An Eclipse Plug-in for Model Checking[*]

Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala
Electrical Engineering and Computer Sciences
University of California, Berkeley, USA

Rupak Majumdar
Computer Science Department
University of California, Los Angeles, USA

Abstract

While model checking has been successful in uncovering subtle bugs in code, its adoption in software engineering practice has been hampered by the absence of a simple interface to the programmer in an integrated development environment. We describe an integration of the software model checker BLAST *into the Eclipse development environment. We provide a verification interface for practical solutions for some typical program analysis problems —assertion checking, reachability analysis, dead code analysis, and test generation— directly on the source code. The analysis is completely automatic, and assumes no knowledge of model checking or formal notation. Moreover, the interface supports incremental program verification to support incremental design and evolution of code.*

1. Introduction and Motivation

Model checking is a well known technique for ensuring correctness of abstract models. However, its potential in improving software quality has, until now, been largely unrealized. This is mainly because of three reasons. First, too much effort is required in manually constructing an abstract model. Second, such models often track too many program facts, leading to state explosion and failure to scale to large programs. Third, to the engineer, there still remains the barrier of complicated formal notation that must be mastered.

During the last few years several new techniques have evolved that, to a large extent, ameliorate the first two problems. First, automatic counterexample-driven refinement methods [3] can construct abstract models directly from source code in a property guided manner. Second, non-uniform abstractions [6] control the state space explosion, and automatically select the minimal information required to prove a property, thus allowing model checking analysis to scale to large programs. However, little work has been done to incorporate a model checker into the software design flow. The model checker is still a separate tool that the engineer must master. To overcome this problem, we have designed a plug-in for the model checker BLAST [6] into the *Eclipse Development Environment*[1], which allows the engineer to apply sophisticated techniques for assuring software quality through a simple, easy-to-use interface for common program analysis tasks.

The dominant software quality assurance tool used today is testing. Unfortunately, for complex systems, testing is often inadequate as a tool to guarantee system reliability. Testing cannot prove the absence of errors, only indicate their presence. In contrast, model checking can provide a *certificate of correctness*, which can serve as the guarantee the engineer wants to have. Moreover, model checking does not require generating and evaluating test sets manually. At the same time, model checking may not scale to large programs and data intensive properties. Hence, we cannot simply replace testing by model checking. Rather we propose to complement the quality assurance process by applying model checking techniques in addition to traditional testing. In our plug-in, model checking is used to support both direct verification of correctness during program development, as well as to generate test sets during the test phase.

Our interface to the model checker allows the programmer to apply the following analysis tasks during program development and verification: assertion checking, reachability analysis, dead code analysis, and test case generation.

Assertions, or invariants, have been used traditionally as a means to improve software quality by making programmer assumptions about correct executions explicit in the program [7]. Annotating programs with assertions is an important software engineering practice since many years. First, they make hidden faults visible at run-time. Second, they provide predicates for test generation, and often reduce test cases. Third, a proved assertion serves as program documentation in later re-engineering. Assertions are therefore introduced by the programmer or tester, and serve as lightweight specifications for correct behavior. Most lan-

* This research was supported in part by the NSF grants CCR-0085949, CCR-0234690, and ITR-0326577.

1 http://www.eclipse.org

guages already provide support to add assertions, which are converted to run-time checks by the compiler. However, the run-time checks may degrade the performance of the program, so in practice, these checks are often turned off. The model checker provides a technique to statically check whether an assertion holds. Once we have proved that an assertion holds, the corresponding run-time check could automatically be removed during the compilation process.

Reachability of program locations is a basic program analysis question: we ask if there is some execution of the program that reaches a certain label. All safety verification questions on a program can be reduced to reachability questions on an instrumented program. In fact, the basic engine of the BLAST model checker implements a program location reachability algorithm, and safety specifications are internally compiled to a reachability query. The model checker provides precise answers to reachability queries. The reachability query for location ℓ returns a feasible program execution trace to ℓ if it is reachable, and returns a proof that ℓ is unreachable otherwise.[2] Moreover, if a location is reachable, the program trace provided by the model checker can be used to generate an appropriate test case. Executing the test case causes program execution to reach the target location.

Dead (or unreachable) code analysis is associated with program reachability: a program location is dead if there is no execution of the program that reaches the location. Since dead statements do not add functionality, and can waste considerable amounts of memory [8], the analysis of unreachable code is an important task in software engineering. Dead code often indicates wrong use of predicates in conditional branching, or obsolete code that has not been deleted. In contrast to static control flow based tools, the model checker can provide a precise analysis that checks exactly which locations are unreachable. Again, for reachable locations, the tool generates appropriate test cases.

Test case generation asks, given a program location ℓ, to produce an input that takes the program execution to the location ℓ. Our plug-in provides test cases with node coverage criterion. In principle, we can handle other testing criteria as well.

Supporting common program analysis tasks is only one aspect of a software quality toolkit. Software evolves, and the toolkit must provide support for evolving code. Doing the analysis from scratch every time the code changes is usually too expensive. To support repeated verification during incremental software development efficiently, our model checker provides an *incremental verification* facility [5]. The system stores the abstract model for each verification task. When the model checker is invoked to re-check a property on a later version of the program, it changes the abstract model incrementally for the pieces of code that have been modified.

2 In theory, the reachability problem for a language like C is undecidable, so our analysis is not guaranteed to terminate. In practice, we find that non-termination is usually due to scalability, or limited expressive power of the predicate language, rather than to inherent undecidability issues.

2. Incremental Software Verification

Model Checking. BLAST is a verification tool for checking safety properties of C programs based on an *abstract-check-refine loop* [3] and *lazy abstraction* [6]. The abstract-check-refine loop starts with a coarse abstraction of the program, and iteratively checks the abstract program against the specification, refining the abstraction whenever there is a spurious bug in the abstract program owing to the imprecision of the model. This goes on until the program is proved safe, or a bug is found. Lazy abstraction is a tightly coupled implementation of this iterative process that searches the abstract space on the fly, and only refines the coarse abstraction along the path of the spurious bug, leaving the abstraction in other parts unchanged. This results in an inhomogeneous abstraction of the model, with fewer facts tracked at each point. Lazy abstraction offers a significant advantage in performance by keeping the abstract state space small, and by avoiding repetition of work from iteration to iteration within the loop.

Incremental Verification. In incremental program development, code is developed in conjunction with tests, and as code evolves, the tester writes new tests and ensures that old tests still run (and produce the right output) [1] ("regression testing"). Similarly, in incremental verification, the engineer writes a suite of properties that should hold of the system, and when code changes, the old properties are verified again to ensure they still hold. However, since verification by model checking is expensive, the verification environment must support incremental program development by stepwise refinement by analyzing which properties may be affected by a change in code, and by optimizing model checking effort by saving information from previous runs.

The idea behind our approach of *incremental verification* is similar to selective regression testing in that we want to spend computational effort only for changed parts of the program. Incremental verification in BLAST maintains the verification tasks, and also information from previous verification cycles [5]. The tool maintains the status (success or failure — i.e., whether the property holds or does not hold) and the abstraction computed by the model checker in the previous run for each verification task. The abstraction is a map from program locations to predicates over program states. This information is reused in the following way. When the code changes, the model checker finds which parts of the previous proof of safety can be affected by the change, and starts model checking from these parts, using the saved abstraction as the initial abstraction. The lazy abstraction loop ensures that little work is done if the saved abstraction is already precise enough to prove the property for the new program, and that the abstraction is refined for the changed program if direct reuse of the previous abstraction is not possible.

3. Verification Tasks

For effective verification, the engineer has to define a set of properties, which has to be maintained for later reuse (similar to regression tests). For testing, *testing frameworks* (e.g., [4]) are available that make the definition and maintenance of tests easy by providing an interface for defining, managing, and running test suites. In order to be effective, the verification system should similarly provide an effective way to manage (i.e., create, verify, maintain) *verification tasks*. Our plug-in provides an easy interface to the programmer to define and use verification tasks. A verification task specifies the object (source code files), the safety property, and all parameters and results for one single invocation of the model checking engine. Each verification task maintains a status (checked, safe, unsafe, unchecked, or changed). Different verification tasks can exist for one piece of code, and the different tasks are executable independently of each other. After each execution of a task, or each change in the source code, the status is updated automatically. For example, when a verification task is defined for the first time, its status is unchecked. When it is executed, its status changes to either checked and safe (the property holds), or checked and unsafe (the property does not hold). When the programmer updates part of the code, the status of a verification task changes to changed, and it must be executed again to confirm whether the property holds.

The advantages of verification tasks are threefold. (1) The specification of the program can be divided into many small properties, each in its own verification task. This gives more flexibility to the engineer and allows reasonable run times. (2) Every verification task can be executed separately and independently from each other. This allows us to parallelize the verification effort and to store customized abstractions for each part of the program. (3) The maintenance of information from previous runs of the model checker enables incremental verification.

A verification task contains the following information: a name to identify the task, the type of the task (assertion checking, reachability analysis, dead code analysis, or test generation), the file containing the source code, the name of the Eclipse project that contains the file, the function name, and the verification status. The verification task manager can be used to define, execute, and review results for different verification tasks. Figure 1 shows an actual example of a verification task list and the context menu for the highlighted task.

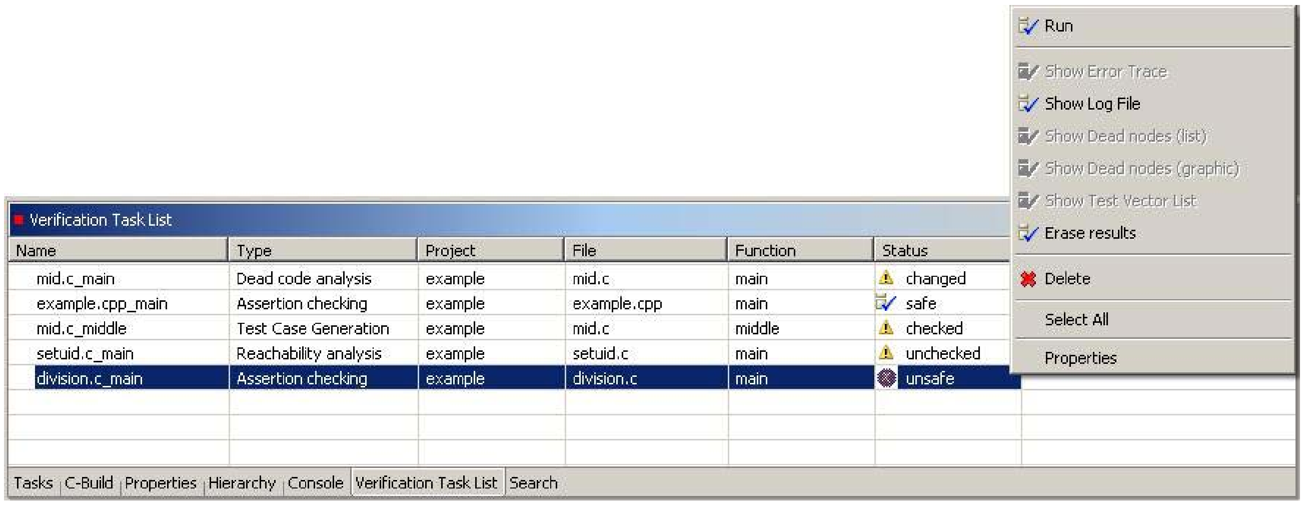
```
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>

int division(int dividend, int divisor) {
    int remainder = dividend;
    int quotient = 0;
    L2: assert(divisor > 0);
    L3: //assert(dividend > 0);
    while (divisor <= remainder) {
        remainder -= divisor;
        ++quotient;
    }
    return quotient;
}
int main(int argc, char *argv []) {
    int d1, d2;
    if (argc < 3) { exit(EXIT_FAILURE); }
    d1 = atoi(argv[1]);
    d2 = atoi(argv[2]);
    L1: if (d2 <= 0) return 2;
    printf("%d", division( d1, d2 ));
    return 0;
}
```

Figure 2. Positive integer division [7]

In the following, we explain the different kinds of verification tasks. As a running example, we use a small algorithm for positive integer division by reducing the operation to subtraction and increment [7]. Figure 2 shows the source code of the program. The function `main` transforms the command line arguments to integers and calls the division function to get the result.

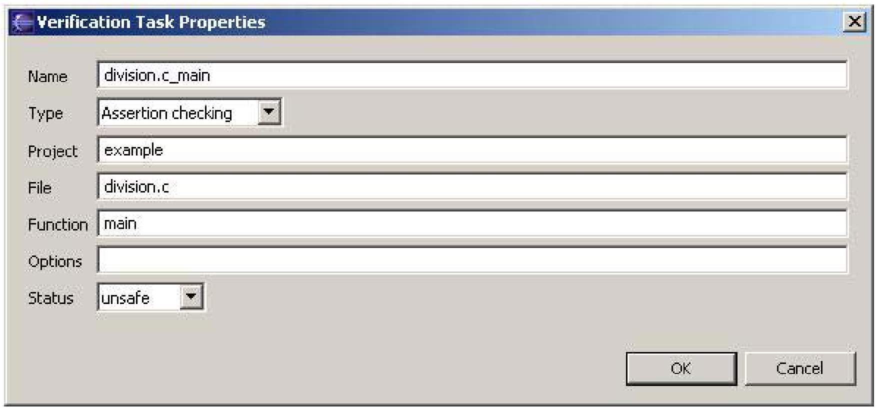
Assertion Checking. An assertion is a predicate that defines an invariant for a particular program location, i.e., the predicate has to be true every time the control flow reaches this location. Assertions are introduced into source code using the `assert(e)` statement. The assertion checking verification task takes as input a name for the task, the file containing the source code, the name of the Eclipse project, and the name of a function `foo` from which program execution is assumed to start. Figure 3 shows the property dialog of the plug-in for assertion checking. The assertion checking task checks if there is some feasible program execution to a failing assert statement. If there is no such execution, all assertions in the program hold, and the status of the task is automatically updated to safe. On the other hand, if the model checker finds a feasible execution to an assert statement such that the condition asserted does not hold, the status is set to unsafe and the model checker provides an error trace that indicates a possible path that leads to the assertion being violated. Internally, assertion checking is mapped to model checking in the following way. The assert function is written as:

```
void assert (bool e) {
  if (!b) { ERROR:    __assert(); }
}
```



Name	Type	Project	File	Function	Status
mid.c_main	Dead code analysis	example	mid.c	main	⚠ changed
example.cpp_main	Assertion checking	example	example.cpp	main	✓ safe
mid.c_middle	Test Case Generation	example	mid.c	middle	⚠ checked
setuid.c_main	Reachability analysis	example	setuid.c	main	⚠ unchecked
division.c_main	Assertion checking	example	division.c	main	● unsafe

Figure 1. Verification task list within the Eclipse plug-in



Name	division.c_main
Type	Assertion checking
Project	example
File	division.c
Function	main
Options	
Status	unsafe

Figure 3. Dialog for verification task properties

The model checker analyzes the program to check if the location labeled with `ERROR` can be reached on some execution path starting from the function `foo` defined in the verification task.

Consider an assertion verification task starting from `main` in the example. BLAST proves that assertion `divisor > 0` (at label `L2`) cannot be violated. The abstract predicates `divisor>0` and `d2>0` are saved. Suppose the user now adds the assertion `dividend >0` (at label `L3`) to ensure that the function terminates on all inputs. BLAST shows that this assertion may be violated. This is because the function `main` explicitly checks that the divisor is greater than zero, but does not check the dividend. The user can fix this bug by adding a second check in `main` similar to `L1`. This time, the model checker starts with the predicates `divisor>0` and `d2>0` and avoids work to show `divisor>0` still holds.

Reachability Analysis. The reachability question asks, for a given function of a C program and a given location (defined by a C label), whether there exists an execution of the program leading to the specified location. The input to the reachability verification task is a name for the task, the file containing the source code, the name of the Eclipse project, the name of a function `foo` from which the reachability analysis will start, and a label `L` in the program (possibly in a function different from `foo`. When executed, the status of the task is changed to unsafe if the label `L` is reachable on some execution starting from the function `foo`, and to safe if no such execution exists. In case the label is reachable, the model checker also outputs a feasible execution trace from `foo` to the label `L`.

Internally, the BLAST model checker implements an algorithm for reachability analysis that takes a reachability query and returns safe (and optionally a proof) if the location is not reachable, and unsafe (and an error trace to the location) if it is reachable. The engineer can use this analysis to prove that a bad location is not reachable, or to compute a path to the location for debugging purposes. As a

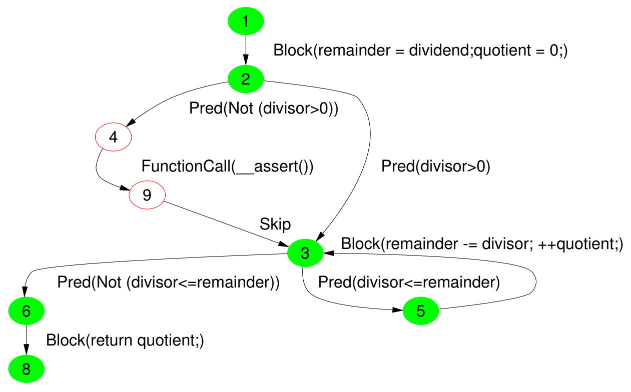


Figure 4. Control flow automaton (`division`)

positive application, reachability provides an example scenario that causes program execution to visit a particular location.

Dead Code Analysis. Dead (or unreachable) code analysis is a special form of reachability analysis that finds pieces of code that are never executed on any path. The input to the dead code analysis verification task is a name for the task, the file containing the source code, the name of the Eclipse project, and the name of a function `foo`. The output of the tool is a list of program locations that can be reached on some actual execution path starting from `foo`, and a list of program locations that are unreachable. For each reachable program location, the tool also provides an execution trace leading to that location. Internally, this is translated to a sequence of calls to the reachability engine for each node. Again, the solution is exact, i.e., there are neither false positives nor false negatives.

Figure 4 shows the control flow automaton for function `division` in our example, when we run dead code analysis starting from `main`. Filled nodes are reachable and unfilled nodes are not reachable. As expected, the call to function `__assert()` (corresponding to the violation of the assertion at `L2`) is unreachable.

Test Generation. Given a program location `L`, the test generation problem is to generate a sequence of input values such that program execution with these input values reaches `L`. The input to the test generation task is a name for the task, the name of the project, the source files, and a function `foo`. When the test generation task is executed, it generates a file with test vectors, one for each program location reachable from `foo`. Internally, the model checker runs a reachability query for each location. If the reachability query generates a trace to the location, the test generator mines this trace to generate a test case [2]. Executing the test case will cause program execution to reach the target location. By default, BLAST generates a set of test vectors for node coverage. Additionally, a predicate can be given to restrict the locations to be covered.

On executing a test generation task starting from `division`, BLAST outputs three test vectors: `(1,1)` for the path ⟨`1,2,3,5`⟩, which enters the body of the while loop, `(0,1)` for the path ⟨`1,2,3,6,8`⟩, which does not enter the while body, and `(0,0)` for the path ⟨`1,2,4`⟩, failing the assertion `divisor>0`. The numbers refer to the corresponding nodes in Figure 4. A test case for the failing assertion is included, because we considered the function `division` in isolation.

4. Conclusion

We have shown how model checking can be used by a software engineer by providing an environment where the functional power of model checking is just a click away. We demonstrated that BLAST can be integrated into an external software development environment. We hide the model checking internals behind a very simple set of use cases: create several verification tasks, execute them, take a look at the results. We provide three simple, but common and useful types of verification tasks: checking that all assertions are fulfilled, finding an example trace to a program location, and finding all pieces of code that can never be executed (dead code). Besides this, we integrated test generation into the user interface to provide simple access to example executions. We applied these tools to C programs up to 30 K lines of code in [2]. The implementation of the Eclipse plug-in and the incremental model checking engine is freely available from `http://www.eecs.berkeley.edu/~blast`.

References

[1] K. Beck. *Test Driven Development: By Example*. Addison-Wesley, 2002.

[2] D. Beyer, A. Chlipala, T. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *Proc. ICSE*. IEEE, 2004.

[3] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. CAV*, LNCS 1855, pages 154–169. Springer, 2000.

[4] E. Gamma and K. Beck. JUnit: A cook's tour. *Java Report*, 4(5):27–38, 1999.

[5] T. A. Henzinger, R. Jhala, R. Majumdar, and M. A. A. Sanvido. Extreme model checking. In *International Symposium on Verification: Theory and Practice*, LNCS 2772, pages 332–358. Springer, 2003.

[6] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. POPL*, pages 58–70. ACM, 2002.

[7] C. Hoare. Assertions: A personal perspective. *IEEE Annals of the History of Computing*, 25(2):14–25, 2003.

[8] P. Sweeney and F. Tip. A study of dead data members in C++ applications. In *Proc. PLDI*, pages 324–332, 1998.