ES-Verify: A Tool for Automated Model-based Verification of Object-Oriented Code

Jonathan S. Ostroff[1], Chen-Wei (Jackie) Wang[1], Faraz Ahmadi Torshizi[1], and Eric Kerfoot[2]

[1] Software Engineering Laboratory, Dept. of Computer Science and Engineering, York University, 4700 Keele Street, Toronto, Canada M3J 1P3
[2] Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, England.

Design by Contract (DbC) is a method for specifying software so that we can check that it behaves according to specification. A class can be specified via expressive preconditions, postconditions and class invariants. ESpec (Eiffel Specification) toolset is a unified environment (Fig. 1) allowing software developers to specify, develop, test and verify the requirements, design and implementation of a software product. It consists of the following three tools: (a) ES-Fit: Write and test customer requirements as Fit tables; (b) ES-Test: Unit testing of code as it is developed; and (c) ES-Verify: Automated model-based verification of code. All the tools are centered around the Eiffel's industrial-strength DbC facility. ESpec is the first implementation of Fit tables and automated verification for Eiffel.

The notion of a contract is central to the toolset with contract violations reported in all three tools. Testing and verification are integrated and displayed together with a green bar to report success in all tests and verification conditions or a red one otherwise. The following reports (at `cs.yorku.ca/techreports/2006`) may be consulted for more surveys and references of detail and literature. (a) CS-2006-04. ESpec – a Tool for Agile Development via Early Testable Specificationss. (b) CS-2006-05. Automated Model-based Verification of Object-Oriented Code. The website for the tool is `cs.yorku.ca/~sel/espec`.

ES-Verify: This report focusses on ES-Verify, which has an immutable mathematical model library (sets, sequences, bags and maps) for high-level specifications, refinement to efficient mutable classes in the base library, and a translator to the Perfect Developer (PD) specification language for automatically discharging complete verification conditions, including all primitive types, arrays, complete client-supplier relationships, genericity, loops (with loop variants and invariants) and partial support for the agent notation for quantification and set comprehension. ES-Verify consists of the following three components:

(a) An Eiffel Model Library (ML) for specifying the abstract state without exposing implementation details. This library is similar to model-based specifications as in B and Z, except that it is object-oriented. ML contains classes such as ML_SEQ, ML_SET, ML_BAG and ML_MAP. These classes are both mathematical (i.e. immutable) and effective (i.e. executable). They are mathematical so that software properties can be specified abstractly and effective so that when

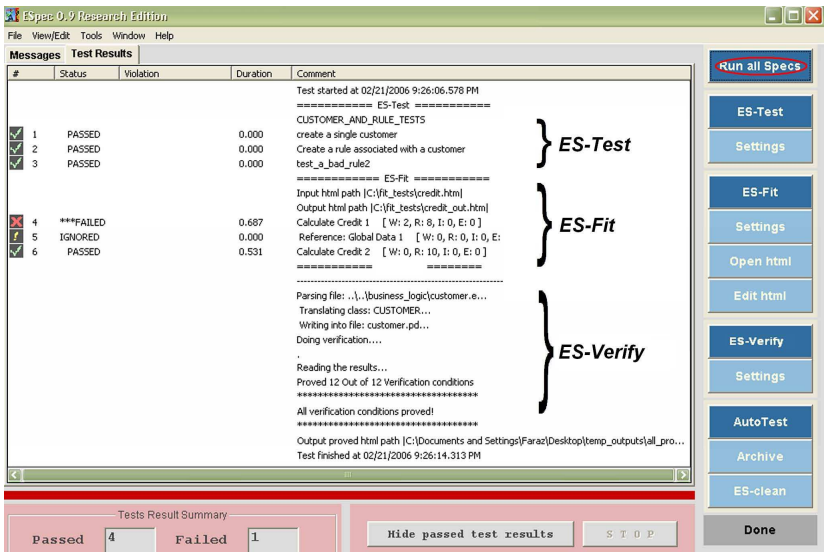

Fig. 1. ESpec toolset – integrated testing and verification

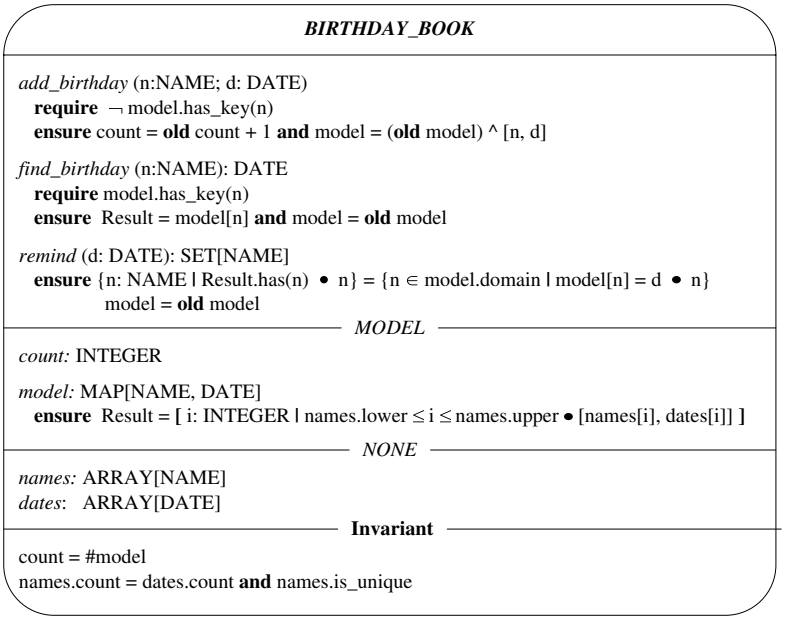

Fig. 2. BON Class and Contract Diagram for Birthday Book

the code (specified via ML) is executed, contract violations will be reported (if any). This mathematical library is thus useful for lightweight verification even in the absence of a theorem prover.

(b) A base library (ES_BASE) of data structures (with classes such as ESV_ARRAY, ESV_LIST, ESV_SET and ESV_TABLE) for the efficient implementation of software products. These classes have a value semantics, but for efficiency are mutable. The classes are descendants of the standard Eiffel base library classes and contracted via ML. The prefix ESV stands for Eiffel Spec Value (semantics).

(c) A translator that will convert Eiffel code implemented via ES_BASE and specified via ML into specifications written in the Perfect Language. The translator benefits from a highly-productive PD theorem prover (`eschertech.com`) for converting the specification (written in the Perfect Language) into complete verification conditions and automatically discharging their proofs.

Our approach is to write the code in Eiffel thus retaining the simple but expressive use of the language constructs. The Eiffel code is then translated to Perfect using (a) the refinement constructs of Perfect for the feature implementations and (b) the Perfect contracting mechanism for Eiffel contracts. The Eiffel model library (ML) was designed in order to avoid impedance mismatches between itself and the Perfect data structures.

The birthday book example (from J.M Spivey's *The Z notation*), nicely illustrates refinement to loops and intensive use of genericity and the mathematical modelling library ML as shown by the BON diagram in Fig. 2. The BON diagram shows the contract view of the class for each of the features. The model for the birthday book is the combination of the number of name-and-date pairs stored (i.e. `count`) together with an `ML_MAP[NAME,DATE]`, i.e. a set of pairs of name and date. The features of the birthday book include the ability to add a new pair (e.g. $[Peter, (March\ 1)]$), find a birthday given a name, and a `remind` function that for a given date d returns the set of names whose birthday is on d. The contract view can be refined to complete Eiffel code (not shown, but see aforementioned reports)

When the Eiffel-to-Perfect translator is applied to the Eiffel code for the birthday book example, the PD theorem prover generates 158 verification conditions which are *all* automatically discharged. This includes proof of termination via the loop variant and invariant.

Comparison to other tools:

The PD theorem prover appears to be at approximately the same level of proving power as B theorem provers. It is capable of dealing with all the primitive types including reals, quantification and set theory. For example, in one report 35,799 lines specification for a web-enabled database system was proven automatically in 4.5 hours (1.6 seconds per proof) on a modest laptop. PD is used to verify itself with about 130,000 verification conditions.

Tools like ESC/Java and Spec#/Boogie allow the developer to increase the confidence of already existing Java or C# code following an Annotated Development approach by adding specifications as annotations. Spark Ada is a successful

example of Annotated Development, but the specifications are usually only partial (in particular, expressing data refinement is difficult). When applied to an object-oriented language that uses reference semantics or makes heavy use of pointers, correctness has to be sacrificed in order to allow more potential bugs to be spotted, otherwise very little can be verified. Spark Ada instead preserves correctness by subsetting the Ada language. Some of the same considerations apply to ESC/Java and Spec# where the goal is to find bugs rather than prove total correctness. An interesting property of these tools is that they do not warn about *all* errors nor do they warn about actual errors *only*.

The updated version of the Java tool (ESC/Java2) compiles Java 1.4 but not Java 1.5 (which has the new facility for generic types). Spec# compiles generic types but does not yet verify them. These newer tools (following JML) include the ability to declare model fields and abstraction functions for representing the relationship between the value of the model field and the implementation so that refinements can be proved. However, these model variables are the standard mutable Java or C# set, list and map types, which makes it harder to specify that adding a new person and date tuple to the birthday book (for example) adds a new tuple without affecting the old tuples already in the database. By contrast, ML is immutable thus making it simple to specify such properties.

ES-Verify (and PD) follow an Abstract Specification and Refinement approach as in the Z and B-method. However, B allows ongoing refinement while ES-Verify allows only a single refinement step. This approach requires a notation that can adequately express both an abstract specification (which is not necessarily directly implementable, and need not have an associated implementation) and an implementation. Data refinement is a key feature, i.e. you develop the specification using an abstract data model (in our case ML), then refine the data model if necessary for an efficient implementation. The notation and the semantics are designed for correctness and provability (unlike traditional programming languages), so there is no need to sacrifice correctness.

Tools such as ESC/Java2 provide precise feedback as to where the error is (e.g. postcondition possibly not established at line Foo in the Java file Bar.java). Our tool does not yet provide such precise feedback. However, the output produced is informative (e.g. Refuted postcondition at line Foo in Bar.pd). That line number easily associates with the Eiffel feature having the same name or assertion tag, so that it is relatively easy to see where the problem is. We hope to improve this feedback in future versions.

Acknowledgements: We deeply appreciate the help we have received from David Crocker of Escher Technologies with the Perfect toolset. Likewise we would like to acknowledge helpful feedback from Bertrand Meyer and Bernd Schoeller of ETH Zurich. This work was funded by a Discovery Grant from NSERC.