Extreme Model Checking*

Thomas A. Henzinger, Ranjit Jhala,
Rupak Majumdar, and Marco A.A. Sanvido

Electrical Engineering and Computer Sciences
University of California, Berkeley
{tah,jhala,rupak,msanvido}@eecs.berkeley.edu
http://www.eecs.berkeley.edu/~tah/blast

To Zohar, for teaching me about logic and life. –Tom

Abstract. One of the central axioms of extreme programming is the disciplined use of regression testing during stepwise software development. Due to recent progress in software model checking, it has become possible to supplement this process with automatic checks for behavioral safety properties of programs, such as conformance with locking idioms and other programming protocols and patterns. For efficiency reasons, all checks must be incremental, i.e., they must reuse partial results from previous checks in order to avoid all unnecessary repetition of expensive verification tasks. We show that the lazy-abstraction algorithm, and its implementation in BLAST, can be extended to support the fully automatic and incremental checking of temporal safety properties during software development.

1 From Extreme Programming to Extreme Verification

Program verification has been a central problem of computer science for many years [39, 40]. The importance of the verification problem has, despite its inherent intractability, led to the development of several research areas —such as type systems, programming logics, formal semantics, static analysis, and model checking— which approach the problem from many different angles. Unfortunately, most of these methods rely heavily on user expertise, which has so far prevented their routine application in the software industry. While many general-purpose verification tools (e.g., [1, 20, 35, 47]) require sophisticated program annotations such as loop invariants and pre- and postconditions for functions, more restrictive domain-specific tools (e.g., [16, 19, 21, 45]) typically demand more modest hints, say, in the form of type annotations. Model checking is an ostensibly fully automatic method for proving systems correct. Yet also in model checking, user intervention is typically required in two places. First,

* This work was supported in part by the NSF grants CCR-9988172, CCR-0085949, and CCR-0234690, the ONR grant N00014-02-1-0671, the DARPA grant F33615-00-C-1693, and the MARCO grant 98-DT-660.

the user must supply a temporal specification, often in the form of a temporal-logic formula or Büchi automaton. Second, and more importantly, the user must supply an abstract model of the system to be checked. The model must be fine enough to satisfy the specification, yet coarse enough not to choke the model checker. Traditional model checking requires that the model be supplied as a finite state-transition graph. Finite-state graphs are a natural model for hardware systems, which, in addition, often exhibit regularities (such as symmetries in the data path) that permit model compression. Software, however, is usually infinite-state and irregular. Typically a human expert must guide the translation of the program under consideration from a high-level programming language to a finite-state graph. This may be done by having the user provide variables and predicates on these variables that are relevant to the program property that is being checked (e.g., [9, 13, 26, 33, 34, 38][1]). While some of these tools offer considerable assistance, the ultimate burden of defining a suitable program abstraction remains on the user.

Both of these shortcomings of model checking have been addressed recently in promising ways. First, the specification question can be avoided when checking for universally desirable program properties —such as memory safety and race freedom— which are a prerequisite for most functional correctness properties. Richer safety specifications of the kind that we consider in this paper —such as whether a program follows a particular device-access protocol, or driver template, or locking idiom— can be specified using simple monitor automata. In fact, even with the availability of expressive specification languages such as monitor automata, it is more rewarding to look at low-level properties of program integrity than at high-level properties of functional correctness. The reason is that the less data-dependent a property, the more likely can we find a program abstraction that is both sufficiently precise and sufficiently compact for automatic analysis.

This brings us to the second area of human intervention, namely, the definition of a suitable abstraction. Recently, a class of techniques have been developed which construct program abstractions fully automatically by counterexample-guided abstraction refinement [3, 5, 12, 36, 52]. These methods extract progressively finer finite-state models from programs. The program abstractions are sound: if the desired property holds on an abstract program, then it holds also on the concrete program. If, however, the property fails on the abstract program, then the model checker produces an abstract counterexample which, if it cannot be realized in the concrete program, is used to produce a finer abstraction that rules out the counterexample. This iterative abstract–check–refine process continues until either a concrete counterexample is found, or a program abstraction is found that satisfies the property, thus proving the program correct. Starting from a coarse seed abstraction, the whole iteration —in particular the discovery of new predicates for refining an abstraction— can be fully automated. The

[1] We limit ourselves to static model checkers. Other approaches perform symbolic execution (e.g., [10, 23], run-time checks (e.g., [27]), and dataflow analysis (e.g., [15, 18]).

approach has been introduced in program verification by [7] and applied successfully in device-driver verification. In [31], the three phases of the abstract–check–refine loop are tightly integrated to achieve performance gains. In particular, whenever an abstract counterexample cannot be concretized, the program abstraction is refined only locally, and the model checking is not repeated for the parts of the abstract state space that are not refined. The algorithm is called *lazy abstraction*, as it builds a program abstraction on-the-fly, and refines it on-demand, during the model-checking process. It has been used successfully for checking software conformance with driver templates, locking disciplines, and race freedom [29, 30].

Despite this recent progress in software model checking, the emerging tools are still relatively immature. In particular, they need to be integrated into the professional software development process. In industry, the standard way of ensuring the quality of software is through testing. Recently, *extreme programming* (XP) [8] has been advocated as a new software-engineering paradigm with a strong emphasis on testing. In XP, the stepwise (incremental) implementation of a program is alternated with extensive regression tests. XP demands that the programmer attach to every functional unit of the program a comprehensive suite of unit tests, which are run every time the program is modified. The motivation for this methodology is that the essential way in which software is created from its conception to its final implementation has not changed radically over the last few decades. While new languages and tools have simplified the process, software is still developed by stepwise refinement [55]. XP ensures that testing is carried out in an incremental, modular, and systematic way, and thus makes it easier for large teams of programmers to build and manage software systems. This programming discipline has gained considerable momentum in both academia and industry. Unit tests, however, provide only partial checks for the functional correctness of the program under development. It is therefore natural to supplement the XP approach with automatic checks for low-level temporal properties, e.g., if the program conforms with safety disciplines such as device-access protocols or locking idioms or design patterns. Such checks can be performed by software model checkers, which —unlike testing— guarantee a full coverage of all program paths. We call the software development process where regression testing is supplemented with model checking for ensuring temporal safety properties, *extreme verification* (XV).

XP assumes that software changes frequently during development and maintenance. Regression testing requires that all unit tests be run (and succeed) every time a change is committed, no matter how small. This is required even if the change made to the program is local and does not affect the success or failure of some tests. Research in regression testing has addressed this issue (e.g., [2, 6, 25, 50, 51]): regression-test selection attempts to determine automatically if a modified program, when run on some test t, would have the same observable behavior as the original program when run on t, without actually running the test t. Such incrementality is even more important in model checking, as it takes far more computation to prove a property than to run a test. If for every small modification of the program the model-checking process would have to be restarted from scratch, on the whole program, then the overall time taken for verification would likely be unacceptable to the software engineer. Hence, in any practical implementation of XV, the model-checking process must be *incremental*: the model checker must be able to check whether a program modification invalidates a proof of correctness obtained from a previous check, and if so, then the model checker must be able to start the abstraction refinement and state-space exploration from the points of the previous model at which either the old abstraction or the old proof fails. This requires the model checker to produce and maintain program abstractions which can serve as efficiently verifiable proofs that the program satisfies the desired properties. To this aim, we adapt the control-flow-based algorithms for regression-test selection: the incremental model-checking algorithm takes as input a modified program, together with an abstract model that serves as proof of correctness for a previous version of the program, and checks if the old proof still applies; if not, it generates predicates for program abstraction and initial states for state-space exploration wherever the old proof breaks. The lazy-abstraction algorithm is particularly suited for this purpose, as it maintains a nonuniform program abstraction, which uses different degrees of detail in different regions of the state space, and thus can serve as compact proof of program correctness [29]. The lazy-abstraction algorithm has been implemented in BLAST, the Berkeley Lazy Abstraction Software verification Tool [32]. In order to support the XV paradigm, we have extended BLAST to maintain all necessary data structures not only during the abstract–check–refine loop of a single verification pass, but from one verification pass to the next during the software development process.

The paper is organized as follows. In Section 2, we give an introduction to software model checking and proof generation using BLAST. In Section 3, we present the XV design flow and the extension of BLAST that permits incremental software verification. In Section 4, we illustrate the methodology and tool by applying it to the stepwise development of a simple Microsoft Windows device driver.

2 Software Verification with BLAST

2.1 The lazy-abstraction algorithm

BLAST, the Berkeley Lazy Abstraction Software verification Tool, is a model checker for C programs which builds and verifies program abstractions against a temporal safety property. At a high level, the BLAST algorithm implements the following loop for counterexample-guided abstraction refinement [7]:

Step 1 (abstraction) A finite set of predicates over the program variables is chosen, and an abstract model of the program is built automatically as a finite or push-down automaton whose states represent truth assignments for the chosen predicates.

Step 2 (verification) The abstract model is checked automatically against the specified property. If the abstract model is error-free, then so is the original program (return "program correct"); otherwise, an abstract counterexample is produced automatically, which demonstrates how the model violates the property.

Step 3 (refinement) It is checked automatically if the abstract counterexample corresponds to a concrete counterexample in the original program. If so, then a program error has been found (return "program incorrect"); otherwise, the chosen set of predicates does not contain enough information for proving program correctness and new predicates must be added. The selection of such predicates is automatic, guided by the failure to concretize the abstract counterexample.

Goto step 1.

Blast short-circuits the loop from abstraction to verification to refinement, and integrates the three steps tightly through "lazy abstraction" [31]. The integration offers significant advantages in performance by avoiding the repetition of abstraction and verification work (steps 1 and 2) from one iteration of the loop to the next.

Intuitively, lazy abstraction proceeds as follows. In step 3, we call the abstract state in which the abstract counterexample fails to have a concrete counterpart, the *pivot state*. The pivot state suggests which predicates should be used to refine the abstract model. However, instead of building an entire new abstract model, we refine the current abstract model "from the pivot state on." Since the abstract model may contain loops, such refinement *on-demand* may, of course, refine parts of the abstract model that have already been constructed, but it will do so only if necessary; that is, if the desired property can be verified without revisiting some parts of the abstract model, then our algorithm succeeds in doing so. The algorithm integrates all three steps by constructing and verifying and refining *on-the-fly* an abstract model of the program, until either the desired property is established or a concrete counterexample is found. The abstract model is not on a global set of predicates but one whose predicates change from state to state. Moreover, from the abstract reachability tree constructed by Blast, invariants that are sufficient to prove the verified property can be mined, and a formal, deductive proof can be constructed [29]. A beta version of Blast is available at www.eecs.berkeley.edu/˜tah/blast.

2.2 A device-handler example

Consider the program in Figure 1(a) and the device-access protocol specified by the *monitor automaton* of Figure 2. The monitor uses a global variable *status*, which has the value *WORK* when the device is ready for I/O operations, *WAIT* when it is waiting to be stopped, and *STOP* when it is stopped. Initially, the driver is in stopped mode. The call to `startDevice()` initializes the driver and puts it into work mode. While in work mode, the driver can process I/O requests from the program (through the function call `ioOperation()`), or take a request

```
1:    startDevice();
      b = false;
      do {
2:       if (*) {
3:          ioOperation();
         } else {
4:          b = requestStop();
         }
5:    } while (!b);
6:    stopDevice();
```

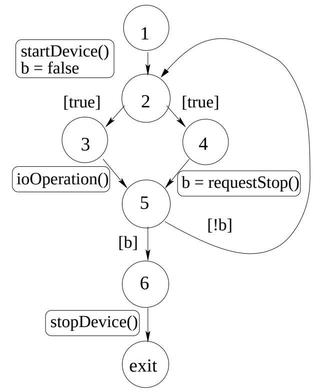

Fig. 1. (a) The program `Example` and (b) its CFA.

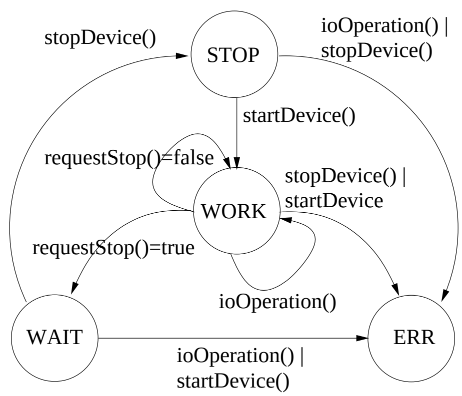

Fig. 2. The device-access specification.

to stop (through the function call `requestStop()`). The call to `requestStop()` may or may not succeed. If the call succeeds (in this case, `requestStop()` returns `true`), the driver goes into waiting mode, from which it can be stopped safely by a call to `stopDevice()`. If the call to `requestStop()` does not succeed (in this case, `requestStop()` returns `false`), the driver remains in work mode. An error occurs if the I/O function `ioOperation()` is called when the driver is in stopped or waiting mode, or if `stopDevice()` is called when the driver is in work or stopped mode, or if `startDevice()` is called when the driver is in work or waiting mode. The monitor checks that whenever an I/O operation is performed, $status = WORK$, whenever `stopDevice()` is called, $status = WAIT$, and whenever `startDevice()` is called, $status = STOP$. If any of these conditions is violated, then the monitor enters an error state. The C code of the program and the monitor (written in Blast's specification language) are given

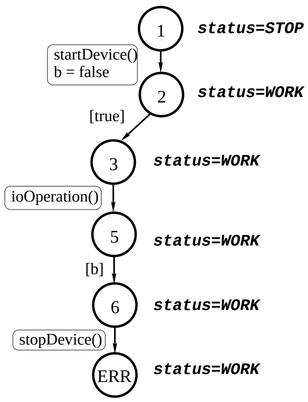

Fig. 3. Forward search.

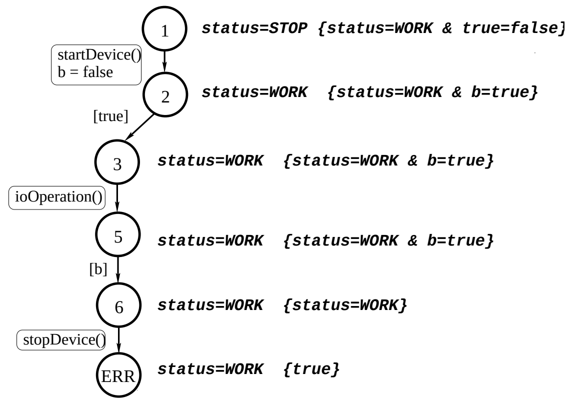

Fig. 4. Backward counterexample analysis.

as input to BLAST. They are then combined, by BLAST, into a single C program with a special error label. Safety checking is so reduced to checking whether the error label of a program is reachable.

Internally, BLAST represents programs as *control flow automata* (CFA). A CFA is a directed graph whose vertices correspond to control points of the program (begins and ends of basic blocks), and whose edges correspond to program operations. An edge is labeled either by a *basic block* of instructions, which are executed along that edge, or by an *assume predicate*, which represents a condition that must hold for the edge to be taken. For ease of exposition, we also allow atomic function calls on the CFA edges. In the actual implementation, these function calls are inlined automatically during model checking.[2] Figure 1(b) shows the CFA for the program `Example`. The edges labeled with boxes represent basic blocks; those labeled with $[\cdot]$ represent assume predicates. We model the call to `requestStop()` as nondeterministically returning either `true` or `false` in an atomic step. The condition of the `if (*)` statement is not modeled. We assume that either branch can be taken; hence both outgoing edges are labeled with the assume predicate `[true]`.

Lazy abstraction The lazy-abstraction algorithm is composed of two phases. In the forward-search phase, BLAST builds an *abstract reachability tree*, which represents a portion of the reachable, abstract state space of the program. Each node of the tree is labeled by a vertex of the CFA and a formula, called the *reachable region*, constructed as a boolean combination of a finite set of *abstraction predicates*. The edges of the tree correspond to edges of the CFA and are labeled by the corresponding basic blocks and assume predicates. Each path in the tree

[2] The current implementation of BLAST does not handle recursive function calls, but this is being remedied.

corresponds to a path in the CFA. The reachable region of a node describes the reachable states of the program in terms of the abstraction predicates, assuming execution follows the sequence of instructions labeling the edges from the root of the tree to the node. If we find that an error node is reachable in the tree, then we proceed to the second phase, which checks if the error is real or results from the abstraction being too coarse (i.e., if we lost too much information by restricting ourselves to a particular set of abstraction predicates). In the latter case, BLAST asks a theorem prover to suggest additional abstraction predicates, which rule out that particular spurious counterexample. By iterating the two phases of forward search and backward counterexample analysis, different portions of the abstract reachability tree will use different sets of abstraction predicates.

We now illustrate how the lazy-abstraction algorithm performs on the program `Example`. We start by considering the three abstraction predicates $status = WORK$, $status = STOP$, and $status = WAIT$ derived from the specification.

Forward search Consider Figure 1. We construct an abstract reachability tree in depth-first order. The root corresponds to the entry vertex of the program (location 1), and is labeled by its precondition $status = STOP$. The reachable region of each node in the tree is obtained from that of the parent by an overapproximate successor computation with respect to the set of abstraction predicates. The computed successor is a Cartesian abstraction of the predicate abstraction of the successor [24, 31]. More precisely, for a reachable region r in disjunctive normal form, and an edge label `op`, we do the following. For every disjunct d of r we construct a corresponding disjunct $post(d, \texttt{op})$. To do this we check, for every abstraction predicate p that is currently being tracked, if $d \Rightarrow wp(p, \texttt{op})$ and if $d \Rightarrow wp(\neg p, \texttt{op})$, where $wp(p, \texttt{op})$ is the weakest precondition of p with respect to `op` [17]. If the first is true, we add p as a conjunct to $post(d, \texttt{op})$; if the second is true, we add $\neg p$ as a conjunct to $post(d, \texttt{op})$. If neither is true, we do not add a literal corresponding to p to $post(d, \texttt{op})$ (note that both queries can never be true). For instance, we compute the successor of the reachable region $status = STOP$ with respect to the block `startDevice(); b = false` as $(status = WORK) \wedge \neg(status = STOP) \wedge \neg(status = WAIT)$. In the example, the forward search finds a path to an error node, namely, $1 \to 2 \to 3 \to 5 \to 6$.

Backward counterexample analysis We check if the path from the root to the error node is a genuine counterexample or results from the abstraction being too coarse. To do this, we symbolically simulate the error path backward in the original program. As we go backward from the error node, we try to find the first node in the abstract reachability tree where the abstract path fails to have a concrete counterpart. If we find such a *pivot node*, then we conclude that the counterexample is spurious and refine the abstraction from the pivot node on by adding new abstraction predicates that rule out the counterexample. If the analysis goes back to the root without finding a pivot node, then we have found a real error in the program.

Figure 4 shows the result of this phase. In the figure, for each node, the formula within curly braces, called the *bad region*, represents the set of states in the reachable region that can go from the corresponding control location to an error. Formally, the bad region of a node is the intersection of the reachable region of the node with the weakest precondition of *true* with respect to the sequence of instructions that label the path in the abstract reachability tree from the node to the error node. It is computed inductively, starting backward from the error node, which has the bad region *true*. Note that unlike the forward-search phase, the backward-counterexample analysis is precise: we track all predicates obtained along the abstract error path. In Figure 4, we find that the bad region at location 1 is *false*, which implies that the counterexample is spurious. Location 1 becomes the pivot node.

We use the result of the counterexample analysis to refine the abstraction: we add a minimal set of new abstraction predicates that are enough to show infeasibility of the abstract error path. The predicates that appear in the proof of infeasibility are produced by a proof-generating theorem. In the example, the reachable region at the pivot node is $status = STOP$, and the abstract error path is $\sigma = \texttt{startDevice}(); \texttt{b} = \texttt{false} \cdot [\texttt{true}] \cdot \texttt{ioOperation()} \cdot [\texttt{b}] \cdot \texttt{stopDevice}()$. Infeasibility of this path is equivalent to the unsatisfiability of the formula $wp(true, \sigma) \wedge (status = STOP)$, where $wp(true, \sigma)$ is the syntactic predecessor of *true* along the path σ. To maintain the syntactic form of the predicates obtained along the path, all substitutions in weakest preconditions must be maintained explicitly. In particular, to compute $wp(p, \texttt{x} = \texttt{e})$, instead of returning the classical $p[e/x]$, we introduce a fresh primed variable x' and return $(x' = e) \wedge p[x'/x]$ (note that the variable x' acts as a Skolem constant). In the example, the precondition of the abstract error path is:

$$b' = false \wedge status' = WORK \wedge b' = true \wedge status' = WORK.$$

To this, we conjoin the region $status = STOP$ from the pivot node and submit the resulting formula to the proof-generating theorem prover. The prover says that the proof of unsatisfiability of the conjunction involves the two predicates $b' = true$ and $b' = false$.

Search with new predicates We proceed with another forward-search phase, starting from the pivot node, but this time we track the abstraction predicate $b = true$ in addition to $status = STOP$, $status = WORK$, and $status = WAIT$. The resulting abstract reachability tree is shown in Figure 5. Notice that we can stop the search at the leaf labeled 2: as the states satisfying the reachable region $(status = WORK) \wedge (b = false)$ are a subset of those satisfying the reachable region at node 2, the subtree constructed from the leaf would be included in the subtree of node 2. In the new abstract reachability tree, no error node is reachable. Hence we conclude that the program `Example` satisfies the device-access specification.

Proof generation While the lazy-abstraction algorithm verifies the temporal safety property specified by a monitor automaton, it does not provide a *deductive*

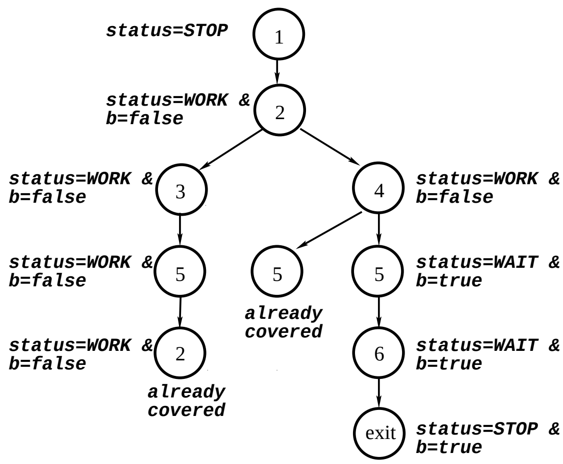

Fig. 5. An abstract reachability tree.

proof. In particular, there is no apparent way to distinguish a correct analysis of the system from a buggy implementation of the algorithm. Therefore, we would like to provide an easily checkable and compact certificate that the program meets its specification.

Lazy abstraction can be used naturally and efficiently to construct deductive correctness proofs for temporal safety properties. This is because the data structures produced by lazy abstraction supply the program annotations required for proof construction and provide a decomposition of the proof leading to a small correctness certificate. In particular, using abstraction predicates only where necessary keeps the proof small, and using the model checker to guide the proof generation eliminates the need for backtracking, e.g., in the proof of disjunctions. To certify that a program satisfies its specification, we use a standard *temporal safety rule* from deductive verification [41]: given a transition system, if we can find a set I of states such that (1) I contains all initial states, (2) I contains no error states, and (3) I is closed under successor states, then the system cannot reach an error state from an initial state. If (1)–(3) are satisfied, then I is called an *invariant* set. In our example, the temporal safety rule reduces to supplying for each vertex q of the CFA an invariant formula $I(q)$ such that

1. $(status = STOP) \Rightarrow I(1)$;
2. $I(\texttt{ERR}) = false$;
3. for each pair q and q' of CFA vertices with an edge labeled `op` between them, $sp(I(q), \texttt{op}) \Rightarrow I(q')$, where sp is the strongest-postcondition operator [17].

Thus, to provide a proof of correctness, it suffices to supply a location invariant $I(q)$ for each vertex q of the CFA, and proofs that the supplied formulas meet the above three requirements.

The location invariants can be mined from the abstract reachability tree produced by the lazy-abstraction algorithm. In particular, the invariant for q is the disjunction of all reachable regions that label the nodes in the tree which correspond to q. For instance, $I(5)$ is $(\mathit{status} = \mathit{WORK} \wedge b = \mathit{false}) \vee (\mathit{status} = \mathit{WAIT} \wedge b = \mathit{true})$. It is easy to check that $(\mathit{status} = \mathit{STOP}) \Rightarrow I(1)$, because the root of the abstract reachability tree is labeled by the precondition of the program $(\mathit{status} = \mathit{STOP})$. Also, as there is no node labeled `ERR` in the tree, we get the second requirement by definition. The interesting part is checking the third requirement, that for each edge $q \xrightarrow{\mathtt{op}} q'$ of the CFA, $sp(I(q), \mathtt{op}) \Rightarrow I(q')$. Consider the edge $5 \xrightarrow{[\mathtt{b}]} 6$. We need to show that

$$sp((\mathit{status} = \mathit{WORK} \wedge \neg b) \vee (\mathit{status} = \mathit{WAIT} \wedge b), [\mathtt{b}]) \\ \Rightarrow (\mathit{status} = \mathit{WAIT} \wedge b).$$

By distributing sp over $\vee$, we are left with the proof obligation $((\mathit{status} = \mathit{WAIT} \wedge b) \vee \mathit{false}) \Rightarrow (\mathit{status} = \mathit{WAIT} \wedge b)$. To prove this, notice that the disjuncts on the left can be broken down into subformulas obtained from the reachable regions of individual nodes. Hence we can show the implication by matching each subformula with the appropriate successor on the right. In this way we obtain the two proof obligations $(\mathit{status} = \mathit{WAIT} \wedge b) \Rightarrow (\mathit{status} = \mathit{WORK} \wedge b)$ and $\mathit{false} \Rightarrow (\mathit{status} = \mathit{WAIT} \wedge b)$. Exactly these obligations were generated in the forward-search phase when computing abstract successors. Each obligation is discharged, and the whole proof assembled, using a proof-generating theorem prover. The above process generates, completely automatically, a deductive proof of correctness for the temporal safety property [29]. Such a proof can then be used as proof certificate in proof-carrying code [43].

3 Incremental Verification with Blast

3.1 The extreme model-checking paradigm

At an abstract level, the extreme-programming (XP) methodology can be viewed as a sequence of programs and test suites (P^i, T^i), where each program P^{i+1} is a modified version of P^i, and each T^{i+1} contains the previous test suite T^i, that is, $T^i \subseteq T^{i+1}$. After the implementation of P^i all tests in T^i are performed on the program. If P^i complies to all tests, then the program is "committed"; this represents a stable version in the development. Any modifications or additions will be implemented in the new, refined version P^{i+1}, and the corresponding test suite T^{i+1} will be expanded from the test suite T^i. In traditional applications of XP, the unit tests are simple functional tests, checking desired input-output behaviors of functions, often written in stylized languages such as JML [11, 37].

In extreme verification (XV), we generalize the XP methodology as follows. In addition to the program P^i and functional test suite T^i, we have at each stage a set Φ^i of temporal safety property specifications that must be satisfied by the program P^i. The behavioral requirements and protocol specifications in Φ^i complement the functional tests in T^i. Like the functional test suites, the set

Φ^i of property specifications never decreases —that is, $\Phi^i \subseteq \Phi^{i+1}$— but usually increases as more properties are added as the program is being developed. After each step in the development, all specifications in Φ^i are checked by a model checker on the program P^i. If all tests in T^i as well as all properties in Φ^i pass, then the program is committed. To support the XV paradigm, we envision a software model checker to be added to popular unit testing frameworks such as JUnit [22]. Alternatively, the temporal safety properties in Φ^i can also be used as test cases, and like functional unit tests, be checked by simulation or run-time monitoring [28]. For example, [53] propose FAUSEL, a language that allows specifying linear temporal logic (LTL) and metrical temporal logic (MTL) expressions that are monitored during program execution. Similarly, [27] describes JavaExplorer, a tool that monitors running Java programs for LTL specifications. However, to obtain complete coverage, it is often desirable to verify temporal properties using a model checker. Also, motivated by functional test-case generation (e.g., [54]), there have been some recent attempts to automatically generate temporal specifications [4].

The XP methodology can be very expensive for testing as every (local) change in the program requires every test to be rerun. To reduce the cost of testing, incremental regression test techniques (also called *regression test selection*) have been proposed. These techniques essentially check the control flow graph of a modified program against the control flow graph of the original program, and compute path coverage information for each test, to decide which tests need to be rerun on the modified program. By applying such techniques, the number of test cases in T^i that are repeated on P^{i+1} is reduced. Even more so, XV can be extremely expensive if the model checker has to reverify each property every time a small change is made in the program. We now show that a model checker which can produce proofs, together with an analysis similar to incremental regression testing, can be used to make the verification process incremental as well. In particular, it is possible to completely skip the model checking of a property $\varphi \in \Phi^i$ on the modified program P^{i+1} if the proof that P^i satisfies φ can be reused. Moreover, even if this is not possible, the state-space exploration for checking φ on P^{i+1} can sometimes be limited to those parts of the state space that have changed from P^i to P^{i+1}.

3.2 Conformance of a program with an abstract reachability tree

Suppose that we are given a program P, a safety property φ, and an abstract reachability tree for P which has been constructed by BLAST in order to verify that P satisfies φ. Further suppose that P is modified to P'. We wish to check if the abstract reachability tree for P can be reused to prove that also P' satisfies φ, in order to avoid repeating the model-checking process if possible. As a simple example, when writing concurrent programs, programmers often write the synchronization skeleton first, and check that it conforms to the locking protocol, and then add the data manipulation. However, the data manipulation does not affect the correctness of the locking protocol, because the variables touched by the control (synchronization code) and the data manipulation are usually disjoint. Thus, a proof of correctness for the locking protocol should still be valid after the data manipulation statements have been added, and an incremental model checker should be able to verify this quickly. In this case, we say that the refined program *conforms* to the previously constructed abstract reachability tree.

We have implemented in BLAST an algorithm for conformance checking, which constructs the CFA for the new program P' and compares it top-down with the abstract reachability tree constructed for the old program P with respect to the safety property φ . As long as no discrepancy is found, BLAST reuses the same tree to generate a proof of correctness. If, however, an edge in the abstract reachability tree cannot be replicated in the new program, or an edge is missing, then BLAST restarts the lazy-abstraction algorithm from the corresponding node. In the following, we describe this algorithm more precisely. We start with a few definitions in order to specify our notion of conformance formally.

A *control flow automaton* (CFA) C is a tuple $\langle Q, q_0, X, \texttt{Ops}, \rightarrow \rangle$, where Q is a finite set of control locations, q_0 is the initial control location, X is a set of typed variables, $\texttt{Ops}$ is a set of operations on X, and $\rightarrow \subseteq (Q \times \texttt{Ops} \times Q)$ is a finite set of edges labeled with operations. An edge $(q, \texttt{op}, q')$ is also denoted $q \xrightarrow{\texttt{op}} q'$. The set $\texttt{Ops}$ of operations contains (1) *basic blocks* of instructions, that is, finite sequences of assignments `lval = exp`, where `lval` is an lvalue from X (i.e., a variable, structure field, or pointer dereference), and `exp` is an arithmetic expression over X; and (2) *assume predicates* [`p`], where `p` is a boolean expression over X (arithmetic comparison or pointer equality), representing a condition that must be true for the edge to be taken. For ease of exposition we describe our method only for CFAs without function calls; the method can be extended to handle function calls in a standard way (and function calls are handled by the BLAST implementation).

The set $\mathcal{V}_X$ of *(data) valuations* over the variables X contains the type-preserving functions from X to values. A *region* is a set of data valuations; let $\mathcal{R}$ be the set of regions. We use quantifier-free first-order formulas over some fixed set of relation and function symbols to represent regions. The semantics of operations is given in terms of the strongest-postcondition operator [17]: $sp(r, \texttt{op})$ of a formula r with respect to an operation $\texttt{op}$ is the strongest formula whose truth holds after $\texttt{op}$ terminates when executed in a valuation that satisfies r. For a formula $r \in \mathcal{R}$ and operation $\texttt{op} \in \texttt{Ops}$, the formula $sp(r, \texttt{op}) \in \mathcal{R}$ is syntactically computable; in particular, after skolemization, the strongest postcondition is again a quantifier-free formula. A location $q \in Q$ is *reachable from a precondition* $Pre \in \mathcal{R}$ if there is a finite path $q_0 \xrightarrow{\texttt{op}_1} q_1 \xrightarrow{\texttt{op}_2} \cdots \xrightarrow{\texttt{op}_n} q_n$ in the CFA and a sequence of formulas $r_i \in \mathcal{R}$, for $0 \leq i \leq n$, such that $q_n = q$, $r_0 = Pre$, $r_n \not\Leftrightarrow false$, and $sp(r_i, \texttt{op}_{i+1}) = r_{i+1}$ for all $0 \leq i < n$. The witnessing path is called a *feasible* path from (q_0, Pre) to q. We write $sp(r, \texttt{op}_1 \texttt{op}_2 \ldots \texttt{op}_n)$ to denote $sp(\ldots sp(sp(r, \texttt{op}_1), \texttt{op}_2) \ldots)$.

Let $T = (V, E, \mathtt{n}_0)$ be a (finite) rooted tree, where each node $\mathtt{n} \in V$ is labeled by a pair $(q, r) \in Q \times \mathcal{R}$, each edge $e \in E$ is labeled by an operation $\mathtt{op} \in \mathtt{Ops}$, and $\mathtt{n}_0 \in V$ is the root node. We write $\mathtt{n} : (q, r)$ if node n is labeled by control location q and region r; in that case, we say that r is the *reachable region* of n and write $reg(n) = r$. If there is an edge from $\mathtt{n} : (q, r)$ to $\mathtt{n}' : (q', r')$ labeled by $\mathtt{op}$, then node $\mathtt{n}'$ is an $(\mathtt{op}, q')$-*child* of node $\mathtt{n}$. The labeled tree T is an *abstract reachability tree* for the CFA C if (1) the root $n_0 : (q_0, r_0)$ is labeled by the initial location q_0 of the CFA; (2) each internal node $n : (q, r)$ has an $(\mathtt{op}, q')$-child $n' : (q', r')$ for each edge $q \xrightarrow{\mathtt{op}} q'$ of C; (3) if $n' : (q', r')$ is an $(\mathtt{op}, q')$-child of $n : (q, r)$, then $sp(r, \mathtt{op}) \Rightarrow r'$; and (4) for each leaf node $n : (q, r)$, either q has no successors in C, or there are internal nodes $n_1 : (q, r_1), \ldots, n_k : (q, r_k)$ such that $r \Rightarrow (r_1 \vee \ldots \vee r_k)$. In the latter case, we say that n is *covered* by $n_1, \ldots, n_k$. Intuitively, an abstract reachability tree is a finite unfolding of the CFA whose nodes annotated with regions, and whose edges are annotated with corresponding operations from the CFA. For a set $F \subseteq V$ of leaf nodes of T, the pair (T, F) is a *partial reachability tree* for C if conditions (1), (2), and (3) hold, and (4$'$) for each leaf node $n : (q, r)$, either $n \in F$, or q has no successors in C, or there are internal nodes $n_1 : (q, r_1), \ldots, n_k : (q, r_k)$ such that $r \Rightarrow (r_1 \vee \ldots \vee r_k)$. Intuitively, a partial reachability tree is a prefix of an abstract reachability tree, where the nodes in F have not yet been explored. Of course, if $F = \emptyset$, then (T, F) is an abstract reachability tree.

An *error function* $\mathcal{E} \colon Q \to \{0, 1\}$ identifies a set $\mathcal{E}^{-1}(1) \subseteq Q$ of error locations of the CFA C. Every safety property φ of a program P can be compiled into an error function on the CFA C that results from composing the CFA of P with a monitor automaton for φ [29]. The automaton C is *safe* with respect to the precondition $Pre \in \mathcal{R}$ and error function $\mathcal{E}$ if no location q with $\mathcal{E}(q) = 1$ is reachable from Pre. An abstract reachability tree T for C is *safe* with respect to the precondition $Pre \in \mathcal{R}$ and error function $\mathcal{E}$ if (1) the root has the form $n : (q_0, Pre)$ and (2) for all nodes of the form $n : (q, r)$ with $\mathcal{E}(q) = 1$, we have $r \Leftrightarrow$ *false*. A partial reachability tree (T, F) is *safe* with respect to the precondition Pre and error function $\mathcal{E}$ if condition (1) holds, and (2$'$) for all nodes of the form $n : (q, r)$ with $\mathcal{E}(q) = 1$, either $n \in F$ or $r \Leftrightarrow$ *false*. A safe abstract reachability tree witnesses the correctness of the CFA for the precondition Pre and error function $\mathcal{E}$, and the reachable regions that label its nodes provide program invariants. The following theorem makes this precise.

Theorem 1. [31] *Let C be a CFA, Pre a precondition, and $\mathcal{E}$ an error function for C. If there exists an abstract reachability tree for C which is safe with respect to Pre and $\mathcal{E}$, then C is safe with respect to the precondition Pre and error function $\mathcal{E}$*

Figure 5 shows a safe abstract reachability tree that witnesses the correctness of the program `Example` for the precondition $status = STOP$ and the device-access specification from Figure 2. From a safe abstract reachability tree we can construct a proof of correctness. For this purpose, it is convenient to augment abstract reachability trees with additional bookkeeping information: (1) for each

```
1:     startDevice();
       b = false;
       numIo = 0;
       do {
2:        if (*) {
3:           ioOperation();
             numIo ++;
          } else {
4:           b = requestStop();
             dataBuf = 0;
          }
5:     } while (!b);
6:     stopDevice();
```

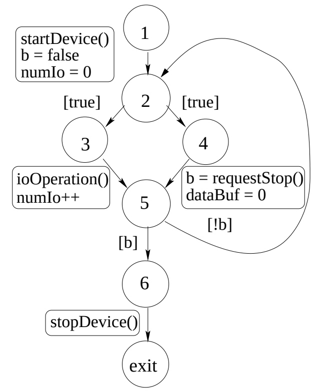

Fig. 6. (a) The modified program `Example1` and (b) its CFA.

node of the tree, BLAST provides the set of abstraction predicates that define the abstract state at this node (these are the predicates from which the reachable region is formed), and (2) for each leaf node, BLAST provides the set of internal nodes that cover this node.

We now define the conformance relation between a given abstract reachability tree and a new program. Informally, the conformance relation specifies a sufficient condition under which a proof of a particular safety specification for a program continues to hold for a modified version of the program. Similar ideas have been used in translation validation [14, 44, 48, 49] to prove that certain optimization transformations behave correctly. Let $T' = (V', E', n'_0)$ and $T = (V, E, n_0)$ be two abstract reachability trees. A relation $\preceq \subseteq V' \times V$ between the nodes of T' and T is a *simulation relation* [42] if $n' \preceq n$ implies (1) $reg(n') \Rightarrow reg(n)$ and (2) for each child $\bar{n}'$ of n', there is a child $\bar{n}$ of n such that $\bar{n}' \preceq \bar{n}$. We write $T' \preceq T$ if if there is a simulation relation $\preceq$ such that $n'_0 \preceq n_0$. A CFA C with the precondition $Pre \in \mathcal{R}$ and error function $\mathcal{E}$ *safely conforms* with the abstract reachability tree T if there exists an abstract reachability tree T' for C such that (1) T' is safe with respect to Pre and $\mathcal{E}$, and (2) $T' \preceq T$. The following is a stronger version of Theorem 1.

Theorem 2. *If a CFA C with precondition Pre and error function $\mathcal{E}$ safely conforms with some abstract reachability tree T, then C is safe with respect to Pre and $\mathcal{E}$.*

Figure 6 shows a refinement `Example1` of the program `Example`, which keeps track of the number of I/O requests served and resets a data buffer after calling `requestStop()`. Since the instructions that have been added do not affect the abstraction predicates, the CFA for `Example1` safely conforms with the abstract reachability tree of Figure 5 with respect to the precondition $status = STOP$ and the device-access specification.

Algorithm 1 Function $\mathsf{checkConformance}(C, Pre, \mathcal{E}, T)$

Require: a CFA $C = \langle Q, q_0', X, \mathtt{Ops}, \rightarrow \rangle$ with precondition Pre and error function $\mathcal{E}$, and an abstract reachability tree $T = (V, E, n_0)$.

```
create the root n'_0 : (q'_0, Pre) for the reachability tree T'
add (n_0, n'_0) to WorkList
initialize Frontier to ∅, and Map to ∅
while there are pairs in WorkList do
  choose a pair (n : (q, r), n' : (q', r')) from WorkList, and remove it from WorkList
  if 𝓔(q') = 1 then
    add n' to Frontier
  end if
  if r' ⇏ r then
    add n' to Frontier
  end if
  if n is a leaf node in T then
    if q' has no successors in C then break
    let n_1 : (q, r_1), ..., n_k : (q, r_k) be nodes in T such that r ⇒ ⋁_{i=1...k} r_i
    if Map(n_i) is undefined for some i = 1, ..., k then
      add n' to Frontier
    else
      let n'_i : (q'_i, r'_i) = Map(n_i) for each i = 1, ..., k
      if r ⇏ ⋁_{i=1...k} {s_i | s_i = r'_i if q'_i = q', and s_i = false otherwise} then
        add n' to Frontier
      end if
    end if
  else
    if number of edges out of n and q' are the same then
      let f = matchEdges(n, n')
      if f ≠ ∅ then
        for each pair ⟨(n : (q, r), op, n̄ : (q̄, r̄)), (q', op', q̄')⟩ ∈ f do
          add a child n̄' : (q̄', r̄) to n' in T'
          add (n̄, n̄') to WorkList
          add (n̄, n̄') to Map
        end for
      else
        add n' to Frontier
      end if
    else
      add n' to Frontier
    end if
  end if
end while
return (T', Frontier)
```

3.3 Checking conformance

We now describe an algorithm for checking conformance of a CFA C with an abstract reachability tree T. In order to check conformance, we have to construct a safe abstract reachability tree for C which is simulated by T. For efficiency reasons, the algorithm is imprecise. If the algorithm declares that C safely conforms with T, then this is indeed the case, but it may happen that the algorithm cannot prove that the CFA safely conforms with the abstract reachability tree even when there is a conformance relation. If the algorithm does not succeed in proving conformance, then it returns a set *Frontier* of nodes of T, which are the nodes from which the lazy-abstraction algorithm can be restarted in order to produce an abstract reachability tree for C from a prefix of T.

The input to checkConformance (Algorithm 1) is a CFA C, a precondition *Pre* and an error function $\mathcal{E}$ for C, and an abstract reachability tree T. The algorithm walks in lock-step through C and T and tries to construct an abstract reachability tree T' for C such that T' is both safe with respect to *Pre* and $\mathcal{E}$, and $T' \preceq T$. The output of the algorithm is a partial reachability tree $(T', \mathit{Frontier})$ for C which is safe with respect to *Pre* and $\mathcal{E}$. Intuitively, the set *Frontier* denotes the set of nodes at which the simulation of T' by T breaks. If *Frontier* is empty, then T' is safe with respect to *Pre* and $\mathcal{E}$, and simulated by T; that is, C safely conforms with T.

The algorithm maintains two lists: *WorkList* contains pairs (n, n') of nodes, n of T and n' of T', for which it needs to be checked if n' is simulated by n; and *Frontier* contains the nodes of T' which are not found to be simulated by corresponding nodes of T, and from which model checking must continue. For efficiency reasons, we sacrifice precision and do not implement an exact simulation check. The function *Map* keeps track of the simulation relation we construct, by mapping nodes of T to nodes of T' such that $\mathit{Map}(n)$ is simulated by n. The fundamental step in our implementation is to establish a correspondence between the edges coming out of the current node of T and the edges coming out of the current CFA location. This is done by the function matchEdges, which takes as input a node $n\!:\!(q, r)$ of T and a node $n'\!:\!(q', r')$ of T', such that the number of children of n in T is equal to the number of successors of q' in C. It then tries to match up the operations on the edges as follows: edge $(n\!:\!(q, r), \mathsf{op}, \bar{n}\!:\!(\bar{q}, \bar{r}))$ of T is matched to edge $(q', \mathsf{op}', \bar{q}')$ of C if $sp(r', \mathsf{op}') \Rightarrow \bar{r}$. If such a pairing cannot be found, then matchEdges returns the empty set.

Our implementation of matchEdges (Algorithm 2) again trades off precision against efficiency. Since the CFA is obtained from a high-level language like C, it has the following property: each CFA location either has one successor with the corresponding edge labeled by a basic block, or it has exactly two successors, and the corresponding edges are labeled by assume predicates. Moreover, in the latter case the two conditions in the assume predicates are complementary. For nodes with two successors (the assume-predicate case), we require that the conditionals are syntactically identical. In other words, we expect the control flow structure of the original and revised programs to look identical. For nodes with one successor (the basic-block case), we match edges in the following

Algorithm 2 Function $\mathsf{matchEdges}(n\!:\!(q,r), n'\!:\!(q',r'))$

initialize f to $\emptyset$
case n and q' have each two outgoing edges:
  let $(n\!:\!(q,r), [\mathrm{p}], n_1\!:\!(q_1,r_1))$ and $(n\!:\!(q,r), [!\mathrm{p}], n_2\!:\!(q_2,r_2))$ be the edges out of n
  if $(q', [\mathrm{p}], q'_1)$ and $(q', [!\mathrm{p}], q'_2)$ are the two edges out of q' **then**
    add $\langle (n, [\mathrm{p}], n_1), (q', [\mathrm{p}], q'_1)\rangle$ and $\langle (n, [!\mathrm{p}], n_2), (q', [!\mathrm{p}], q'_2)\rangle$ to f
  end if
case n and q' have each one outgoing edge:
  let $(n\!:\!(q,r), \mathtt{op}, \bar{n}\!:\!(\bar{q},\bar{r}))$ be the edge out of n
  let $(q', \mathtt{op}', \bar{q}')$ be the edge out of q'
  if $wp(p, \mathtt{op}) = wp(p, \mathtt{op}')$ for each abstraction predicate p of n **then**
    add $\langle (n, \mathtt{op}, \bar{n}), (q', \mathtt{op}', \bar{q}')\rangle$ to f
  end if
return f

way. Suppose that the two basic blocks are $\mathtt{op}$ and $\mathtt{op}'$, respectively. For each abstraction predicate p of the current node of the abstract reachability tree T, we compute the weakest preconditions $wp(p, \mathtt{op})$ and $wp(p, \mathtt{op}')$, and then check that they are syntactically identical. (In fact, we can assume that the weakest preconditions for $\mathtt{op}$ are part of T, so these need not be recomputed.) The benefit of using weakest preconditions is that the weakest-precondition operator is robust with respect to syntactic details such as permutation of independent instructions, renaming of auxiliary variables, and substitutions. For example, consider the edge $(4, \mathtt{b} = \mathtt{requestStop}();, 5)$ in the CFA of Figure 1(b), and the edge $(4, \mathtt{b} = \mathtt{requestStop}(); \mathtt{dataBuf} = 0;, 5)$ of the refined program `Example1` (discussed in the next section), where some data manipulation has been added. In this example, $\mathtt{op}$ is `b = requestStop();` and $\mathtt{op}'$ is `b = requestStop(); dataBuf = 0`. Then the weakest preconditions $wp(status = WAIT, \mathtt{op})$ and $wp(status = WAIT, \mathtt{op}')$ are identical, and similarly, $wp(b, \mathtt{op})$ and $wp(b, \mathtt{op}')$ are identical. In other words, there is no effect of the new assignment on the behavior of the program with respect to the predicates of interest.

Theorem 3. *If* $\mathsf{checkConformance}(C, Pre, \mathcal{E}, T) = (T', \emptyset)$*, then the CFA C with the precondition Pre and error function $\mathcal{E}$ safely conforms with the abstract reachability tree T. If* $\mathsf{checkConformance}(C, Pre, \mathcal{E}, T) = (T', F)$*, then (T', F) is a partial reachability tree for C which is safe with respect to Pre and $\mathcal{E}$.*

In particular, in case a program modification does not interfere with the abstraction predicates, the old abstract reachability tree T still encodes a valid proof for the modified program C. However, even if we cannot prove conformance, we need only to model check the new program from the abstract states in the set *Frontier* returned by the checkConformance algorithm. Thus, unlike translation validation, incremental model checking can "fall back" to model checking from the points of disagreement. We have found that in practice this reduces the model-checking time significantly.

Incremental model checking is implemented in BLAST as follows. The algorithm consists of two main parts: the checkConformance function, which checks a modified program against an old proof, and the lazyModelChecker function, which does the model checking. The lazyModelChecker algorithm tries to construct an abstract reachability tree for a CFA which is safe with respect to a precondition and an error function, as outlined in Section 2. More precisely, lazyModelChecker takes as input a CFA C with initial location q_0, a precondition *Pre* and an error function $\mathcal{E}$ for C, and a partial reachability tree (T, F) for C which is safe with respect to *Pre* and $\mathcal{E}$. If the algorithm terminates, it returns either the pair ("safe", T'), where T' is an abstract reachability tree for C which is safe with respect to *Pre* and $\mathcal{E}$, or the pair ("unsafe", σ), where σ is an error path of C, that is, a path from (q_0, Pre) to some node q with $\mathcal{E}(q) = 1$. In the former case, C is safe with respect to *Pre* and $\mathcal{E}$, and a proof can be extracted from T' (see Section 2).

In the nonincremental version of BLAST, the partial reachability tree passed to the function lazyModelChecker is $(T'_0, \{n'_0 : (q_0, Pre)\})$, where T'_0 is the tree consisting of the single root node $n'_0 : (q_0, Pre)$ (and no edges) [31]. In the incremental version, BLAST first calls the checkConformance algorithm, and then passes the partial reachability tree $(T', Frontier)$ returned by checkConformance to the lazyModelChecker. The following makes this precise:

Algorithm $\textsc{Blast}(C, Pre, \mathcal{E}, T)$ {
  Require: a CFA C with precondition *Pre* and error function $\mathcal{E}$, and
    an abstract reachability tree T.
  $(T', Frontier) = \mathsf{checkConformance}(C, Pre, \mathcal{E}, T)$
  return $\mathsf{LazyModelChecker}(C, Pre, \mathcal{E}, (T', Frontier))$
}

4 Example: Windows Driver Development

To give a better understanding of incremental software development using extreme model checking, we will illustrate the development of `KbFilter`, a simple Windows 2000 device driver.[3] A *device driver* is a piece of code that provides an interface between the operating system and hardware devices. The Windows Driver Model [46] prescribes how device drivers are organized within the operating system.

For each hardware device, there are several device drivers (at different levels of abstraction) organized in a driver stack. The I/O manager and device drivers use a data structure called the I/O Request Packet (IRP) to manage the details of I/O operations. The IRP is created by some kernel mode component to perform an operation on a device, or to send a query or instruction to the driver. The I/O manager sends this IRP to one or more of the subroutines that the driver interface exports. Each subroutine performs some work on the IRP and returns control to the I/O manager. Eventually, some subroutine completes the IRP,

[3] Available with the Microsoft Windows DDK.

```
NTSTATUS KbFilter_PnP(PDEVICE_OBJECT DeviceObject, PIRP Irp)
{
    PDEVICE_EXTENSION           devExt;
    PIO_STACK_LOCATION          irpStack;
    NTSTATUS                    status = STATUS_SUCCESS;
    KIRQL                       oldIrql;
    KEVENT                      event;

    PAGED_CODE();

    devExt = (PDEVICE_EXTENSION) DeviceObject->DeviceExtension;
    irpStack = IoGetCurrentIrpStackLocation(Irp);

    switch  (irpStack->MinorFunction) {
    case  IRP_MN_START_DEVICE:
    case  IRP_MN_SURPRISE_REMOVAL:
    case  IRP_MN_REMOVE_DEVICE:
           /* Other cases removed */
    default :
        IoSkipCurrentIrpStackLocation(Irp);
        status = IoCallDriver(devExt->TopOfStack, Irp);
        break ;
    }

    return  status;
}
```

Fig. 7. First version of the KbFilter_PnP function.

and then the I/O manager destroys the IRP and reports the ending status back to the kernel component that originated the request. Usually the request is sent to the topmost driver in the driver stack for the device, and can percolate down the stack to lower drivers. At each level, the driver decides what to do with the request. The driver can pass the request to lower drivers, or process the request, or partially process the request and then pass it to a lower driver.

We concentrate on one particular property which each driver in the driver stack has to satisfy with respect to the way I/O request packets are handled [46]. The property says that if an IRP is passed to a lower driver for processing, then the return status of the driver must be the same as the return status of the lower driver.

To simplify the exposition we focus on one core procedure of the device driver, namely, the function KbFilter_PnP. This procedure is called whenever a plug-and-play operation on the device is required by the Plug-and-Play (PnP) Manager of the operating system. The PnP requests instruct the driver when and how to configure or deconfigure itself and the hardware. A PnP request can designate about twenty minor functions; most of these are simply passed down the stack by the KbFilter driver. For this driver, the three interesting requests are:

- IRP_MN_START_DEVICE, which configures and initializes the driver;
- IRP_MN_SURPRISE_REMOVAL, which notes the fact that the physical hardware has been unexpectedly removed; and
- IRP_MN_REMOVE_DEVICE, which shuts down and removes the device.

The procedure KbFilter_PnP has two parameters: DeviceObject is a pointer to the device object, and IRP is a pointer to the I/O request packet. The return status of each driver routine is a value of type NTSTATUS, which encodes the operation status, such as STATUS_SUCCES or STATUS_PENDING.

The procedure KbFilter_PnP might be implemented through stepwise refinement in the following four steps.

First step In the first incarnation of KbFilter_PnP (Figure 7), devExt points to the DeviceExtension field of the device object which holds local variables used by the KbFilter driver. The function IoGetCurrentIrpStackLocation returns a pointer to the I/O stack location for the driver. The driver calls IoGetCurrentIrpStackLocation with each IRP it receives in order to get the parameters for the current request (in particular, the MinorFunction parameter, which determines the particular request).

The case statement has a separate case for each possible PnP request. Notice that we have not implemented any of the functionality in the case statements; these will be written only in later refinements. In fact, the default implementation simply passes the IRP down to the next driver on the stack and returns whatever status the lower driver returns. For this first version of the program, Blast checks successfully that the IRP completion property is satisfied.

Second step Now we are ready to continue the development of the device driver by adding the implementation of one of the case statements. We begin with the case IRP_MN_START_DEVICE, which is needed to start the device (Figure 8). It calls the following function KbFilter_Complete:

```
NTSTATUS KbFilter_Complete(PDEVICE_OBJECT DeviceObject,
                           PIRP Irp, PVOID Context) {
       PKEVENT event ;
       event = (PKEVENT )Context;
       /* Set the event on which KbFilter_PnP can be blocked */
       KeSetEvent(event, 0, FALSE);
       return  (NTSTATUS )STATUS_MORE_PROCESSING_REQUIRED;
}
```

The refined program is handed again to Blast. The incremental algorithm checkConformance starts with the abstract reachability tree from the first step and checks if it is still valid. This effort fails at the case

```
case  IRP_MN_START_DEVICE: {
    IoCopyCurrentIrpStackLocationToNext(Irp);
    KeInitializeEvent(&event, NotificationEvent, FALSE);

    IoSetCompletionRoutine(Irp,
                    (PIO_COMPLETION_ROUTINE) KbFilter_Complete,
                    &event,
                    TRUE,
                    TRUE,
                    TRUE);

    status = IoCallDriver(devExt->TopOfStack, Irp);

    if  (STATUS_PENDING == status) {
        KeWaitForSingleObject(
                    &event,
                    Executive,
                    KernelMode,
                    FALSE,
                    NULL);
    }

    if  (NT_SUCCESS(status) && NT_SUCCESS(Irp->IoStatus.Status)) {
        devExt->Started = TRUE;
        devExt->Removed = FALSE;
        devExt->SurpriseRemoved = FALSE;
    }

    Irp->IoStatus.Status = status;
    Irp->IoStatus.Information = 0;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);

    break ;
}
```

Fig. 8. Second version: refining the case `IRP_MN_START_DEVICE`.

`IRP_MN_START_DEVICE`, which has been added, and the model checker starts exploring the state space from this branch of the case statement (the other branches still point to unchanged code, and checkConformance realizes that these branches need not be checked again). At this point, BLAST detects an error in the program. The reason is that the status returned by the lower driver is not overwritten by the subsequent call to `KeWaitForSingleObject`. We fix this by changing the call `KeWaitForSingleObject(&event, Executive, KernelMode, FALSE, NULL)` to `status = KeWaitForSingleObject(&event, Executive, KernelMode, FALSE, NULL)`. A new check by BLAST of the previously buggy branch reports that there is no more error.

```
case  IRP_MN_SURPRISE_REMOVAL:
     devExt->SurpriseRemoved = TRUE;

     /* Remove code here */

     IoSkipCurrentIrpStackLocation(Irp);
     status = IoCallDriver(devExt->TopOfStack, Irp);
     break ;
```

Fig. 9. Third version: refining the case `IRP_MN_SURPRISE_REMOVAL`.

Third step Now we proceed to refine another case statement, namely, the one that implements the surprise removal of hardware (Figure 9). In response to this request, a device driver must disable all registered interfaces, then release I/O resources, and finally pass down the request to the lower driver. Our implementation is correct: BLAST detects no error in the program. Again, the model checker is run only on the case that has been added. Our driver is particularly simple, in that no I/O resources need to be released. For a more complicated driver, this step may be written in two stages. First, the programmer writes the above skeleton and verifies that the IRP completion property holds. Second, the programmer adds the routines that release resources where it says "Remove code here," and the resulting program is revalidated. Typically, the proof of correctness from the first step will continue to hold after the second step.

Fourth step Finally we code the functionality of the routine that removes the device (Figure 10). In this version of the program BLAST finds again an error. The reason is that the procedure returns always `STATUS_SUCCESS`, independently of the result of the function `IofCallDriver`. To correct the problem we have to consider the status returned by the `IofCallDriver` as shown in Figure 11. This version of the driver again passes the check by BLAST. Of course, at this stage the programmer will write more tests for correct functionality. For example, the removal code must call `IoDetachDevice` to balance the call to `IoAttachDeviceToDeviceStack` from `AddDevice`, which added the device to the device stack, and `IoDeleteDevice` to balance the call to `IoCreateDevice` from `AddDevice`. These tests can be coded as additional safety monitors and checked again with BLAST.

Acknowledgment. We thank Gregoire Sutre for helping to implement BLAST.

References

1. J.-R. Abrial. *The B Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

```
case  IRP_MN_REMOVE_DEVICE:

     devExt->Removed = TRUE;

     /* Remove code here */

     IoSkipCurrentIrpStackLocation(Irp);
     IoCallDriver(devExt->TopOfStack, Irp);

     IoDetachDevice(devExt->TopOfStack);
     IoDeleteDevice(DeviceObject);

     status = STATUS_SUCCESS;
     break ;
```

Fig. 10. Fourth version: refining the case `IRP_MN_REMOVE_DEVICE`.

2. H. Agrawal, J.R. Horgan, E.W. Krauser, and S.A. London. Incremental regression testing. In *ICSM 93: International Conference on Software Maintenance*, pages 348–357. IEEE, 1993.
3. R. Alur, A. Itai, R.P. Kurshan, and M. Yannakakis. Timing verification by successive approximation. *Information and Computation*, 118(1):142–157, 1995.
4. G. Ammons, R. Bodik, and J. Larus. Mining specifications. In *POPL 02: Principles of Programming Languages*, pages 4–16. ACM, 2002.
5. F. Balarin and A.L. Sangiovanni-Vincentelli. An iterative approach to language containment. In *CAV 93: Computer-Aided Verification*, Lecture Notes in Computer Science 697, pages 29–40. Springer-Verlag, 1993.
6. T. Ball. On the limit of control-flow analysis for regression test selection. In *ISSTA 98: International Symposium on Software Testing and Analysis*, pages 134–142. ACM, 1998.
7. T. Ball and S.K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pages 1–3. ACM, 2002.
8. K. Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 1999.
9. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation, Complexity, Analysis, Transformation: Essays Dedicated to Neil D. Jones*, Lecture Notes in Computer Science 2566, pages 85–108. Springer-Verlag, 2002.
10. W.R. Bush, J.D. Pincus, and D.J. Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 30(7):775–802, 2000.
11. Y. Cheon and G.T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *ECOOP 02: European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science 2374, pages 231–255. Springer-Verlag, 2002.
12. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 00: Computer-Aided Verification*, Lecture Notes in Computer Science 1855, pages 154–169. Springer-Verlag, 2000.

```
case  IRP_MN_REMOVE_DEVICE:

     devExt->Removed = TRUE;

     /* Remove code here */

     IoSkipCurrentIrpStackLocation(Irp);
     status = IofCallDriver(devExt->TopOfStack, Irp);

     IoDetachDevice(devExt->TopOfStack);
     IoDeleteDevice(DeviceObject);

     break ;
```

Fig. 11. Corrected version of Figure 10.

13. J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, R.S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *ICSE 00: International Conference on Software Engineering*, pages 439–448. ACM/IEEE, 2000.
14. D.W. Currie, A.J. Hu, S. Rajan, and M. Fujita. Automatic formal verification of DSP software. In *DAC 00: Design Automation Conference*, pages 130–135. ACM/IEEE, 2000.
15. M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI 02: Programming Language Design and Implementation*, pages 57–68. ACM, 2002.
16. R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI 01: Programming Language Design and Implementation*, pages 59–69. ACM, 2001.
17. E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
18. D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI 00: Operating System Design and Implementation*, pages 1–16. Usenix Association, 2000.
19. D. Evans and D. Larochelle. Improving security using extensible light-weight static analysis. *IEEE Software*, 19(1):42–51, 2002.
20. C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI 02: Programming Language Design and Implementation*, pages 234–245. ACM, 2002.
21. J.S. Foster, T. Terauchi, and A. Aiken. Flow-Sensitive Type Qualifiers. In *PLDI 02: Programming Language Design and Implementation*, pages 1–12. ACM, 2002.
22. E. Gamma and K. Beck. JUnit: A cook's tour. *Java Report*, 4(5):27–38, 1999.
23. P. Godefroid. Model checking for programming languages using VeriSoft. In *POPL 97: Principles of Programming Languages*, pages 174–186. ACM, 1997.
24. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV 97: Computer-Aided Verification*, Lecture Notes in Computer Science 1254, pages 72–83. Springer-Verlag, 1997.
25. M.J. Harrold, J. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. Spoon, and A. Gujarathi. Regression test selection for java software. In *OOPSLA 01:*

Object-Oriented Programming, Systems, Languages, and Applications, pages 312–326. ACM, 2001.

26. K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Software Tools for Technology Transfer*, 2(4):72–84, 2000.
27. K. Havelund and G. Rosu. Monitoring Java programs with Java PathExplorer. *Electronic Notes in Theoretical Computer Science*, 55(2), 2001.
28. K. Havelund and G. Rosu, editors. *Workshop on Run-Time Verification*, volume 70(4) of *Electronic Notes in Theoretical Computer Science*, 2002.
29. T.A. Henzinger, R. Jhala, R. Majumdar, G.C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *CAV 02: Computer-Aided Verification*, Lecture Notes in Computer Science 2404, pages 526–538. Springer-Verlag, 2002.
30. T.A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *CAV 03: Computer-Aided Verification*, Lecture Notes in Computer Science 2725, pages 262–274. Springer-Verlag, 2003.
31. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL 02: Principles of Programming Languages*, pages 58–70. ACM, 2002.
32. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *SPIN 03: SPIN Workshop*, Lecture Notes in Computer Science 2648, pages 235–239. Springer-Verlag, 2003.
33. G.J. Holzmann. The Spin model checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
34. G.J. Holzmann. Logic verification of ANSI-C code with Spin. In *SPIN 00: Model Checking of Software*, Lecture Notes in Computer Science 1885, pages 131–147. Springer-Verlag, 2000.
35. M. Kaufmann, P. Manolios, and J.S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.
36. R.P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, 1994.
37. G.T. Leavens, K.R.M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: Notations and tools supporting detailed design in Java. In *OOPSLA 00: Object-Oriented Programming, Systems, Languages, and Applications*, pages 105–106. ACM, 2000.
38. T. Lev-Ami and S. Sagiv. TVLA: A system for implementing static analyses. In *SAS 02: Static Analysis Symposium*, Lecture Notes in Computer Science 2280, pages 280–301. Springer-Verlag, 2000.
39. Z. Manna. The correctness of programs. *Journal of Computer and Systems Sciences*, 3(2):119–127, 1969.
40. Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1972.
41. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.
42. R. Milner. An algebraic definition of simulation between programs. In *Second International Joint Conference on Artificial Intelligence*, pages 481–489. The British Computer Society, 1971.
43. G.C. Necula. Proof-carrying code. In *POPL 97: Principles of Programming Languages*, pages 106–119. ACM, 1997.
44. G.C. Necula. Translation validation for an optimizing compiler. In *PLDI 00: Programming Languages Design and Implementation*, pages 83–95. ACM, 2001.
45. R. O'Callahan and D. Jackson. Lackwit: A program-understanding tool based on type inference. In *ICSE 97: International Conference on Software Engineering*, pages 338–348. ACM/IEEE, 1997.
46. W. Oney. *Programming the Microsoft Windows Driver Model*. Microsoft Press, 1999.

47. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In *CAV 96: Computer-Aided Verification*, Lecture Notes in Computer Science 1102, pages 411–414. Springer-Verlag, 1996.
48. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS 98: Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science 1384, pages 151–166. Springer-Verlag, 1998.
49. M. Rinard and D. Marinov. Credible compilation. Technical Report MIT-LCS-TR-776, MIT, 1999.
50. G. Rothermel and M. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996.
51. G. Rothermel and M. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, 1997.
52. H. Saidi. Model-checking-guided abstraction and analysis. In *SAS 00: Static-Analysis Symposium*, Lecture Notes in Computer Science 1824, pages 377–396. Springer-Verlag, 2000.
53. M.A.A. Sanvido, W. Schaufelberger, and V. Cechticky. Testing embedded control systems using hardware-in-the-loop simulation and temporal logic. In *15th IFAC World Congress on Automatic Control*, 2002.
54. D. Stotts, M. Lindsey, and A. Antley. An informal formal method for systematic JUnit test-case generation. In *XP/Agile Universe 2002*, Lecture Notes in Computer Science 2418, pages 132–143. Springer-Verlag, 2002.
55. N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, 1971.