challenge of verifying software that takes advantage of the best programming language ideas and meets the software engineering constraints on mainstream industry projects.

2 Eiffel Goals

One of the distinctive features of the Eiffel approach is its effort to encompass the software engineering process as a whole, not just implementation. While also a language, Eiffel is more than anything else a method, covering the entire lifecycle. That method rejects the traditional separation of notations used at different stages: an analysis notation (usually graphical), perhaps a design notation, a programming language. Instead it uses the same formalism — the Eiffel language — for all tasks, from the most abstract and user-oriented stages of analysis and specification down to implementation, testing and maintenance. The use of a programming-like notation for analysis goes back to the advice of Kristen Nygaard, one of the inventors of object-oriented programming, who stated that "*to program is to understand*"; it is served in Eiffel by high-level modeling constructs such as deferred classes and contracts; see [13] for examples of such uses of the notation for purely modeling purposes, such as the specification of a TV station's scheduling, independently of any implementation concerns and in fact of any software aspects. This means that as a language Eiffel attempts to cover not only the traditional applications of programming languages but also the realm of analysis notations, specification languages and design tools. One of the advantages of such an integrated, seamless approach, where the software in all its aspects — description as well as implementation — is considered a single product, is to facilitate change, as there is only this one product to update. While retaining the attraction of "model-driven" approaches and their use of a high-level formalism, this technique avoids their separation between expression of intent and expression of realization. Suchg a gap can be, for large programs developed over a long period by many people and with many evolutions, detrimental to quality.

Another aspect relevant to verification work is that the very design of Eiffel was a direct result of verification concerns. In this respect Eiffel occupies a special place among mainstream tools; one has to go back to research languages such as Euclid to find the influence of similar concerns. In particular:

- Eiffel puts the notion of contract (specification of routines and classes in terms of preconditions, postconditions and class invariants) at the center of the method and notation. This idea, which of course follows directly from verification work

(Hoare semantics, Z and other specification languages) has a profound effect on how software is developed. The most important practical observation here is that Eiffel users do not view these techniques as "formal" (and hence to many people formidable) methods, simply as good analysis, design and programming practices closely integrated with the development process, and no more difficult to use than standard programming constructs such as conditionals.

- The contract mechanism is closely integrated with the object-oriented fabric of the language and in particular with inheritance and associated techniques of polymorphism and dynamic binding, through rules of invariant accumulation, precondition weakening and postcondition strengthening [13], variants of which are also present in other approaches using contracts such as JML [8] [9] and Spec# A. This makes it possible to harness advanced and potentially unsafe O-O techniques, in particular redefinition.
- Contracts are not a theoretical possibility but heavily used by Eiffel users in practice, as attested by studies of the actual code base [2].
- The exception mechanism of Eiffel differs from standard "try...catch" mechanisms by relying on the contract concept: an exception is not just some event that will be handled by a special control structure, but the indication that some operation failed to fulfill its contract; the task of exception handling is rigorously defined as an attempt to achieve the contract through another strategy or, if this is impossible, to pass on the issue to an agent higher up in the call chain, which might succeed in such a replacement strategy [13]. This gives a more systematic way of handling erroneous and special cases, a delicate component of software correctness (as attested for example by the famous Ariane 5 failure [7]).
- Eiffel development is supported by libraries of reusable software elements such as EiffelBase [12], refined over two decades, and extensively specified through contracts. The MML (Mathematical Model Library) provides a technique for completely specifying classes through models [16], and has been used to define a fully formally specified subset of EiffelBase. The advantage here is to allow complete specifications within the framework of the language, without adding higher-level, non-executable constructs such as first-order quantifiers.
- The type system includes powerful mechanisms of constrained and unconstrained genericity, multiple and repeated inheritance, tuples and agents. This enables Eiffel users to construct sophisticated models and rely on the compiler to perform advanced checks that amount to proofs of consistency. A recent addition to the type system takes these ideas further by sttically removing from the language the possibility of void calls (attempts to dereference null pointers) through a compile-time check [14].
- A concurrency mechanism, SCOOP, extends the notion of contract to concurrent programs of widely different kinds — multi-threading, multitasking, Web services, distribution —, with a precisely defined semantics [15].
- The language specification [4], while not formal, is strongly influenced by formal techniques; in particular it gives all static semantic rules in "if and only if" form, guaranteeing the validity of a construct if it satisfies certain properties. This provides programmers with an increased confidence in their basic tool

The next two sections describe benefits that can be derived from Eiffel's contract techniques today, and new developments currently in progress to meet the objectives of the verification Grand Challenge.

3 Current Applications of Contracts

The traditional applications of contracts available in Eiffel include [13]: better analysis and specification (as compared for example to purely graphical notations) through an encouragement to describe the precise semantics of system elements; guidance for the design and implementation process through encouragement to state not only how the software works but what it is supposed to achieve; automatic documentation, as offered in the EiffelStudio environment [5]; support for project management, by enabling managers to understand the essentials of a system without having to read the detailed code; support for evolution, by leaving a clear trace of key design decisions independently of implementation, and retaining the work of the best designers even when they have left; support for reuse; safe use of inheritance, as noted above; support for debugging and testing, through run-time monitoring of assertions.

Recent developments have extended these techniques, in particular the last application mentioned. For a long time Eiffel developers have been accustomed to the benefit of having bugs detected through run-time contract violations during debugging and testing. This is a much more effective way than having to prepare test oracles manually. Two important tasks, however, have so far remained manual: test case preparation, and integration of failed tests in the regression testing database.

For the first task, we have developed the AutoTest framework for automatic testing [3], which provides push-button testing of classes, without any human intervention such as preparing test cases. The basic idea is simple: AutoTest takes a set of classes and automatically produces numerous valid instances of these classes, then executes numerous calls to all their routines with arguments selected through various strategies, waiting for a postcondition or invariant to be broken. This always signals a bug. While the approach may at seem naïve, it is actually effective by the only criterion that counts: it finds real bugs (not artificially seeded ones) in actual software, including released libraries and production applications. Work is proceeding to integrate AutoTest in the EiffelStudio environment and ensure automatic, continuous background testing of software as it is being developed. What makes a fully automatic process possible is the presence of contracts, which provide the test oracles. Contracts are, as noted, a natural component of software for Eiffel programmers, allowing AutoTest to work effectively on software as it is written, rather than software that has to be instrumented for verification purposes. Here the approach benefits from *not* being a fully formal method: while proofs require complete specifications, tests as performed by AutoTest (and more classically by monitoring contracts at execution) can take advantage of any contract elements, however partial, which the programmers have cared to write.

Building on some of the same ideas, the CDD tool (Contract-Driven Development [10]) integrates failures found during development, typically through contract violations, into regression testing. The idea is that any failed test is precious information about the project and should forever become part of its test base, automatically replayed

— without explicit requests by the programmer — after any new compilation or release. It is a common phenomenon of software development that bugs have many lives; they will pop back even when thought to have died once or more. CDD, integrated in an experimental version of EiffelStudio, makes sure to reëxecute in the background every test that ever failed.

For both AutoTest and CDD, a necessary task is *test case minimization* [10], which for any execution sequence that led to a failure produces another sequence, usually much shorter, producing the same effect. This is essential if these execution sequences are to become part of the regression test database and hence be reëxecuted frequently.

4 Proofs

Eiffel's contracts have so far been applied mostly to dynamic checks, because the benefits are so clear and immediate. With improvements in proof technology — including semantic modeling, theorem provers, abstract interpretation and model checking — it becomes attractive to support proofs, as has already been the plan behind Eiffel. Several efforts are in progress at ETH and elsewhere, in particular the Ballet proof environment based on BoogiePL, the development of a full formal semantics for Eiffel including its most advanced constructs (by Martin Nordio, in collaboration with Peter Müller), and the formalization of SCOOP semantics [15]. We hope in the future to follow the lead of such developments as the Spec# framework and integrate proof technology, as unobtrusive as possible, into the EiffelStudio environment.

This will not remove the need for other approaches, in particular support for automatic testing and debugging. Proofs and tests, long considered rivals, are in fact complementary, if only because not all proofs can succeed and a failed proof can help narrow down the issues and devise better tests. The Tests And Proofs conference organized at ETH after VSTTE [6] has explored that complementarity.

Eiffel, as noted, was designed from the start with a central concern for verifiability. The long experience of designing Eiffel software — including some very large systems driving stock exchanges, simulating complex environmental or defense problems, managing billions of investment dollars, handling complex health care needs — with contracts and a constant search for quality provides the verification community with useful lessons; by contributing Eiffel concepts, tools such as the open-source EiffelStudio, carefully crafted component libraries such as EiffelBase, specification libraries such as MML, as well as books and teaching materials to the Grand Challenge effort, we hope to help in the search for fully verified software.

References

[1] Barnett, M., Leino, R., Schulte, W.: The Spec# Programming System: An Overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)

[2] Chalin, P.: Logical Foundations of Program Assertions: What do Practitioners Want? ENCS-CSE Technical Report 2005-05, revision 02, Concordia University (June 2005)

[3] Ciupa, I., Leitner, A., Liu, L(L.)., Meyer, B.: Automatic testing of object-oriented software. In: van Leeuwen, J., Italiano, G.F., van der Hoek, W., Meinel, C., Sack, H., Plášil, F. (eds.) SOFSEM 2007. LNCS, vol. 4362, pp. 114–129. Springer, Heidelberg (2007), http://se.ethz.ch/~meyer/publications/lncs/testing_sofsem.pdf
[4] ECMA Technical Committee 39 (Programming and Scripting Languages) Technical Group 4 (Eiffel): Eiffel Analysis, Design and Programming Language, ECMA and ISO standard, June 2005 (revised November 2006)
[5] Eiffel Software: EiffelStudio open-source download, http://www.eiffel.com
[6] Gurevich, Y., Meyer, B. (eds.): TAP 2007. LNCS, vol. 4454. Springer, Heidelberg (2007)
[7] Jézéquel, J.-M., Meyer, B.: Design by Contract: The Lessons of Ariane. In: Computer (IEEE), vol. 30(1), pp. 129–130 (1997), http://se.ethz.ch/~meyer/publications/computer/ariane.pdf
[8] Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C., Cok, D.R.: How the Design of JML Accommodates Both Runtime Assertion Checking and Formal Verification. In: Science of Computer Programming, vol. 55(1-2), pp. 185–208 (March 2005), http://dx.doi.org/10.1016/j.scico.2004.05.015
[9] Leavens, G.T., Cheon, Y.: Design by Contract with JML, draft paper. (with other JML documents from JML), www.eecs.ucf.edu/~leavens/JML/
[10] Leitner, A., Oriol, M., Zeller, A., Ciupa, I., Meyer, B.: Efficient Unit Test Case Minimization. In: proceedings of Automated Software Engineering 2007 (ASE 2007) (to appear)
[11] Leitner, A., Ciupa, I., Oriol, M., Meyer, B., Fiva, A.: Contract Driven Development = Test Driven Development – Writing Test Cases. In: proceedings of ESEC/FSE 2007 (to appear)
[12] Meyer, B.: Reusable Software: The Base Object-Oriented Component Libraries. Prentice-Hall, Englewood Cliffs (1994)
[13] Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice-Hall, Englewood Cliffs (1997)
[14] Meyer, B.: Attached Types and their Application to Three Open Problems of Object-Oriented Programming. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 1–32. Springer, Heidelberg (2005), http://se.ethz.ch/~meyer/publications/lncs/attached.pdf
[15] Nienaltowski, P.: Practical framework for contract-based concurrent object-oriented programming, PhD thesis, ETH Zurich (February 2007), http://se.ethz.ch/people/nienaltowski/papers/thesis.pdf
[16] Schoeller, B., Widmer, T., Meyer, B.: Making Specifications Complete Through Models. In: Reussner, R., Stafford, J., Szyperski, C. (eds.) Architecting Systems with Trustworthy Components. LNCS, Springer, Heidelberg (2006), se.ethz.ch/~meyer/publications/lncs/model_library.pdf

A Discussion on Bertrand Meyer's Presentation

Greg Nelson

Bertrand, I was intrigued by your remark in passing that Eiffel's type system has recently been extended, so that null-dereference errors are no longer possible in a type-safe program, if I understood you correctly, and I was wondering if this meant that your type checker uses automatic theorem proving techniques, or whether it means, that there is a new class of error, where a general expression is assigned to a variable, declared to be of a non-null type?

Bertrand Meyer

The solutions are not very different from those of Spec#, except we think they are better, and a good reason for this is that we took quite a few ideas from them, and it's of course always easier to improve on someone else's design than to start from scratch. In particular I think our approach is simpler.

Type checking is theorem proving already. So it is just a matter of making the theory a bit more powerful. In just two words: First, one of the differences with the Spec# work is that we decided that by default types are non-void—"attached", as we call them. That default seems to match reality, as in practice most business objects have to be there. Second, the key issue is automatic variable initialization. It seems that's all I have the time to say.