# The Blast Query Language for Software Verification*

Dirk Beyer[1] Adam J. Chlipala[2] Thomas A. Henzinger[1,2]
Ranjit Jhala[2] Rupak Majumdar[3]

[1] EPFL, Switzerland
[2] University of California, Berkeley
[3] University of California, Los Angeles

**Abstract.** Blast is an automatic verification tool for checking temporal safety properties of C programs. Blast is based on lazy predicate abstraction driven by interpolation-based predicate discovery. In this paper, we present the Blast specification language. The language specifies program properties at two levels of precision. At the lower level, monitor automata are used to specify temporal safety properties of program executions (traces). At the higher level, relational reachability queries over program locations are used to combine lower-level trace properties. The two-level specification language can be used to break down a verification task into several independent calls of the model-checking engine. In this way, each call to the model checker may have to analyze only part of the program, or part of the specification, and may thus succeed in a reduction of the number of predicates needed for the analysis. In addition, the two-level specification language provides a means for structuring and maintaining specifications.

## 1 Introduction

Blast, the Berkeley Lazy Abstraction Software verification Tool, is a fully automatic engine for software model checking [11]. Blast uses counterexample-guided predicate abstraction refinement to verify temporal safety properties of C programs. The tool incrementally constructs an abstract reachability tree (ART) whose nodes are labeled with program locations and truth values of predicates. If a path that violates the desired safety property is found in the ART, but is not a feasible path of the program, then new predicate information is added to the ART in order to rule out the spurious error path. The new predicate information is added on-demand and locally, following the twin paradigms of *lazy abstraction* [11] and *interpolation-based predicate discovery* [8]. The procedure stops when either a genuine error path is found, or the current ART represents a proof of program correctness [9].

In this paper we present the Blast input language for specifying program-verification tasks. The Blast specification language consists of two levels. On

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

the lower level, *observer automata* are defined to monitor the program execution and decide whether a safety property is violated. Observer automata can be infinite-state and can track the program state, including the values of program variables and type-state information associated with individual data objects. On the higher level, *relational queries* over program locations are defined which may specify both structural program properties (e.g., the existence of a syntactic path between two locations) and semantic program properties (e.g., the existence of a feasible path between two locations). The evaluation of a semantic property invokes the BLAST model-checking engine. A semantic property may also refer to an observer automaton, thus combining the two levels of specification.

Consider the following example. If we change the definition of a variable in a program, we have to review all subsequent read accesses to that variable. Using static analysis we can find all statements that use the variable, but the resulting set is often imprecise (e.g., it may include dead code) because of the path-insensitive nature of the analysis. Model checking can avoid this imprecision. In addition, using an observer automaton, we can ensure that we compute only those statements subsequent to the variable definition which (1) use the variable and (2) are not preceded by a redefinition of the variable. The two specification levels allow the natural expression of such a query: on the higher level, we specify the location-based reachability property between definition and use locations, and at the lower level, we specify the desired temporal property by a monitor automaton that watches out for redefinitions of the variable. The resulting query asks the model checker for the set of definition-use pairs of program locations that are connected by feasible paths along which no redefinitions occur.

The BLAST specification language provides a convenient user interface: it keeps specifications separate from the program code and makes the model checker easier to use for non-experts, as no manual program annotations with specification code (such as assertions) are required. On one hand it is useful to orthogonalize concerns by separating program properties from the source code, and keeping them separated during development, in order to make it easier to understand and maintain both the program and the specification [13]. On the other hand it is preferable for the programmer to specify program properties in a language that is similar to the programming language. We therefore use as much as possible C-like syntax in the specification language. The states of observer automata are defined using C type and variable declarations, and the automaton transitions are defined using C code. The query language is an imperative scripting language whose expressions specify first-order relational constraints on program locations.

The two-level specification structure provides two further benefits. First, such structured specifications are easy to read, compose, and revise. The relational query language allows the programmer to treat the program as a database of facts, which can be queried by the analysis engine. Moreover, individual parts of a composite query can be checked incrementally when the program changes, as in regression testing [10]. Second, the high-level query language can be used to break down a verification task into several independent model-checking problems, each checking a low-level trace property. Since the number of predicates in the ART is the main source of complexity for the model-checking procedure, the decomposition of a verification task into several independent subtasks, each involving only a part of the program and/or a part of the specification, can greatly contribute to the scalability of the verification process [14,17]. A simple instance of this occurs if a specification consists of a conjunction of several properties that can be model checked independently. The relational query engine allows the compact definition of such proof-decomposition strategies.

For a more instructive example, suppose that we wish to check that there is no feasible path from a program location $\ell_0$ to a program location $\ell_2$, and that all syntactic paths from $\ell_0$ to $\ell_2$ go through location $\ell_1$. Then we may decompose the verification task by guessing an intermediate predicate $p_1$ and checking, independently, the following two simpler properties: (1) there is no feasible path from $\ell_0$ to $\ell_1$ such that $p_1$ is false at the end of the path (at $\ell_1$), and (2) there is no feasible path from $\ell_1$ to $\ell_2$ such that $p_1$ is true at the beginning of the path (at $\ell_1$). Both proof obligations (1) and (2) may be much simpler to model check, with fewer predicates needed, than the original verification task. Moreover, each of the two proof obligations can be specified as a reachability query over locations together with an observer automaton that specifies the final (resp. initial) condition $p_1$.

The paper is organized as follows. In Section 2, we define the (lower-level) language for specifying trace properties through observer automata. In Section 3, we define the (higher-level) language for specifying location properties through relational queries. In Section 4, we give several sample specifications, and in Section 5, we briefly describe how the query processing is implemented in Blast.

**Related work.** Automata are often used to specify temporal safety properties, because they provide a convenient, succinct notation and are often easier to understand than formulas of temporal logic. For example, SLIC [2] specifications are used in the SLAM project [1] to generate C code for model checking. However, SLIC does not support type-state properties and is limited to the specification of interfaces, because it monitors only function calls and returns. Metal [7] and MOPS [4] allow more general pattern languages, but the (finite) state of the automaton must be explicitly enumerated. Temporal-logic specifications, often enriched with syntactic sugar ("patterns"), are used in Bandera [5] and Feaver [12]. Type-state verification [16] is an important concept for ensuring the reliability of software, but the generally used assumption in this field is to consider all paths of a program as feasible. Relational algebra has been applied to analyze the structure of large programs [3] and in dynamic analysis [6]. Also the decomposition of verification tasks has been recognized as a key issue and strategy-definition languages have been proposed [14,17]. However, the use of a relational query language to group queries and decompose proof obligations in a model-checking environment seems novel.

## 2 Trace Properties: Observer Automata

Trace properties are expressed using *observer automata*. These provide a way to specify temporal safety properties of C programs based on syntactic pattern matching of C code. An observer automaton consists of a collection of syntactic patterns that, when matched against the current execution point of the observed program, trigger transitions in the observer. Rather than being limited to a finite number of states, the observer may have global variables of any C type, and it may track type-state information associated with the program variables. The observer transitions are also specified in C syntax; they may read program variables and both read and write observer variables.

### 2.1 Syntax

The definition of an observer automaton consists of a set of declarations, each defining an observer variable, a type state, an initial condition, a final condition, or an event. Figure 1 gives the grammar for specifying observer automata.

```
Observer:     DeclSeq
DeclSeq:      Declaration | DeclSeq Declaration
Declaration:  'GLOBAL' CVarDef
            | 'SHADOW' CTypeName '{' CFieldSeq '}'
            | 'INITIAL' '{' CExpression '}'
            | 'FINAL' '{' CExpression '}'
            | 'EVENT' '{'
                 Temporal
                 'PATTERN' '{' Pattern '}'
                 Assertion
                 Action
              '}'
Temporal:     'BEFORE' | 'AFTER' | empty
Pattern:      ParamCStmt | ParamCStmt 'AT' LocDesc
Assertion:    'ASSERT' '{' CExpression '}' | empty
Action:       'ACTION' '{' CStatementSeq '}' | empty
```

**Fig. 1.** The grammar for the observer specification language.

**Observer variables.** The control state of an observer automaton consists of a global part and a per-object part. The global part of the observer state is determined by a set of typed, global observer variables. Each observer variable may have any C type, and is declared following the keyword `GLOBAL`, where the nonterminal `CVarDef` stands for any C variable declaration. For example, in the case of a specification that restricts the number of calls to a certain function, an observer variable `numCalls` of type `int` might be used to track the number of calls made: "`GLOBAL int numCalls;`".

**Type states.** The keyword `SHADOW` allows the programmer to define additional control state of the observer automaton on a per-object basis. For this purpose, each distinct C type `CTypeName` which occurs in the program may have a type state declared in the specification. The type-state information is declared by the nonterminal `CFieldSeq`, which stands for any sequence of field definitions for a C structure. These fields are then added as type state to every program variable of type `CTypeName`. For example, in the case that the program uses a type `stack` to declare stacks, the following type state may be used to track the size of each program variable of type `stack`: "`SHADOW stack {int size;}`". Then, during verification, the type `stack` is replaced by a new structure type with the additional field `size`.

**Initial and final conditions.** The initial states of the observer automaton are defined by initial conditions. Each initial condition is declared following the keyword `INITIAL` as a boolean expression. The nonterminal `CExpression` is a (side-effect free) C expression that may refer to observer variables, but also to global program variables and associated type-state information. This allows us to encode a precondition when starting the verification process. We call the conjunction of all initial conditions the *precondition* of the observer automaton. If no initial condition is specified, then the precondition is true. Final conditions are just like initial conditions, and their conjunction is called the *postcondition* of the observer automaton. The postcondition is used to check the program and observer states after any finite trace.

**Events.** The transitions of the observer automaton are defined by events. Each event observes all program steps and, if a match is obtained, specifies how the state of the observer (global variables and type states) changes. The keyword `EVENT` is followed by up to four parts: a temporal qualifier, a pattern, an assertion, and an action. Intuitively, at each point in the program execution, the observer checks the current program statement (i.e., AST node) being executed against the pattern of each event. If more than one pattern matches, then BLAST declares the specification to be invalid for the given program. If only one pattern matches, then the corresponding assertion is checked. If the assertion is violated, then the observer rejects the trace; otherwise it executes the corresponding action. The `Temporal` qualifier is either `BEFORE` or `AFTER`. It specifies whether the observer transition is executed before or after the source-code AST node that matches the pattern. If a temporal qualifier is omitted, it is assumed to be `BEFORE`.

The keyword `PATTERN` is followed by a statement that is matched against the program source code. The pattern is defined by the nonterminal `ParamCStmt`, followed by an optional program-location descriptor. A pattern is either a C assignment statement or a C function call that involves side-effect free expressions. The pattern may refer to variables named `$`$i$, for $i \geq 1$, which are matched against arbitrary C expressions in the program. Each such pattern variable may appear at most once in a pattern. There is also a pattern variable named `$?`, which plays the role of a wild-card. It may occur multiple times in a pattern, and different occurrences may match the empty string, a C expression, or an arbitrary number of actual parameters in a function call. The location descriptor

`LocDesc` is either a C label, or a string that concatenates the source file name with a line number; e.g., the string "`file_19`" refers to line number `19` of the source file `file`. If a location descriptor is specified, then the pattern is matched only against program locations that match the descriptor.

The keyword `ASSERT` is followed by a program invariant that must hold every time the corresponding pattern matches. Here, `CExpression` is a boolean condition expressed as a C expression that may refer to global program variables, observer variables, numbered pattern variables $\$i$ that occur in the corresponding pattern (which may match local program variables), and type-state information associated with any of these. Numbered pattern variables in an assertion refer to the expressions with which they are unified by the pattern matching that triggers the event. If an assertion is omitted, it is assumed to be always true. If during program execution the pattern of an event matches, but the current state violates the assertion, then the observer is said to *reject* the trace.

The keyword `ACTION` is followed by a sequence of C statements that are executed every time the corresponding pattern matches. The code in `CStatementSeq` has the following restrictions. First, as in assertions, the only read variables are global program variables, observer variables, numbered pattern variables, and associated type states. Second, the action code may write only to observer variables and to type-state information. In particular, an observer action must not change the program state. If an action is omitted, it is assumed to be the empty sequence of statements.

*Example 1.* [**Locking**] Consider the informal specification that a program must acquire and release locks in strict alternation. The observer automaton defined in Figure 2(a) specifies the correct usage of locking functions. An observer variable `locked` is created to track the status of the (only) global lock. Simple events match calls to the relevant functions. The event for `init` initializes the observer variable to `0`, indicating that the lock is not in use. The other two events ensure that the lock is not in use with each call of the function `lock`, and is in use with each call of `unlock`. When these assertions succeed, the observer variable is updated and execution proceeds; when an assertion fails, an error is signaled. The wild-cards `$?`'s match either a variable to which the result of a function call is assigned, or the absence of such an assignment, thus making the patterns cover all possible calls to the functions `lock` and `unlock`.

Figure 2(b) shows the same specification, but now the program contains several locks, and the functions `lock` and `unlock` take a lock as a parameter. A lock is assumed to be an object of type `lock_t`. The observer introduces a type state `locked` with each lock of the program, and checks and updates the type state whenever one of the functions `init`, `lock`, and `unlock` is called. □

```
GLOBAL int locked;

EVENT {
  PATTERN { $? = init(); }
  ACTION { locked = 0; }
}
EVENT {
  PATTERN { $? = lock(); }
  ASSERT { locked == 0 }
  ACTION { locked = 1; }
}
EVENT {
  PATTERN { $? = unlock(); }
  ASSERT { locked == 1 }
  ACTION { locked = 0; }
}
```

```
SHADOW lock_t { int locked; }

EVENT {
        PATTERN { init($1); }
        ACTION { $1->locked = 0; }
}
EVENT {
        PATTERN { lock($1); }
        ASSERT { $1->locked == 0 }
        ACTION { $1->locked = 1; }
}
EVENT {
        PATTERN { unlock($1); }
        ASSERT { $1->locked == 1 }
        ACTION { $1->locked = 0; }
}
```

**Fig. 2.** (a) Specification for a global lock. (b) Specification for several locks.

### 2.2 Semantics

The semantics of a trace property is given by running the observer automaton in parallel with the program. The automaton accepts a program trace if along the trace, every time an observer event matches, the corresponding assertion is true, and moreover, if the trace is finite, then the values of the variables at the end of the trace satisfy the postcondition of the observer automaton. Dually, the automaton rejects the trace if either some assertion or the postcondition fails. We give the semantics of the composition of a program and an observer automaton by instrumenting the program source code with C code for the observer variable, type-state, and event declarations, i.e., the original program is transformed into a new program by a sequence of simple steps. This transformation is performed statically on the program before starting the model-checking engine on the transformed program.

Syntactic pattern matching on literal C code must deal with code structuring issues. Blast performs pattern matching against a simplified subset of C statements. In our implementation, first a sound transformation from C programs to the simplified statement language is performed by CIL [15]. These simplified statements consist only of variable assignments and function calls involving side-effect free expressions. Second, Blast's instrumentation of the program with the observer is performed on the simplified language. Third, Blast performs model checking on the instrumented program, which is represented by a graph whose nodes correspond to program locations and whose edges are labeled with sequences of simplified statements [11]. The model checker takes as input also the pre- and postconditions of the observer automaton, as described in the next section.

**Instrumenting programs.** In the following we define the program instrumentation with the observer automaton by describing a transformation rule for each construct of the observer specification.

*Observer variables.* Declarations of observer variables are inserted as global declarations in the C program.

*Type state.* The type-state fields declared by the observer automaton are inserted into the declarations section of the C program by replacing the original declarations of the corresponding types. The actual transformation depends on the "shadowed" type. If the shadowed type is abstract, then the type itself is replaced. In this case, the fields of the original type cannot be analyzed, because their definition is not available. If the shadowed type is not abstract, then the original type becomes one field of the new type, with the other fields holding the type-state information. All type accesses in the program are modified accordingly. For the example in Figure 2(b), first assume that `lock_t` is an abstract type for the locking data structure. Then the type-state declaration

```
SHADOW lock_t { int locked; }
```

is transformed and inserted as follows in the declarations section of the program:

```
struct __shadow0__ { int locked; };
typedef struct __shadow0__ *lock_t;
```

If, on the other hand, the type `lock_t` is defined as

```
struct lock_t_struct { int lock_info; }
typedef struct lock_t_struct *lock_t;
```

then the type name is changed to `lock_t_orig` and the type-state declaration is transformed and inserted as follows:

```
struct __shadow0__ { lock_t_orig shadowed; int locked; };
typedef struct __shadow0__ *lock_t;
```

Additionally, in this case, for every instance `mylock` of type `lock_t`, each occurrence of `mylock->lock_info` is replaced by `mylock->shadowed->lock_info`.

*Events.* For every event declaration of the observer automaton an if-statement is generated. The condition of that if-statement is a copy of the assertion, where the pattern variables `$`$i$ are replaced by the matching C expressions. The then-branch contains a copy of the action code, again with the place holders substituted accordingly. The else-branch contains a transition to the rejecting state of the automaton. Then the original program is traversed to find every matching statement for the pattern of the event. The pattern is matched if the place holders (`$`$i$ and `$?`) in the pattern can be replaced by code fragments such that the pattern becomes identical to the examined statement. If two or more patterns match the same statement, then BLAST stops and signals that the specification is invalid (ambiguous) for the given program. As specified by the temporal qualifier `BEFORE` or `AFTER`, the generated if-statement is inserted before or after each matching program statement. Consider, for example, the second event declaration from Figure 2(a). For this event, every occurrence of the code fragment `lock();` matches the pattern, whether or not the return value is assigned to a variable (because of the wild-card `$?` on the left-hand side of the pattern). The instrumentation adds the following code before every call to `lock` in the program:

```
if (locked == 0) {
    locked = 1;
} else {
    { __reject = 1; }    // transition to rejecting state
}
```

Note that the rejecting state of the observer automaton is modeled by the implicitly defined observer variable `__reject`. This variable must not occur in the program nor in the observer declaration.

**Observer semantics.** A state of a program $P$ is a pair $(\ell, v)$ consisting of a program location $\ell$ and a memory valuation $v$. Let $\ell$ and $\ell'$ be two program locations, and let $p$ and $p'$ be two predicates over the program variables. The pair $(\ell', p')$ is *reachable in* $P$ from the pair $(\ell, p)$ if there exists an executable state sequence (finite trace) of $P$ from a state $(\ell, v)$ to a state $(\ell', v')$, for some memory valuation $v$ that satisfies $p$ and some valuation $v'$ that satisfies $p'$.

We can now define the semantics of an observer automaton $A$ over a program $P$ in terms of the traces of the instrumented program $P_A$. Let *pre* be the predicate $pre_A \wedge (\texttt{\_\_reject} = 0)$, where $pre_A$ is the precondition of $A$, and let *post* be the predicate $post_A \wedge (\texttt{\_\_reject} = 0)$, where $post_A$ is the postcondition of $A$. The location $\ell'$ is *$A$-accept-reachable in $P$* from $\ell$ if $(\ell', post)$ is reachable in $P_A$ from $(\ell, pre)$. The location $\ell'$ is *$A$-reject-reachable in $P$* from $\ell$ if $(\ell', \neg post)$ is reachable in $P_A$ from $(\ell, pre)$. Note that both accept- and reject-reachability postulate the existence of a feasible path in $P$ from $\ell$ to $\ell'$; the difference depends only on whether the observer automaton accepts or rejects. In particular, it may be that $\ell'$ is both $A$-accept- and $A$-reject-reachable in $P$ from $\ell$.

## 3 Location Properties: Relational Queries

Every observer automaton encodes a trace property. At a higher level, observer automata can be combined by *relational queries*. The queries operate on program locations and specify properties using sets and relations over program locations. The query language is an imperative scripting language that extends the predicate calculus: it provides first-order relational expressions (but no function symbols) as well as statements for variable assignment and control flow.

### 3.1 Syntax

A simple query is a sequence of statements, where each statement is either an assignment or a print statement. There are three types of variables: string, property, and relation variables. A string variable may express a program location, a

```
Statement:  PropVar ':=' '[' Observer ']' ';'
          | RelVar '(' StrExp ',' StrExp ')' ':=' BoolExp ';'
          | 'PRINT' StrExp ';' | 'PRINT' BoolExp ';'

BoolExp:    RelVar '(' StrExp ',' StrExp ')'
          | 'TRUE' '(' StrVar ')' | 'FALSE' '(' StrVar ')'
          | BoolExp '&' BoolExp       // conjunction
          | BoolExp '|' BoolExp       // disjunction
          | '!' BoolExp               // negation
          | 'EXISTS' '(' StrVar ',' BoolExp ')'
          | 'MATCH' '(' RegExp ',' StrVar ')'
          | 'A-REACH' '(' BoolExp ',' BoolExp ',' PropVar ')'
          | 'R-REACH' '(' BoolExp ',' BoolExp ',' PropVar ')'

StrExp:     StrLit | StrVar
```

**Fig. 3.** Partial syntax of the query language.

function name, or a code fragment. The property variables range over observer automata (i.e., trace properties), as defined in the previous section. The relation variables range over sets of tuples of strings. There is no need to declare the type of a variable; it is determined by the value of the first assignment to the variable. For the convenient and structured expression of more complex queries, the language also has constructs (`IF`, `WHILE`, `FOR`) for the conditional execution and iteration of statements.

The expression language permits first-order quantification over string variables. In the right-hand side expression of an assignment, every variable must either be a relation variable and have been previously assigned a value, or it must be a string variable that is quantified or occurs free. The implemented query language allows relations of arbitrary arity, but for simplicity, let us restrict this discussion to binary relation variables. Also, let us write $x$ and $y$ for the values of the string variables `x` and `y`, and $R$ for the set of pairs denoted by the binary relation variable `R`. Then the boolean expression `R(x,y)` evaluates to true iff $(x, y) \in R$. To assign a new value to the relation variable `R` we write "`R(x,y) := e`" short for "for all `x,y` let `R(x,y) := e`," where `e` is a boolean expression that may contain free occurrences of `x` and `y`.

Each print statement has as argument a boolean expression, with possibly some free occurrences of string variables. The result is a print-out of all value assignments to the free variables which make the expression true. For example, "`PRINT R(x,y)`" outputs the header (`x`, `y`) followed by all pairs $(x, y)$ of strings such that $(x, y) \in R$.

The grammar for queries without control-flow constructs is shown in Figure 3. The nonterminals `StrVar`, `PropVar`, and `RelVar` refer to any C identifier; `StrLit` is a string literal; `Observer` is a specification of an observer automaton, as defined in Section 2; and `RegExp` is a Unix regular expression.

### 3.2 Semantics

The first-order constructs (conjunction, disjunction, negation, existential quantification) as well as the imperative constructs (assignments, control flow, output) have the usual meaning. The boolean expression `MATCH(e,x)` evaluates to true iff the value of the string variable `x` matches the regular expression `e`.

**Reachability queries.** Consider an input program $P$, and a property variable `A` denoting an observer automaton $A$. Let *source* and *target* be two boolean expressions each with a single free string variable, say `loc_s` and `loc_t`. The boolean expression `A-REACH(`*source*`,` *target*`, A)` evaluates to true for given values $\ell$ for `loc_s` and $\ell'$ for `loc_t` iff *source* and *target* evaluate to true for $\ell$ and $\ell'$, respectively, and $\ell'$ is $A$-accept-reachable in $P$ from $\ell$. The boolean expression `R-REACH(`*source*`,` *target*`, A)` evaluates to true for given values $\ell$ for `loc_s` and $\ell'$ for `loc_t` iff *source* and *target* evaluate to true for $\ell$ and $\ell'$, respectively, and $\ell'$ is $A$-reject-reachable in $P$ from $\ell$. These relations are evaluated by invoking the Blast model checker on the instrumented program.

**Syntactic sugar.** Using the above primitives, we can define some other useful queries as follows. The property variable `Empty` denotes the empty observer automaton, which has no events and pre- and postconditions that are always true. The macro `REACH(`*source*`,` *target*`)` is short-hand for `A-REACH(`*source*`,` *target*`, Empty)`; it evaluates to true for given values $\ell$ for `loc_s` and $\ell'$ for `loc_t` iff both *source* and *target* evaluate to true and there is a feasible path in $P$ from $\ell$ to $\ell'$. The macro `SAFE(`*source*`, A)` is short-hand for

*source* `& !EXISTS(loc_t, R-REACH(`*source*`, TRUE(loc_t), A))`

This boolean expression evaluates to true for a given value $\ell$ for `loc_s` iff *source* evaluates to true and there is no feasible path in $P$ from $\ell$ which makes the observer $A$ enter a rejecting state.

**Syntactic relations.** There are a number of useful predefined syntactic relation variables. These are restricted to relations that can be extracted from the AST of the program. The following relations are automatically initialized after starting the query interpreter to access information about the syntactic structure of the program:

- `LOC_FUNC(loc,fname)` evaluates to true iff the program location `loc` is contained in the body of the C function `fname`.
- `LOC_FUNC_INIT(loc,fname)` evaluates to true iff the program location `loc` is the initial location of the C function `fname`.
- `LOC_LABEL(loc,lname)` evaluates to true iff the location `loc` contains the C label `lname`.
- `LOC_LHSVAR(loc,vname)` evaluates to true iff the location `loc` contains the variable `vname` on the left-hand side of an assignment.
- `LOC_RHSVAR(loc,vname)` evaluates to true iff the location `loc` contains the variable `vname` on the right-hand side of an assignment.
- `LOC_TEXT(loc,sourcecode)` evaluates to true iff the C code at the location `loc` is identical to `sourcecode`.

– CALL(fname,funcname_callee) evaluates to true iff the function fname (syntactically) calls the function funcname_callee.

Other relations that reflect the syntactic structure of the program can be added as needed.

*Example 2.* [**Reachability analysis**] The following query computes all reachable lines that contain the program code abort:

```
source(loc) := LOC_FUNC_INIT(loc,"main");
target(loc) :=
    EXISTS(text, LOC_TEXT(loc,text) & MATCH("abort",text));
result(loc1,loc2) := REACH(source(loc1),target(loc2));
PRINT result(loc1,_);
```

The first statement of the query assigns a set of program locations to the relation variable source. The set contains all locations that are contained in the body of function main. The second statement constructs the set of program locations that contain the code abort. The third statement computes a set of pairs of program locations. A pair of locations is contained in the set result iff there is an executable program trace from some location in source to some location in target. The last statement prints out a list of all source locations with a feasible path to an abort statement. The symbol "_" is used as an abbreviation for an existentially quantified string variable which is not used elsewhere. □

*Example 3.* [**Dead-code analysis**] The following query computes the set of locations of the function main that are not reachable by any program execution (the "dead" locations):

```
live(loc1,loc2) :=
    REACH(LOC_FUNC_INIT(loc1,"main"),LOC_FUNC(loc2,_));
reached(loc) := live(_,loc);
PRINT "Following locations within 'main' are not reachable:";
PRINT !reached(loc) & LOC_FUNC(loc,"main");
```

We first compute the set of all program locations that are reachable from the initial location of the function main. We print the complement of this set, which represents dead code, restricted to the set of locations of the function main. □

Both of the above examples are simple reachability queries. Examples of more advanced queries, which combine location and trace properties, are presented in the next section.

## 4 Examples

**Impact analysis.** Consider the C program displayed in Figure 4(a). At the label START, the variable j is assigned a value. We wish to find the locations that are affected by this assignment, i.e., the reachable locations that use the variable j

```
1 int j;
2 void f(int j){};
3 int compute() {
4   int i;
5   START: j = 1;
6   i = 1;
7   if (i==0) {
8     f(j);      // affected if i==0
9   }
10  if (i==0) {
11    j = 2;
12  }
13  if (j==2) {  // affected if i==1
14    f(j);      // not affected if i==0
15  }
16  return 0;
17 }
18 int main() {
19   compute();
20   return 0;
21 }
```

```
GLOBAL int gDefined ;
INITIAL { gDefined == 0 }
EVENT {
    PATTERN { j = $1; }
    ACTION { gDefined ++ ; }
}
FINAL  { gDefined == 1 }
```

**Fig. 4.** (a) C program. (b) Impact automaton `A`.

before it is redefined. Consider the observer automaton `A` shown in Figure 4(b). Along a trace, every assignment to `j` increments the variable `gDefined`. Thus, `gDefined` is equal to `1` only when there has been exactly one definition of `j`. The final condition ensures that along a finite trace, no redefinition of `j` has occurred. Hence, the desired set of locations is computed by the following query:

```
affected(l1,l2) :=
    A-REACH(LOC_LABEL(l1,"START"), LOC_RHSVAR(l2,"j"), A);
PRINT affected(_,l2);
```

For our example, Blast reports that the definition of the variable `j` at line 5 has impact on line 13. It has no impact on line 8, as that line is not reachable because of line 6. On the other hand, if line 6 is changed to "`i=0;`", then line 8 is reachable and affected. Now, line 11 is reachable and therefore a redefinition of `j` takes place. Thus, line 13 is not affected. To compute the effect of each definition of `j`, we can change the first argument of `A-REACH` to `LOC_LHSVAR(l1,"j")`.

```
GLOBAL int __E;
INITIAL (__E == 0);
EVENT {
  PATTERN { $? = seteuid($1); }
  ACTION { __E = $1; }
}
```

```
EVENT {
  PATTERN { $? = system($?); }
  ASSERT { __E != 0 }
}
```

**Fig. 5.** (a) Effective UID automaton. (b) Syscall privilege automaton.

**Security analysis.** Consider a simplified specification for the manipulation of privileges in setuid programs [4]. Unix processes can execute at several privilege levels; higher privilege levels may be required to access restricted system resources. Privilege levels are based on user id's. The `seteuid` system call is used to set the effective user id of a process, and hence its privilege level. The effective user id `0` (or root) allows a process full privileges to access all system resources. The `system` call runs a program as a new process with the privilege level of the current effective user id. The observer automaton `B` in Figure 5(a) tracks the status of the effective user id by maintaining an observer variable `__E`, which denotes the current effective user id. Initially, `__E` is set to `0`. The `$1` pattern variable in the `seteuid` pattern matches the actual parameter. Every time `seteuid` is called, the value of `__E` is updated to be equal to the parameter passed to `seteuid` in the program.

Suppose we want to check that the function `system` is never called while holding root privileges. This can be done by adding the event in Figure 5(b) to the automaton `B` (call the resulting automaton `B'`) and computing the query "`SAFE(LOC_FUNC_INIT(loc, "main"), B')`". The `$?` wild-card in the `system` pattern is used to match all remaining parameters. As long as the assertion is satisfied, the observer does nothing, because the action is empty; however, if the assertion is not satisfied, the trace is rejected.

Now suppose we want to know which locations of the program can be run with root privileges, i.e., with `__E = 0`. This can be accomplished by the following query:

```
target(loc) := LOC_FUNC(loc,_);
rootPriv(loc1,loc2) :=
    A-REACH(LOC_FUNC_INIT(loc1, "main"), target(loc2), B");
PRINT rootPriv(_,loc);
```

where automaton `B"` is automaton `B` with the final condition "`FINAL (__E==0);`".

**Decomposing verification tasks.** We now show how the relational query language and observer automata can be combined to decompose the verification process [14]. Consider an event $e$ of an observer automaton $A$ with the postcondition $post_A$. We say that $e$ *extends* $A$ if (1) the assertion of $e$ is always true, and (2) the action of $e$ writes only to variables not read by $A$. Let $A.e$ be the observer automaton obtained by adding to $A$ (1) a fresh observer variable `x_e`, (2) the initial condition `x_e == 0`, and (3) the code `x_e = 1` as the first instruction in the body of the action of $e$. Define $\mathsf{RSplit}.A.e$ to be the pair of observer automata $(A.e_r^+, A.e_r^-)$ which are $A.e$ with the postconditions changed to $\mathtt{x\_e} = 1 \Rightarrow post_A$ and $\mathtt{x\_e} \neq 1 \Rightarrow post_A$, respectively. Define $\mathsf{ASplit}.A.e$ to be the pair of automata $(A.e_a^+, A.e_a^-)$ which are $A.e$ with the postconditions changed to $\mathtt{x\_e} = 1 \wedge post_A$ and $\mathtt{x_e} \neq 1 \wedge post_A$, respectively.

**Lemma 1.** *Let $P$ be a program, let $A$ be an observer automaton, and let $e$ be an event that extends $A$. Let $(A1, A_2) = \mathsf{RSplit}.A.e$ (resp. $(A_1, A_2) = \mathsf{ASplit}.A.e$). A location $\ell'$ is $A$-reject-reachable (resp. $A$-accept-reachable) in $P$ from $\ell$ iff* *either $\ell'$ is $A_1$-reject-reachable (resp. $A_1$-accept-reachable) in $P$ from $\ell$, or $\ell'$ is $A_2$-reject-reachable (resp. $A_2$-accept-reachable) in $P$ from $\ell$.*

The split partitions the program traces into those where the event $e$ occurs and those where it doesn't occur. We can now extend our query language to allow for boolean macro expressions of the following kind: $b$ `SPLIT` $e$, where $b$ is a boolean expression and $e$ is an event. This macro stands for $b$ with each occurrence of a subexpression of the form `R-REACH(`$\cdot$`,` $\cdot$`,` $A$`)`, where $e$ extends $A$, replaced by `R-REACH(`$\cdot$`,` $\cdot$`,` $A_1$`) | R-REACH(`$\cdot$`,` $\cdot$`,` $A_2$`)`, where $(A_1, A_2) = \mathsf{RSplit}.A.e$, and each occurrence of a subexpression of the form `A-REACH(`$\cdot$`,` $\cdot$`,` $A$`)` replaced with `A-REACH(`$\cdot$`,` $\cdot$`,` $A_1$`) | A-REACH(`$\cdot$`,` $\cdot$`,` $A_2$`)`, where $(A_1, A_2) = \mathsf{ASplit}.A.e$. By Lemma 1, the boolean expression $b$ `SPLIT` $e$ is equivalent to $b$. With a judicious choice of events, we can therefore break down the evaluation of a complex query into multiple simpler queries.

We illustrate this using the example of a Windows device driver for a floppy disk[4], and concentrate the Plug and Play (PNP) manager, which communicates requests to devices via I/O request packets. For example, the request `IRP_MN_START_DEVICE` instructs the driver to do all necessary hardware and software initialization so that the device can function. Figure 6 shows the code for the PNP manager. The code does some set-up work and then branches to handle each PNP request. We wish to verify a property of the driver that specifies the way I/O request packets must be handled[5]. Let `A` be the observer automaton for the property.

```
NTSTATUS FloppyPnp( IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp) {
    ...
    PIO_STACK_LOCATION irpSp = IoGetCurrentIrpStackLocation( Irp );
    ...
    switch ( irpSp->MinorFunction ) {
      L_1: case IRP_MN_START_DEVICE:
             ntStatus = FloppyStart(DeviceObject, Irp);
             break;
      L_2: case IRP_MN_QUERY_STOP_DEVICE:
              ...
             break;
             // several other cases
      L_k: default:
             ...
    }
    ...
    return ntStatus;
}
```

**Fig. 6.** A floppy driver.

4 Available with the Microsoft Windows DDK.

5 Personal communication with T. Ball and S. Rajamani.

Intuitively, the verification can be broken into each kind of request sent by the PNP manager, that is, if we can prove the absence of error for each case in the switch statement, we have proved the program correct with respect to the property. Let `e_1`, ..., `e_k` be the events that denote the reaching of the program labels `L_1`, ..., `L_k`, which correspond to each case in the switch statement. The following relational query encodes the proof decomposition:

```
( ... (SAFE(LOC_FUNC_INIT(loc,"FloppyPnp"), A) SPLIT e_1)
                                                   ... SPLIT e_k)
```

This query breaks the safety property specified by `A` into several simpler queries, one for each combination of possible branches of the switch statement. While this results in exponentially many subqueries, all but $k$ of these subqueries (where more than one, or none of the events happens) are evaluated very efficiently by exploiting the syntactic control-flow structure of the program, by noting that a violation of the subproperty is *syntactically* impossible. The remaining $k$ cases, which are syntactically possible, are then model checked independently, leading to a more efficient check, because independent abstractions can be maintained.

## 5 Tool Architecture

The overall architecture of the implementation is shown in Figure 7. CIL [15] parses the input program and produces the AST used by the program transformer. The *query parser* parses the specification file and extracts program-transformation rules to later guide the program instrumentation. It also prepares the data structures for the relational computations. The *program transformer* takes as input the representation of the original program and the transformation rules. When required by the query interpreter, it takes one particular set of transformation rules at a time (corresponding to one observer automaton) and performs the instrumentation. The result is the AST of the instrumented code. The *query interpreter* is the central controlling unit in this architecture. It dispatches the current query from the query queue to the relational-algebra engine for execution. If the next statement is a `REACH` expression, it first requests the instrumented version of the program from the transformer, then requests the relational-manipulation engine to transfer the input relations to the model-checking engine, and then starts the model checker BLAST. When the model checking is completed, the relational-manipulation engine stores the results of the query and gives the control back to the query interpreter.

The *relational-algebra engine* is a calculator for relational expressions. It uses a highly optimized BDD-based library for querying and manipulating relations [3]. This library deals with relations on the level of predicate calculus. There is no need to encode variables and values to bit representations, because the library provides automatic value encoding and efficient high-level operations to abstract from the core BDD algorithms.

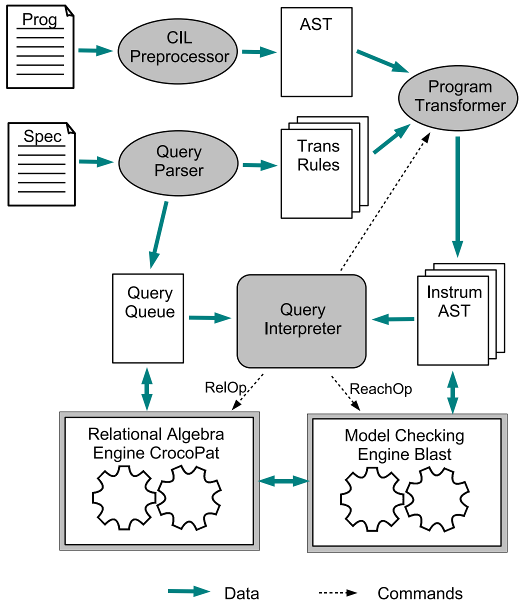


**Fig. 7.** Architecture of the verification toolkit.

## References


1. T. Ball and S.K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. POPL*, pages 1–3. ACM, 2002.
2. T. Ball and S.K. Rajamani. SLIC: A specification language for interface checking (of C). Technical Report MSR-TR-2001-21, Microsoft Research, 2002.
3. D. Beyer, A. Noack, and C. Lewerentz. Simple and efficient relational querying of software structures. In *Proc. WCRE*, pages 216–225. IEEE, 2003.
4. H. Chen and D. Wagner. MOPS: An infrastructure for examining security properties of software. In *Proc. CCS*, pages 235–244. ACM, 2002.
5. J.C. Corbett, M.B. Dwyer, J. Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software. In *Proc. SPIN*, LNCS 1885, pages 205–223. Springer, 2000.
6. S. Goldsmith, R. O'Callahan, and A. Aiken. Lightweight instrumentation from relational queries on program traces. Technical Report CSD-04-1315, UC Berkeley, 2004.
7. S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific static analyses. In *Proc. PLDI*, pages 69–82. ACM, 2002.
8. T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from proofs. In *Proc. POPL*, pages 232–244. ACM, 2004.
9. T.A. Henzinger, R. Jhala, R. Majumdar, G.C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *Proc. CAV*, LNCS 2404, pages 526–538. Springer, 2002.
10. 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.

11. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. POPL*, pages 58–70. ACM, 2002.
12. G.J. Holzmann. Logic verification of ANSI-C code with SPIN. In *Proc. SPIN*, LNCS 1885, pages 131–147. Springer, 2000.
13. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. ECOOP*, LNCS 1241, pages 220–242. Springer, 1997.
14. K.L. McMillan. A methodology for hardware verification using compositional model checking. *Science of Computer Programming*, 37(1–3):279–309, 2000.
15. G.C. Necula, S. McPeak, S.P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. CC*, LNCS 2304, pages 213–228. Springer, 2002.
16. R.E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Engineering*, 12(1):157–171, 1986.
17. E. Yahav and G. Ramalingam. Verifying safety properties using separation and heterogeneous abstractions. In *Proc. PLDI*, pages 25–34. ACM, 2004.