Journal of Logical and Algebraic Methods in Programming 85 (2016) 1292–1317

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

LCTD: Test-guided proofs for C programs on LLVM ☆

Olli Saarikivi *, Keijo Heljanko

Helsinki Institute for Information Technology HIIT, Department of Computer Science, School of Science, Aalto University, PO Box 15400, FI-00076 Aalto, Finland

ARTICLE INFO

Article history:
Received 25 March 2014
Received in revised form 30 October 2015
Accepted 30 October 2015
Available online 6 November 2015

Keywords:
Automated testing
Verification
Dynamic symbolic execution
Abstraction refinement
Predicate abstraction
Weakest precondition

ABSTRACT

Recently there has been much interest in combining underapproximation and overapproximation based approaches to software verification. Such a technique is employed by the Dash algorithm originally developed at Microsoft, which generates tests to gradually improve the accuracy of an underapproximation of the program under test. Simultaneously, an overapproximating abstraction of the program is refined with information gathered from the test generation.

We present LCTD, an open source tool that implements the Dash algorithm for the verification of C programs compiled on the LLVM compiler framework. Our implementation is an extension of the dynamic symbolic execution tool LCT. We also present a detailed description of our method for constructing the weakest precondition based refinement operator employed by Dash for instructions of the LLVM internal representation. Our construction handles pointers and array indexing.

To maintain a mapping between concrete executions and the abstraction Dash needs to evaluate predicates on the concrete states visited during test executions. A straightforward implementation might store the complete concrete states of each executed test or might employ expensive re-executions to recover the concrete states. We present a technique which allows only the concrete values of pointer variables to be stored while still requiring no re-executions.

Finally we present a case study to show the viability of our tool. We also document a more powerful abstraction refinement method for Dash that exploits unsatisfiable regions and evaluate its effect.

1. Introduction

As our reliance on software grows, ensuring software correctness is an increasingly important task for the software industry. The cost of a single Microsoft security bulletin has been estimated to be in the order of millions of dollars [1]. In safety critical systems software errors can place human lives in danger. For example, the accident of the Qantas Flight 72 in 2008 resulting in serious injuries was in part attributable to software defects [2]. Software defects have also been linked to four spacecraft accidents during the years 1996–1999 [3]. Thorough verification and rigorous requirements engineering have been identified as key contributors to the success of the NASA Space Shuttle program [4].

Software verification is one approach to making sure that programs do not contain errors. Automated verification methods involve exploration of the *state space* of the program to either find an error or a proof of program correctness. However,

☆ This article is a full version of the extended abstract presented at the 25th Nordic Workshop on Programming Theory (NWPT 2013) in Tallinn.

* Corresponding author. Tel.: +358 405240696.
E-mail addresses: olli.saarikivi@aalto.fi (O. Saarikivi), keijo.heljanko@aalto.fi (K. Heljanko).

http://dx.doi.org/10.1016/j.jlamp.2015.10.010
2352-2208/

the state space of a program can be very large as it often is exponential in the number of program locations (lines of code) and amount of memory used by the program. This problem, dubbed state space explosion [5], often makes enumerating all states of the program infeasible.

One approach to exploring the state space of a program is *testing*, where a program is executed with sets of input values that will drive the execution to different parts of the state space. Many kinds of software errors, e.g., assertion violations, involve the reachability of specific program locations and therefore it is desirable for the set of test executions to provide a good coverage of the program's source code. Dynamic symbolic execution (DSE) is a testing method that systematically increases coverage by generating input values that will drive the tests to previously unexplored execution paths (see for example [6–9]). In DSE the execution of a test is monitored to collect a symbolic constraint over the program's input variables for the path taken by the execution. This constraint can then be supplied to a satisfiability modulo theories (SMT) solver to generate new inputs that will drive subsequent executions to follow previously unexplored paths.

Even with each new test exploring a previously unexplored path, DSE does not yet scale well to large real-world programs while maintaining full path coverage [10]. Therefore, verification methods based on testing will in practice underapproximate the state space, i.e., the set of states explored by the test executions will be a subset of the program's state space. This means that once the testing is done we cannot say for sure whether the program is correct: there could still exist unexplored inputs to the program that would result in errors.

Another approach to state space exploration is model checking, where a model of a program is exhaustively explored to check whether it meets a specification. While explicit state model checking of software is often infeasible for large programs, it may be possible to use an abstracted version of the program to check it. With abstraction the behavior of the program is overapproximated by leaving out some details. For example, in predicate abstraction [11] an abstract state space is formed from the different valuations of a set of predicates on concrete variables of the program to be verified. The abstract state space can often have a much smaller representation than the state space of the original program, which may make it feasible to prove that the abstracted program is correct. Because the abstracted program is an overapproximation, any bad behavior of the original program is also present in the abstraction and therefore if the abstraction is correct then so is the original program. However, if the abstracted program is not correct then we cannot necessarily say anything about the original program. The bad behavior that leads to an error, called a *counterexample*, may be an actual behavior of the original program, in which case it should be reported as a bug. On the other hand, the counterexample might not be a feasible behavior of the original program, in which case it is called a *spurious* counterexample.

To get rid of spurious counterexamples various *abstraction refinement* methods have been developed. An abstracted program may contain a spurious counterexample when too many details have been left out in the abstraction process. This can be remedied by refining the abstraction, i.e., adding back some previously ignored facts about the program. The process of selecting how to refine the abstraction can be guided by analysis of the counterexample [12]. This approach, called counterexample guided abstraction refinement (CEGAR), has been successfully used in a number of tools [13–16]. These tools work by iteratively refining the abstraction until it is sufficiently precise to prove that the program is correct (assuming that it indeed is). Constructing the abstract transition relation after each refinement may be costly. For example, with the technique described by Graf and Saidi [11] computing a successor state of a transition requires potentially a large number of theorem prover calls. One way in which this can be alleviated is by only refining the parts of the abstraction needed for eliminating the spurious counterexample. An algorithm following this approach, called lazy abstraction, has been implemented in the Blast software verification tool [15]. With the lazy abstraction approach the work done to construct the abstract transition relation can be partly reused when the abstraction is refined.

Combining DSE-based testing with abstraction refinement is attractive because the two approaches can efficiently handle different types of programs [17]. Abstraction refinement works well for programs that require tracking a relatively small number of predicates to prove a property. Testing, on the other hand, can quickly find feasible paths through code which could otherwise require refining the abstraction many times. The Dash algorithm [18] combines abstraction refinement with DSE in such a way that it retains advantages from both. It simultaneously constructs a concrete execution tree of the program and a partition of the program into abstract regions. From a testing point of view, Dash works similarly to DSE with the addition that the branches to expand are selected along error traces found in the abstraction. From an abstraction refinement point of view, the test generation of Dash is how counterexamples are produced. If the counterexample is spurious, then at some point an SMT solver call to generate a test further along the error trace will fail. At this point the abstraction is refined in a lazy manner to remove the error trace.

In this work we present an implementation of the Dash algorithm for C programs. The tool, called LCTD, is a modification to the DSE tool LCT [19–22]. We have implemented our program transformations with the LLVM compiler framework [23]. The main contributions of this work are as follows:

1. We present LCTD, an open source tool implementing the Dash algorithm for verifying C programs compiled to the LLVM intermediate representation.
2. We describe a strategy for constructing weakest preconditions for LLVM basic blocks in the presence of pointers and arrays. How to handle arrays in Dash has not been described before (see Section 3).
3. We describe a method for mapping traces from test executions back to the abstraction that does not require directly evaluating predicates of abstract regions. The method allows only concrete pointer values instead of the concrete state of the whole program to be stored (see Section 4.1).

```
int main() {                                                    1
    int x = input();                                            2
    if (x == 13) {                                              3
        if (x < 10)                                             4
            error();                                            5
        int y = input();                                        6
        x = x + y;                                              7
        if (x < 0)                                              8
            error();                                            9
    }                                                           10
    return 0;                                                   11
}                                                               12
```

Fig. 2.1. An example program for DSE.

4. We document and evaluate the effectiveness of a more powerful abstraction refinement method that can be applied when an abstract region's predicate becomes unsatisfiable (see Section 4.3).
5. We present initial case studies showing the viability of our tool (see Section 5).

The rest of this work is structured as follows. Section 2 presents the necessary background for understanding our contributions: dynamic symbolic execution is explained and the DASH algorithm is described in the context of a program model reminiscent of the LLVM intermediate representation. In Section 3 we present our strategy for constructing weakest preconditions and the alternate abstraction refinement method for unsatisfiable predicates in regions. Section 4 describes our implementation, including our method for mapping execution traces back to the abstraction. In Section 5 we present preliminary case studies and discuss their results. In Section 6 we review related work. Finally, Section 7 provides some concluding remarks and discussion on directions for future research.

2. Background

2.1. Dynamic symbolic execution

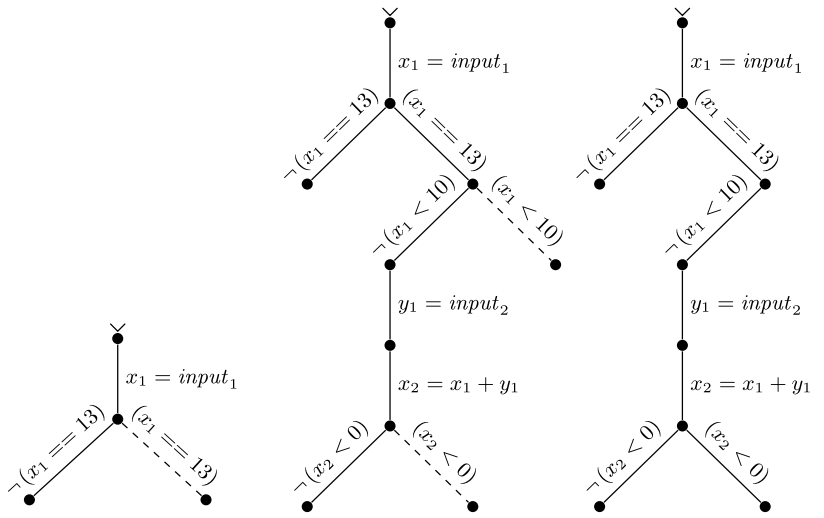
Dynamic symbolic execution (see for example [6–8]) is a technique for systematically exploring all execution paths of a program. In DSE symbolic constraints for program inputs are gathered during execution to form a *path constraint* for the execution. With an SMT-solver, this path constraint can be solved to derive new sets of inputs that will drive subsequent executions to follow previously unexplored paths.

We now apply DSE to the example program in Fig. 2.1. The example program marks its inputs by calls to `input()`. The program also has two lines, 5 and 9, which have calls to `error()`. We would like to find a set of inputs that will cause the program to execute one of these calls.

Initially the program will be executed with random inputs. Let us assume the first random input given to the program is 5, which means that `x == 13` will not hold and the next statement executed will be the **`return`** `0`, terminating the program. During this execution symbolic counterparts for the statements executed are recorded in the *execution tree* shown in Fig. 2.2(a). For the statement on line 2 we get the constraint $x_1 = input_1$ indicating that the first input gets assigned to `x`'s first symbolic value x_1. In terms of the SMT solver, x_1 and $input_1$ are free constants that are constrained by the path constraint. For the if statement two child nodes corresponding to the two branches are added to the tree. The edges to the children are labeled with constraints that must be true for the program to follow the path to the child in question. In our example the constraint for the branch taken in the first execution is $\neg(x_1 = 13)$ and the constraint for the unexplored branch is the negation of the first one, $x_1 = 13$.

To generate the inputs for the next execution we select the unexplored branch which is shown in Fig. 2.2(a) drawn with a dashed line. The path constraint for reaching this branch is $(x_1 = input_1) \wedge (x_1 = 13)$. This constraint is passed to an SMT solver, which will return a satisfying assignment where $input_1$ gets the value 13. The program is then executed again with the new input assignment. This time the program will go to the true branch of the first if statement and the false branch of the second one. On line 6 a second call to `input()` is encountered. Because this value was not set in the input assignment we instead use a random value. For this example let us assume the value 20 is used, which results in the execution taking the false branch also for the last if statement on line 8.

The path followed by the second execution is added to the execution tree. The updated tree can be seen in Fig. 2.2(b). For the next execution let us assume the path selected is the one to the `error()` call on line 5, for which the path constraint is $(x_1 = input_1) \wedge (x_1 = 13) \wedge (x_1 < 10)$. When querying an SMT solver for an assignment for this constraint we will find that the constraint is unsatisfiable (evident from x_1 being required to be both 13 and less than 10). From this we know that there exist no inputs with which the program would reach the node at the end of the target path. Therefore, we remove that node from the execution tree.



(a) After initial test run (b) Second test added (c) Final execution tree

Fig. 2.2. Various stages of an execution tree constructed with dynamic symbolic execution.

The next iteration begins by selecting the lone unexplored path in Fig. 2.2(b), and passing its path constraint, $(x_1 = input_1) \wedge (x_1 = 13) \wedge \neg(x_1 < 10) \wedge (y_1 = input_2) \wedge (x_2 = x_1 + y_1) \wedge (x_2 < 0)$, to an SMT solver. We get a satisfying input assignment of, for example, $input_1 = 13$ and $input_2 = -15$. Executing the program with these inputs will cause it to follow the desired path after which it executes line 9 and thus reaches an error. The final execution tree with the previous infeasible path removed and the path to the error expanded is shown in Fig. 2.2(c).

For a more in depth explanation of DSE see for example the paper by Godefroid, Klarlund and Sen [6].

2.2. Program model

In our model a program P is a tuple (N, E, n_0, λ, B), where N is a finite set of nodes corresponding to program locations (points between statements), $E \subseteq N \times N$ is a set of control flow edges, n_0 is the program's entry point, $\lambda : E \to Stmts$ labels each control flow edge with a statement, and B is the set of variables in the program. We assume that all variables are of either integer or pointer type.

In the following explanation we use the notion of a *map*, which is a partial function with some additional notation for updating it. Given a map M we adopt the following notations:

- $M(x)$ is the value of x in the map M.
- $M[x \mapsto y]$ is a map otherwise identical to M, except that now x maps to y.
- $M[x \not\mapsto]$ is a map otherwise identical to M, except that the mapping for x has been removed.

The set *Stmts* in the definition above is the set of statements present in the program P. Table 2.3 lists the types of statements used in our program model, which compared to that of Beckman et al. [18] has been extended with explicit `allocate`, `load` and `store` statements and also includes support for array indexing with the `gep` instruction. Programs written with these statements correspond to single procedure programs in a subset of the LLVM intermediate representation (LLVM IR) [24]. Instructions not supported include all instructions related to calling functions, instructions for atomic handling of memory, instructions for exception handling, `inttoptr`, `ptrtoint`, `addrspacecast`, `va_arg`, `switch` and instructions for floating point numbers, vectors and aggregates. For better readability we have chosen to write the binary and the comparison expressions in the style of the C programming language. The program model does not explicitly support constants in operations and constants can instead be encoded as extra variables, the values of which are set in the initial state of the program.

The *state* of a program is a tuple (n, Y, W, U), where n is the current program location, Y and W map the program's variables and memory addresses, respectively, to their current values and U is a set of addresses that have already been used for memory allocation. The type of addresses is the natural numbers, and all other values are 32 bit integers with two's complement semantics. Note that the specific semantics of arithmetic operations in the program model affect how they are to be modeled in the SMT solver, but the algorithm per se is agnostic to them.

The state space Σ of the program is the product of all possible values of n, Y, W and U. Now we define the transition relation $\to \subseteq \Sigma \times \Sigma$ so that for all $s, s' \in \Sigma$ it holds that $s \to s'$ if and only if there is a statement $op \in Stmts$ that is enabled in s such that executing op in s according to the semantics laid out in Table 2.3 would take the program to the state s'. Note that the semantics do include undefined operations. For example we do not define what happens on division by zero. From

Table 2.3
The statements of the program model and their semantics.

Given the current state (n, Y, W, U), a statement op is enabled if there exists a control flow edge $(n, n') \in E$ such that $op = \lambda(n, n')$. After a statement is executed the new state is (n', Y', W', U'). How Y', W' and U' change depends on the statement.	
Statement	Semantics
`r = v1 <op> v2`	The program moves to the state (n', Y', W, U), where $Y' = Y[r \mapsto \text{Eval}(Y(v_1) < \text{op} > Y(v_2))]$. The function Eval($e$) computes the result of an expression e. The operation `<op>` is one of the binary operators available in the C language. All operands are 32 bit integers. Booleans are represented with zero as false and other values as true. We do not consider pointers to be integers, i.e., all pointer arithmetic is forbidden. We assume that no undefined operations (for example divide by zero) are encountered.
`r = v`	The program moves to the state (n', Y', W, U), where $Y' = Y[r \mapsto Y(v)]$, i.e., the value of v is assigned to r.
`r = input`	The program moves to a state where the value of the variable r in Y is replaced with an arbitrary 32 bit integer. Because this statement induces multiple transitions the result of executing it is nondeterministic.
`r = load a`	The program moves to the state (n', Y', W, U), where $Y' = Y[r \mapsto W(Y(a))]$. $Y(a)$ must be a memory address.
`store v a`	The program moves to the state (n', Y, W', U), where $W' = W[Y(a) \mapsto Y(v)]$. $Y(a)$ must be a memory address.
`a = allocate c`	A natural number $m \in \mathbb{N}$ such that $[m \dots m + Y(c)) \notin U$ is selected as the memory address and the program moves to the state (n', Y', W, U'), where $Y' = Y[a \mapsto m]$ and $U' = U \cup [m \dots m + Y(c))$. $Y(c)$ must be a strictly positive 32 bit integer.
`a = gep b v`	The program moves to the state (n', Y', W, U), where $Y' = Y[a \mapsto Y(b) + Y(v)]$. We assume the result is always in the same address range allocated by a single `allocate` statement as b is. $Y(b)$ must be a memory address and $Y(v)$ must be a 32 bit integer.
`(v1 <comp> v2)` `!(v1 <comp> v2)`	In addition to the requirement that the program location is correct these statements are only enabled if also the condition represented is true. Otherwise these guard statements have no effect. The comparison operator `<comp>` is one of the comparison operators available in the C language.

now on we assume that programs have no concrete executions which would result in undefined behavior. Also we assume that all pointer arithmetic through the `gep` statement stays in the same range allocated by a single `allocate` statement as the base pointer. However, these assumptions do not result in loss of generality as all programs can be transformed to meet both assumptions by adding appropriate assertions before the potentially undefined operations. Note that assertions are not directly represented in the program model, but instead surface in Dash as extra targets for checking reachability and thus it will be verified that the program cannot execute an undefined operation.

For an example of encoding a C program in our program model see Fig. 2.6 for a C program and Fig. 2.7(a) for its representation in the program model (ignore the wavy line for now). Note in particular how the `if` statements have been encoded as nodes with two outgoing edges that have the positive and negative versions of the branch condition as guards.

The `allocate` statement deserves some explanation. The parameter c gives the size of the allocation and the memory address returned is such that a contiguous block of c addresses starting from m are not in the set of previously allocated addresses U. `allocate` also amends this set with the new range of addresses. In effect, the semantics of `allocate` specify the least acceptable semantics of a memory allocator. The fact that `allocate` is nondeterministic does not complicate how it is handled (see Section 3).

Let $s_0 \in \Sigma$ be the initial state of the program. A property $\varphi \subseteq \Sigma$ is a set of states that we want to verify to not be reachable from s_0. The problem instance is now a pair (P, φ) of the program and the property. Let $\rightarrow^+$ be the transitive closure of the relation $\rightarrow$. Now the answer to (P, φ) is "fail" if there exists a state $s \in \varphi$ such that $s_0 \rightarrow^+ s$ and "pass" otherwise.

As a set of states, the property φ can be any property expressible as a predicate over the program's variables and memory. This includes any property expressible with the C language's `assert` macro.

2.3. Dash

In this section we explain the Dash algorithm [18], which combines dynamic symbolic execution with counterexample guided abstraction refinement. Dash attempts to generate tests based on counterexamples found in the abstraction. When Dash fails to generate a test it refines the abstraction to remove the counterexample. The tests can be seen as an underapproximation of the reachable states of the program under test, which Dash tries to expand to include an error. The abstraction on the other hand is an overapproximation which, if error free, also proves the program under test to be so.

Determining the reachability of error states for arbitrary programs is in general undecidable and as such running Dash on a program has three potential outcomes:

- An error is found and inputs that lead the program to the error are returned.
- The verification succeeds if the abstraction is refined to remove all error traces.
- The algorithm may not terminate.

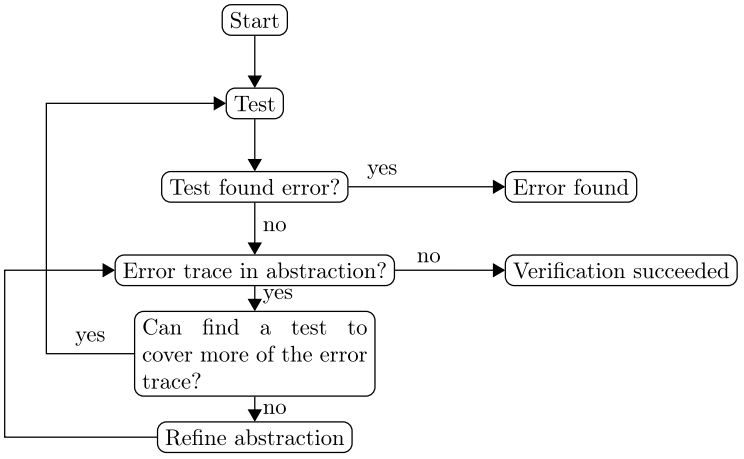


Fig. 2.4. Flowchart for the DASH algorithm.

The following description of the DASH algorithm has been adapted from Beckman et al. [18]. We extend this description in Section 3 with a detailed strategy for constructing the predicates used by DASH for refinement. That description includes our strategy for handling arrays.

For a given problem instance (P, φ) to detect "fail" instances the DASH algorithm will attempt to find a sequence of states $(s_0, s_1, \ldots, s_n)$ such that $s_0 \to s_1 \to \cdots \to s_n$ and $s_n \in \varphi$. We call such a sequence of states an *error trace*.

To detect "pass" instances the DASH algorithm maintains an abstraction $\Sigma_\simeq$ which partitions the state space Σ into a finite number of equivalence classes. We refer to the equivalence classes of the abstraction $\Sigma_\simeq$ as *regions*. Let $\to_\simeq \subseteq \Sigma_\simeq \times \Sigma_\simeq$ be a transition relation such that for all regions $R, R' \in \Sigma_\simeq$ it holds that $\exists s \in R, s' \in R' : s \to s'$ implies $R \to_\simeq R'$. In other words, the transition relation $\to_\simeq$ may be an overapproximation. Let $R_0 \in \Sigma_\simeq$ be the region that contains the initial state s_0 and $\varphi_\simeq$ be the set regions that contain a state from φ. An *abstract error trace* is a sequence of regions $(R_0, R_1, \ldots, R_n)$ such that $R_0 \to_\simeq R_1 \to_\simeq \cdots \to_\simeq R_n$ and $R_n \in \varphi_\simeq$. If a problem instance has an abstraction for which no abstract error trace exists then the answer to the instance is "pass".

In addition to the abstraction, the DASH algorithm maintains a set C of test runs, which are pairs consisting of an *execution trace*, represented by a sequence of states, and a sequence of input values.

The flowchart in Fig. 2.4 presents a rough sketch of the DASH algorithm. Pseudocode for its main procedure is presented in Fig. 2.5. DASH follows a modified version of counterexample guided abstraction refinement (CEGAR) [12,15,25]. Whenever an abstract error trace is found, DASH will attempt to generate a test that extends the set of regions that are known to be reachable along the abstract error trace. If the test generation fails then the counterexample was spurious and DASH will refine the abstraction to eliminate the abstract error trace. The main procedure uses the auxiliary procedures `GetAbstractTrace`, `GetCombinedConstraint`, `Solve`, `RunTest`, `RefinePred` and `SplitFrontier`. The purpose of these procedures will be explained in the following explanation of the algorithm. The procedures named `ExtendFrontier` and `GetOrderedAbstractTrace` in Beckman et al. [18] correspond to the procedures `GetCombinedConstraint` and `Solve` in this paper. The procedure `RunTest` is called `AddTestToForest` in Beckman et al. [18].

To explain the algorithm we will step through a running example of applying DASH to verify the program in Fig. 2.6. The example program takes one input, which is marked by the call to `input()`. We wish to verify that no matter what this input value is, the program cannot execute the error statement on line 6.

At startup DASH creates the initial abstraction $\Sigma_\simeq$ and related transition function $\to_\simeq$ from the program's control flow graph: states from Σ are partitioned into regions such that there is one region for each separate program location. There is a transition from a region R to another region R' if the program location of R has an edge to the program location of R'. The initial abstraction of our example program is shown in Fig. 2.7(a). Regions are represented by a program location that the contained states must be in together with an optional constraint written as a predicate over the program's variables and memory. None of the regions in the initial abstraction have any constraints yet, which can be seen by the regions in Fig. 2.7(a) being labeled only by line numbers from Fig. 2.6.

We have also labeled edges in Fig. 2.7(a) with their corresponding statements. Notice how the edges $(3, 4)$ and $(3, 8)$ are labeled with guard statements. These are used to represent the "then" and "else" branches of the if statement on line 3 of the program in Fig. 2.6.

Before entering the main loop DASH will run one test on the program with random inputs, which is accomplished by calling the procedure `RunTest` with an empty sequence of inputs. This results in a random test because once values in the provided input sequence run out, `RunTest` will supply random values to any `r = input` statements executed. The procedure `RunTest` returns a pair containing the sequence of states executed together with the input values used (including random ones).

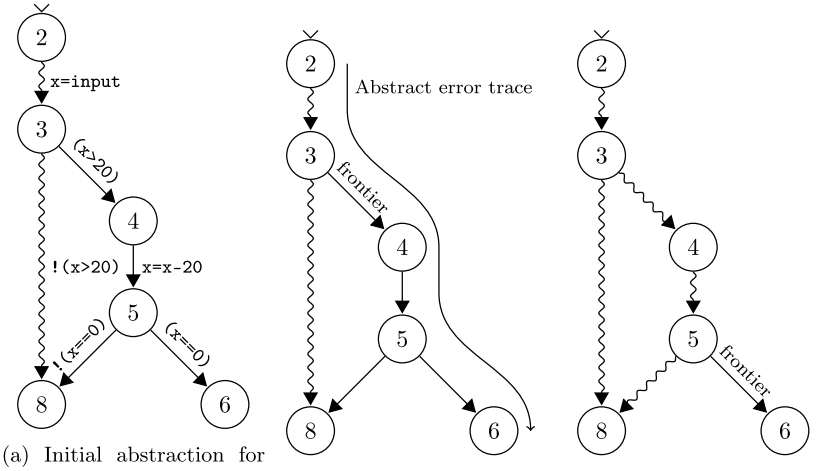
Input : P, φ
Output: ("fail", t), where t is an error trace of P, or
("pass", $\Sigma_{\simeq}$), where $\Sigma_{\simeq}$ does not have an abstract error trace

$\Sigma_{\simeq} := \bigcup_{n \in N} \{\{s \in \Sigma \mid \text{the program location of } s \text{ is } n\}\}$
$\rightarrow_{\simeq} := \{(R, R') \in \Sigma_{\simeq} \times \Sigma_{\simeq} \mid \text{exist } s \in R \text{ and } s' \in R' \text{ such that } (s, s') \in E\}$
$C := \{\texttt{RunTest}(\epsilon, P)\}$
loop
  if $\Sigma_{\simeq}$ *has an abstract error trace* **then**
    $\tau := \texttt{GetAbstractTrace}(\Sigma_{\simeq}, \rightarrow_{\simeq}, \varphi)$ `// Find counterexample`
    $(\phi, A) := \texttt{GetCombinedConstraint}(\tau, P, C)$
    $(isSat, I_{test}) := \texttt{Solve}(\phi)$ `// Check if spurious`
    if *isSat* **then**
      $(t, I_{all}) := \texttt{RunTest}(I_{test}, P)$ `// If not, extend frontier`
      $C := C \cup \{(t, I_{all})\}$
      if t *has a state from* φ **then**
        return ("fail", t)
      end if
    else
      $p := \texttt{RefinePred}(\tau, A)$ `// If spurious, refine abstraction`
      $(\Sigma_{\simeq}, \rightarrow_{\simeq}) := \texttt{SplitFrontier}(\Sigma_{\simeq}, \rightarrow_{\simeq}, \tau, p)$
    end if
  else
    return ("pass", $\Sigma_{\simeq}$)
  end if
end loop

Fig. 2.5. The DASH algorithm.

```
int main() {                                                    1
    int x = input();                                            2
    if (x>20) {                                                 3
        x = x-20;                                               4
        if (x==0)                                               5
            error();                                            6
    }                                                           7
    return 0;                                                   8
}                                                               9
```

Fig. 2.6. Example program to verify with DASH.



(a) Initial abstraction for the program in Figure 2.6

(b) After arbitrary initial test has been run

(c) After the frontier has been extended

Fig. 2.7. Stages of the abstraction when the initial and a subsequent test are run.

Output: (ϕ, A), where ϕ is the combined constraint to the target region and A is the assignment of pointers at the frontier

procedure `GetCombinedConstraint`(τ, P, C)
  $(R_{k-1}, R_k) :=$ Frontier(τ, C)
  $I_{frontier} :=$ inputs I such that $\exists (t, I) \in C : t \cap R_{k-1} \neq \emptyset$
  $(\phi, A) :=$ `GatherConstraints`$(I_{frontier}, \tau, P)$
  return (ϕ, A)
end

Fig. 2.8. The procedure for constructing the combined constraint to the target region.

For our example let us assume that on the line 2 in Fig. 2.6 the variable `x` is initialized to a value of 15. The set C is initialized to contain this random test, which is a pair $((2, 3, 8), (15))$. This is visualized in Fig. 2.7(a) with the edges that are covered by the trace being wavy. Note that for brevity, in this example we use a sequence of program locations $(2, 3, 8)$ in place of a sequence of full states. The concrete states are recoverable from the input values.

After initialization, the first iteration of DASH's main loop starts by finding the abstract error trace $\tau = (2, 3, 4, 5, 6)$, also shown in Fig. 2.7(b). The call to the procedure `GetAbstractTrace` in Fig. 2.5 always returns an abstract error trace such that after the first region *not visited* by a test from C no subsequent region has been visited either. If any abstract error trace exists, we can be sure that there is one that satisfies this requirement because any abstract error trace can be transformed to fulfill it. Given an arbitrary abstract error trace $\tau' = (R_0, R_1, \ldots, R_n)$ let R_{k-1} be the last region that contains a state $s_{i-1} \in C$ and let $(s_0, s_1, \ldots, s_{i-1})$ be the execution trace leading up to s_{i-1}. Now, the desired abstract error trace is $\tau'' = (R'_0, R'_1, \ldots, R'_{i-1}, R_i, \ldots, R_m)$, such that $(R_i, \ldots, R_m) = (R_k, \ldots, R_n)$ and for each $h \in [0 \ldots i-1]$ it holds that $s_h \in R'_h$.

After finding the abstract error trace, DASH will try to extend the *frontier* of the abstract error trace τ. Frontier(τ, C) is a pair of adjacent regions (R_{k-1}, R_k) such that R_{k-1} has been visited by a test in C and R_k has not. On the first iteration of our example the frontier is $(3, 4)$.

First the procedure `GetCombinedConstraint` in Fig. 2.8 is called to construct a constraint ϕ that is satisfiable if and only if there exists a sequence of inputs that will cause the program to follow the abstract error trace up to the frontier and across it. To construct ϕ, `GetCombinedConstraint` selects from C a sequence of inputs $I_{frontier}$ that are known to take the program to the frontier. In our example the only test that has a state in the frontier region R_{k-1} is the one with the input sequence (15), which is therefore selected.

To create the constraint we call the procedure `GatherConstraints`, which uses techniques similar to those used in dynamic symbolic execution (see Section 2.1) to gather a *path constraint* of the program's execution to and over the frontier. In our example the statements executed are `x = input` and `x > 20`. The path constraint ϕ_{path} for these is $(x_1 = input_1) \wedge (x_1 > 20)$. Note that x_1 is a variable that represents the first *value* assigned to the program's variable `x`. Variables are versioned so that on every assignment the version number of the variable being assigned to is incremented, which in terms of the formula that will be sent to an SMT solver corresponds to using a fresh uninterpreted constant for the new value of the variable. In general a path constraint is the weakest predicate over the inputs of the program such that the execution will still follow the given path. The combined constraint ϕ is a conjunction of the path constraint ϕ_{path} and the constraint from the region R_k. In our example R_k has no additional constraint and, therefore, we have $\phi := \phi_{path}$. In addition to the path constraint ϕ, the procedure `GatherConstraints` records pointer aliasing information, which is returned in A (see Section 4). We can ignore this for now as the example program does not use any pointers.

Be aware that when the part of the path constraint for crossing the frontier is generated, the condition value for the branching operator may be concrete (i.e., not dependent on input values). In DSE such a situation is never encountered as only symbolic branches can be subject to test generation. In DASH this situation must be handled by evaluating the concrete constraint at the frontier. If the constraint is true then it can be omitted. If it is false then the whole path constraint will be unsatisfiable and the solver call can be skipped. Alternatively, the concrete constraint can just be added to the path constraint, letting the SMT solver handle the situation.

Once `GetCombinedConstraint` has returned, the constraint ϕ is passed to the procedure `Solve` (see Fig. 2.5), which will indicate whether ϕ is satisfiable or not. `Solve` will also return an assignment as a sequence of input values (in this case only a value for $input_1$) if it is satisfiable. Note that this is the only place in the algorithm where the SMT solver is called. In Section 4.3 we describe an extra solver that can be made here to achieve stronger refinement in some cases.

On the first iteration of our example ϕ is satisfiable and we get a satisfying sequence of inputs. Let us say the sequence is (38). This input sequence is then used to run a test with a call to the procedure `RunTest`, which executes the program with the given inputs. `RunTest` returns an *execution sequence* as a sequence t of states the execution visited and the sequence I_{all} of all the inputs given to the program. The sequence I_{all} has as a prefix the inputs given to `RunTest` followed by any that the program may have retrieved after the supplied sequence ran out. For our example program the call `RunTest`$((38), P)$ returns a pair $((2, 3, 4, 5, 8), (38))$. The new execution trace and inputs are then added to the set C, which is illustrated in Fig. 2.7(c).

Note that here we identify the states in the execution trace by the regions they belong to. For how we have implemented this mapping from states in execution traces to regions see Section 4.1.

We are now finished with the first iteration of DASH on our example program. The next iteration starts like the first one. We still find the same abstract error trace $\tau = (2, 3, 4, 5, 6)$, which is passed to `GetCombinedConstraint`. However, now

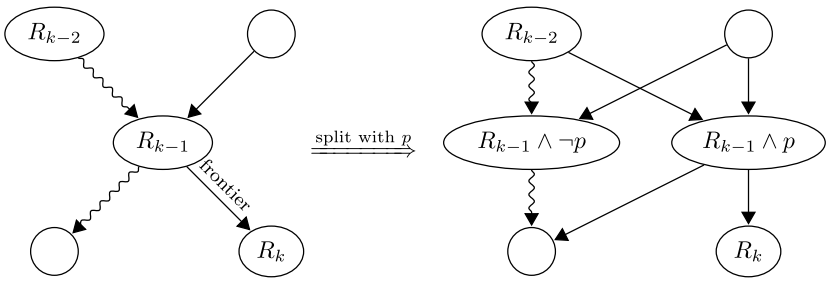
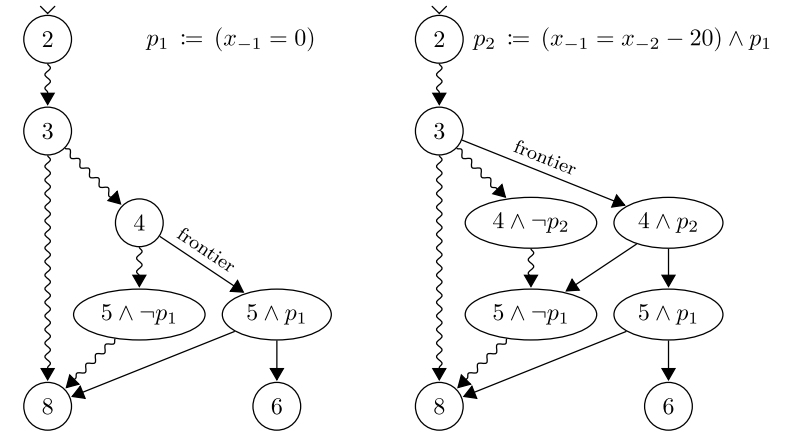


Fig. 2.9. The refinement performed by DASH.



(a) After splitting the frontier $(5, 6)$ (b) After splitting the frontier $(4, 5 \wedge p_1)$

Fig. 2.10. Stages of the abstraction as splitting continues.

the frontier has moved forward to $(5, 6)$ and the resulting constraint is $\phi := (x_1 = input_1) \wedge (x_1 > 20) \wedge (x_2 = x_1 - 20) \wedge (x_2 = 0)$. This constraint is unsatisfiable and return value from `Solve(`ϕ`)` will indicate this.

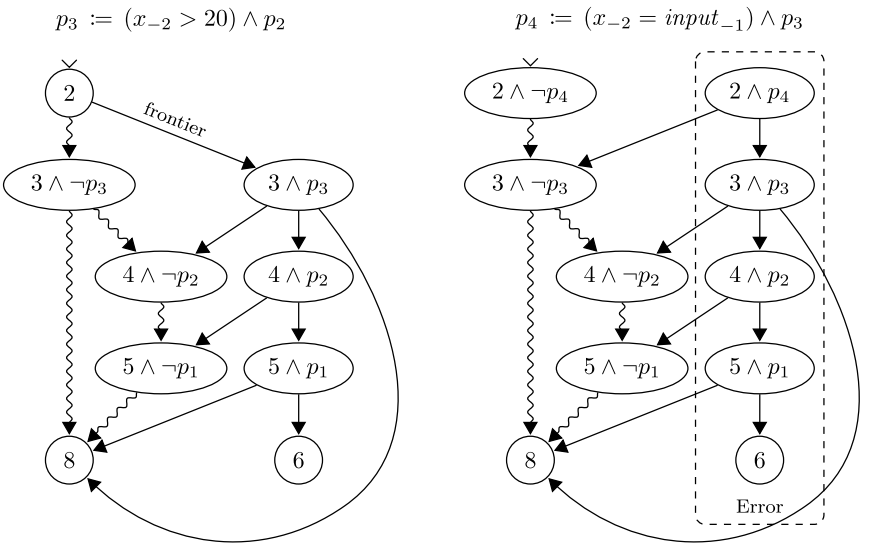
The constraint ϕ being unsatisfiable means that there cannot exist a sequence of input values that would cause the program to follow τ over the frontier. DASH will exploit this information to refine the abstraction by *splitting* the frontier region R_{k-1} in the manner presented in Fig. 2.9. Notice how after the split, the region $R_{k-1} \wedge \neg p$ is the one visited by the execution trace and the edge from it to the region R_k has been eliminated. To perform this split DASH needs a predicate p such that no state $s \in R_{k-1}$ reachable by executions following τ for which p does not hold has a transition $s \to s'$ such that $s' \in R_k$. One example of such a predicate is the *weakest precondition* [26], which is defined as the weakest predicate over the program's variables such that after executing a statement some postcondition holds. In this context the postcondition is that specified by the region R_k. While we will use weakest preconditions while going through the current example, the actual algorithm uses a slightly different type of predicate which we will describe in Section 2.3.1. Also note that due to the slightly stronger predicate that would actually be used, the region $(R_{k-1} \wedge p)$ still has both outgoing edges present in Fig. 2.9.

To produce the splitting predicate DASH calls the procedure `RefinePred`. For our example the statement associated with the frontier is `x == 0` and because the region across the frontier has no additional predicate, the weakest precondition p is simply $(x_{-1} = 0)$. The index is -1 because negative indices are used for symbolic values in the splitting predicates to differentiate them from ones in path constraints.

The predicate p is passed along to the procedure `SplitFrontier`, which performs the splitting operation illustrated in Fig. 2.9. For the pseudocode of `SplitFrontier` see Beckman et al. [18]. The region R_{k-1} is removed from the abstraction and replaced by two new regions, $(R_{k-1} \wedge p)$ and $(R_{k-1} \wedge \neg p)$. All transitions for R_{k-1} that were in $\to_\simeq$ are replicated for the new regions except the transition $(R_{k-1} \wedge \neg p) \to_\simeq R_k$, which is the one we want to eliminate. The new abstraction and its transition relation, shown in Fig. 2.10(a), are returned and the main DASH algorithm overwrites the previous versions with the refined ones. In the figure the predicate has been renamed to p_1. This concludes the second iteration of DASH.

On the third iteration the abstract error trace found in the abstraction (see Fig. 2.10(a)) is $(2, 3, 4, 5 \wedge p_1, 6)$ and the frontier examined is $(4, 5 \wedge p_1)$. The only input sequence found in C that will take the program to the frontier region 4 is still (38).

For this frontier the call to `GatherConstraints` in `GetCombinedConstraint` will also have to consider the predicate p_1 in the region over the frontier. The constraint returned will be $\phi := (x_1 = input_1) \wedge (x_1 > 20) \wedge (x_2 = x_1 - 20) \wedge (x_{-1} = x_2) \wedge (x_{-1} = 0)$, where the path constraint part is $(x_1 = input_1) \wedge (x_1 > 20) \wedge (x_2 = x_1 - 20)$ and the constraint from the re-



(a) After splitting the frontier $(3, 4 \wedge p_2)$ (b) Final abstraction for the example program

Fig. 2.11. The final stages of the abstraction where splitting separates the error region from the reachable part of the program.

gion $5 \wedge p_1$ is $(x_{-1} = 0)$. The two parts have been "glued" together with the constraint $(x_{-1} = x_2)$, which links the symbolic value of `x` in p_1 with its last symbolic value in the path constraint part.

The constraint is effectively the same as on the previous iteration and a call to `Solve(`ϕ`)` will again return a value indicating it is unsatisfiable. The procedure `RefinePred` now has to create a weakest precondition for the statement `x = x - 20` and the postcondition $(x_{-1} = 0)$. The resulting predicate is $p_2 := (x_{-1} = x_{-2} - 20) \wedge (x_{-1} = 0)$. This predicate is used to split the frontier region 4 in the abstraction, the result of which is shown in Fig. 2.10(b).

On the fourth iteration the abstract error trace in the current abstraction (see Fig. 2.10(b)) is $(2, 3, 4 \wedge p_2, 5 \wedge p_1, 6)$ and its frontier is $(3, 4 \wedge p_2)$. This time there are two input sequences in C that will take the program to the frontier, (38) and, from the initial test, (15). Assume we select the input sequence (15). The resulting constraint is $\phi := (x_1 = input_1) \wedge (x_1 > 20) \wedge (x_{-2} = x_1) \wedge (x_{-1} = x_{-2} - 20) \wedge (x_{-1} = 0)$, which is again unsatisfiable.

The procedure `RefinePred` will then be called to create a weakest precondition for the statement `x > 20` and the postcondition p_2. For these we get the predicate $p_3 := (x_{-2} > 20) \wedge p_2$, which is then used to split region 3. The resulting abstraction can be seen in Fig. 2.11(a). Note that we have written predicate p_3 as a conjunction with the postcondition it was constructed for.

The final iteration proceeds largely as before. For the frontier $(2, 3 \wedge p_3)$ the constraint $(x_1 = input_1) \wedge (x_{-2} = x_1) \wedge (x_{-2} > 20) \wedge (x_{-1} = x_{-2} - 20) \wedge (x_{-1} = 0)$ is created. It is again unsatisfiable. From the statement `x = input` and the postcondition p_3 we get the weakest precondition $p_4 := (x_{-2} = input_{-1}) \wedge p_3$.

The final version of the abstraction, which has been refined with p_4 is shown in Fig. 2.11(b). The regions that have a path to the error region 6 have been boxed. Notice that the region containing the initial state (marked with the wedge above it) no longer has such a path. Because of this, at the beginning of the next iteration Dash will not find an abstract error trace. This proves that there is no set of inputs that would cause the program to encounter the error and Dash will return a pair of ("pass", $\Sigma_{\simeq}$).

2.3.1. Suitable predicates

To do the refinement outlined in Fig. 2.9 we need to have a predicate p such that having no edge from $(R_{k-1} \wedge \neg p)$ to R_k is sound, which means that the predicate must not be too strong. As a trivial example if we use $p = \bot$ as the predicate then all possible concrete traces would belong to $R_{k-1} \wedge \neg p$ and removing the edge to R_k would not be sound. On the other hand the predicate p must not be too weak for Dash to make progress. For example, using $p = \top$ as the predicate would cause all concrete traces to belong to $R_{k-1} \wedge p$ and the frontier for the next iteration to be effectively the same. To capture these requirements for predicates we use the following definition of a *suitable predicate* [18].

Definition 1 *(Suitable predicate).* Let τ be an abstract error trace and let (R_{k-1}, R_k) be its frontier. A predicate p is said to be suitable with respect to τ only if all possible concrete states obtained by executing τ up to the frontier belong to the region $R_{k-1} \wedge \neg p$, and if there is no transition from any state in $R_{k-1} \wedge \neg p$ to a state in R_k.

When the predicate used for splitting (see Fig. 2.9) is a suitable predicate the DASH algorithm is sound. A suitable predicate also ensures that DASH makes progress in the sense that on every iteration the frontier moves forward along the abstract error trace or concrete states are removed from a region in the abstract error trace. For detailed proofs of soundness and progress see [18].

Next we outline how the procedure `RefinePred` can construct predicates that satisfy Definition 1. The construction is based on *weakest preconditions* [26]. We will explain this method with example statements from our program model. For a list of statement types and their semantics refer to Table 2.3. The following description has been adapted from [18].

Given a statement $op \in Stmts$ and a predicate ϕ (also called the postcondition), the weakest precondition $\mathrm{WP}(op, \phi)$ is the weakest predicate whose truth before op's execution implies ϕ after op is executed. For example, consider the statement op = "`x = x + 1`" and postcondition $\phi = (x_{-1} < 7)$. The weakest precondition for these is $\mathrm{WP}(op, \phi) = (x_{-1} = x_{-2} + 1) \wedge (x_{-1} < 7)$.

A weakest precondition p constructed for an abstract error trace τ and its frontier (R_{k-1}, R_k) is also a suitable predicate. Because DASH only splits the frontier when no inputs exist to force the execution across the frontier then no concrete state reachable by executing τ to the frontier belongs to $R_{k-1} \wedge p$. If there was such a state then the weakest precondition being true would imply that the execution would cross the frontier. Also, if there was a transition from any state in $R_{k-1} \wedge \neg p$ to a state in R_k then p would not be the weakest precondition because a precondition that also includes the concrete state in question would be weaker.

In the presence of memory and pointers constructing weakest preconditions may not be as straightforward as in the example above. For example, if we have a statement o_1 = "`x = load p`" together with some postcondition ϕ we could have $\mathrm{WP}(o_1, \phi) = (x_{-1} = m) \wedge \phi$, where m is used to represent the memory pointed at by `p`. Now consider using this weakest precondition as the postcondition for another statement o_2 = "`store 5 q`". Because the result of the store depends on whether `p` and `q` are equal the weakest precondition must handle both cases:

$$\begin{aligned}\mathrm{WP}(o_2, \mathrm{WP}(o_1, \phi)) &= ((p_{-1} = q_{-1}) \wedge ((m = 5) \wedge (x_{-1} = m) \wedge \phi)) \vee \\ &= ((p_{-1} \neq q_{-1}) \wedge ((x_{-1} = m) \wedge \phi))\end{aligned}$$

The number of disjuncts in the predicate scales exponentially with the number of memory locations referenced by the postcondition. In the general case if we have k memory locations then we must handle 2^k different aliasings. A solution to this problem is to use aliasing information gathered from a concrete test run. For this purpose we define the projection of a weakest precondition down to a specific aliasing constraint α as follows [18]:

$$\mathrm{WP}{\downarrow_\alpha}(op, \phi) = \alpha \wedge \mathrm{WP}(op, \phi)$$

This projected weakest precondition can be constructed to only consider one aliasing situation thus avoiding the exponential number of disjuncts. In the example above if the concrete aliasing situation was that `q` and `p` are equal, then we would have:

$$\mathrm{WP}{\downarrow_\alpha}(o_2, \mathrm{WP}{\downarrow_\alpha}(o_1, \phi)) = (p_{-1} = q_{-1}) \wedge ((m = 5) \wedge (x_{-1} = m) \wedge \phi)$$

Using this projected weakest precondition and adding a term to handle the remaining aliasings we get a predicate that can be used for refinement as the predicate returned by the procedure `RefinePred`:

$$\mathrm{WP}_\alpha(op, \phi) = \neg\alpha \vee \mathrm{WP}{\downarrow_\alpha}(op, \phi)$$

Consider a predicate p constructed with WP_α in the context of the splitting operation in Fig. 2.9. The region $R_{k-1} \wedge \neg p$ contains only states where the aliasing constraint α is true. Other aliasing situations belong to the region $R_{k-1} \wedge p$ and the predicate p does not constrain their subsequent control flow.

Theorem 1. *The predicate* $\mathrm{WP}_\alpha(op, \phi)$ *is a suitable predicate when it is returned from the procedure* `RefinePred`.

For a proof see [18].

3. Suitable predicate construction rules for LLVM

We will now provide a detailed set of rules for constructing suitable predicates for the statements in our program model (see Table 2.3). In our implementation the suitable predicates are constructed using temporary variables (instead of substitution) to represent assignments. For this purpose we associate each predicate with a *variable version map* V which maps each variable to an integer representing its current version. For example, if we have a variable `x` then its current version is $x_{V(x)}$. When variables are assigned to, the predicate is written to assign to the current version of the variable, after which the version number of the variable is decremented. This ensures that any further suitable predicates constructed with this predicate as their postcondition will refer to the version of the variable that exists before the assignment. Initially all variables are mapped to the version number -1. This way these negative variable version numbers do not get mixed up with the positive variable version numbers created during dynamic symbolic execution. Be aware that the construction is somewhat subtle due to how the predicates are created in reverse program order.

Table 3.12
Suitable predicate construction rules for arithmetic and guard statements.

Statement	Actions
$x = y$ <op> z	$p' = p \wedge (x_{V(x)} = y_{V(y)}$ <op> $z_{V(z)})$ $V' = V[x \mapsto V(x) - 1]$
$x =$ input	$p' = p \wedge (x_{V(x)} = input_{V(input)})$ $V' = V[x \mapsto V(x) - 1][input \mapsto V(input) - 1]$
$x = y$	$p' = p \wedge (x_{V(x)} = y_{V(y)})$ $V' = V[x \mapsto V(x) - 1]$
(x <comp> y)	$p' = p \wedge (x_{V(x)}$ <comp> $y_{V(y)})$
! (x <comp> y)	$p' = p \wedge \neg(x_{V(x)}$ <comp> $y_{V(y)})$

Arrays will be handled by encoding `store` and `load` statements to the *store* and *select* functions in the functional theory of arrays [27] available in many SMT solvers. The *store* function takes as parameters an array constant, an index and a value, and produces a new array constant with the new value at the index. The *select* function takes as parameters an array constant and an index, and returns the value at that index. We associate one array with each *base pointer*, which are any pointers created by an `alloc` statement. Derived pointers created with `gep` statements refer to the same base pointer with an appropriate offset. To simplify the construction rules we handle single variable allocations also as arrays of size one.

Consider the situation where a suitable predicate is to be constructed for the statement `store 7 p` and the predicate for the target region is $(y_{-1} = select(m_1, 1)) \wedge (x_{-1} = select(m_2, 1)) \wedge (x_{-1} \neq y_{-1})$, where m_1 and m_2 are array variables that represent memory. To construct the predicate we need to know which of these memory ranges are modified by the `store` statement. Array variables for memory are added when constructing suitable predicates for statements that read from memory. To construct suitable predicates for statements that write to memory the following information will suffice:

1. the base pointer associated with each array m_i appearing in the predicate for the target region, and
2. an equivalence relation for base pointers at the point before the statement for which we are constructing the suitable predicate.

To address the first item we introduce as part of our contributions the notion of a *mentions map* M, which is a mapping from versioned variables to array variables representing memory. The mentions map provides a mechanism for tracking which array variables may be updated by a `store` statement, which was not described in detail by Beckman et al. [18]. Each suitable predicate is associated with a separate mentions map. When we construct a new suitable predicate on top of an existing one the mentions map is inherited with the appropriate modifications. For `load` statements a new mapping from the current version of the pointer to a new array variable for the memory is introduced (or an existing mapping may be reused). Assignments to pointers also modify the mentions map by moving mentions from one pointer variable to another.

The equivalence relation for pointers is obtained from a concrete execution that has visited the frontier region. During the test run we record the values of all assignments to pointers. Using the recorded information we then construct a map A from all pointers to their current values at the frontier. We can check if two pointers are equivalent by checking whether they map to the same value.

In the example above the indices for the *select* functions were constant ones. However, when a suitable predicate is constructed for a `load` statement the index will be determined by any `gep` statements preceding that `load`. To represent this we introduce an *index version map* I, which works exactly like the variable version map V introduced before. The variables indexed with values from the I map are separate from those indexed with values from the V map, which is indicated with a dot above the variables (for example "$\dot{a}_{I(a)}$").

Now each suitable predicate constructed is associated with a variable version map V, an index version map I and a mentions map M. Given a statement op and the assignment A of pointers at the frontier, the suitable predicate $p' = \mathrm{WP}_\alpha(op, p)$ can be constructed by following the rules in Table 3.12 (for arithmetic and guard statements) or Table 3.13 (for statements on pointers and memory). These rules, which provide a detailed description of how suitable predicates can be constructed, are one of our contributions. Note that the statement x = y appears in both tables, but the rule in Table 3.13 should only be used when x and y are pointers. Each rule describes how a new suitable predicate p' is constructed. The rules also produce new versions of the version and the mentions maps as V', I' and M'. These are stored with p' for constructing further suitable predicates where p' is the postcondition. For updating the maps the rules employ the notation introduced at the start of Section 2.2. In particular note the previously unused notation $M[x \not\mapsto]$ for removing the entry for x from the map M.

The rule for statements of the form x = y <op> z is straightforward: a new constraint encoding the assignment is added to p' and the variable version number of x is decremented in V'. No special handling for pointers is needed because we assume our programs do not contain general pointer arithmetic. Note that while some additional pointer arithmetic could be supported by expanding this rule, to support general pointer arithmetic the straightforward tracking of a single

Table 3.13
Suitable predicate construction for statements involving pointers and memory.

Statement	Actions
$a = b$	$p' = p \wedge (a_{V(a)} = b_{V(b)}) \wedge (\dot{a}_{I(a)} = \dot{b}_{I(b)})$ $V' = V[a \mapsto V(a) - 1]$ $I' = I[a \mapsto I(a) - 1]$ If M contains $a_{V(a)}$ then also do: $M' = M[a_{V(a)} \not\mapsto][b_{V(b)} \mapsto M(a_{V(a)})]$
$x = \texttt{load}\ a$	Let $m = \begin{cases} M(a_{V(a)}), \text{ if } M \text{ contains } a_{V(a)} \\ \text{a new array otherwise} \end{cases}$ $p' = p \wedge (x_{V(x)} = select(m, \dot{a}_{I(a)})$ $M' = M[a_{V(a)} \mapsto m]$ $V' = V[x \mapsto V(x) - 1]$
$\texttt{store}\ x\ a$	Let $k = \{b \mid b \in \text{keys}(M) \text{ and } A(a_{V(a)}) = A(b)\}$ $N(b) =$ a new array for key b $\alpha = \bigwedge\{a_{V(a)} = b \mid b \in k\} \wedge$ $\bigwedge\{a_{V(a)} \neq b \mid b \in \text{keys}(M),\ b \notin k\}$ in $p' = \left(\bigwedge\{N(b) = store(M(b), \dot{a}_{I(a)}, x_{V(x)}) \mid b \in k\} \wedge p\right) \vee \neg\alpha$ $M' = M[b_1 \mapsto N(b_1)] \ldots [b_n \mapsto N(b_n)]$ where $b_i \in k$
$a = \texttt{allocate}$	Let $s =$ an integer not used in any previous `allocate` $p' = p \wedge (a_{V(a)} = s) \wedge (\dot{a}_{I(a)} = 0)$ $M' = M[a_{V(a)} \not\mapsto]$ $V' = V[a \mapsto V(a) - 1]$ $I' = I[a \mapsto I(a) - 1]$
$a = \texttt{gep}\ b\ x$	$p' = p \wedge (a_{V(a)} = b_{V(b)}) \wedge (\dot{a}_{I(a)} = \dot{b}_{I(b)} + x_{V(x)})$ $M' = M[a_{V(a)} \not\mapsto][b_{V(b)} \mapsto M(a_{V(a)})]$ $V' = V[a \mapsto V(a) - 1]$ $I' = I[a \mapsto I(a) - 1]$

base pointer does not work. In LLVM it is legal to access a range of memory through an integer (interpreted as an address via the `inttoptr` instruction) as long as a pointer to that range of memory contributed to the value of the integer. This could result in situations where it is not clear what should be considered to be the base pointer tracked by the rules in this section.

Statements of the form x `= input` also encode the assignment in p' and decrement the version number of x. The input is implemented here as a variable *input*, the current version of which is assigned to x. Because each input statement should use a new input we also decrement the version number of *input* in V'.

For simple assignments the rule in Table 3.12 is similar to the one for assignment of binary expressions in how p' and V' are updated. However, now the assignment may be from a pointer to another, which is handled by the rule in Table 3.13. To explain this handling we will first describe how the statements of the form x `= load` a are handled. A `load` statement creates a new array variable m to represent the memory range from which the read happens. This array variable m is stored in the mentions map M' so that the current version of the pointer $a_{V(a)}$ maps to m. If M already contains an entry for $a_{V(a)}$ then the variable representing the memory is reused as m and the mentions map is not modified. The mapping from $a_{V(a)}$ to m is added so that we can resolve for any subsequent `store` statements which array variables the added constraints should refer to. Now the handling of a simple assignment of the form a `=` b, where a and b are pointers, can be understood. If a `load` has mentioned the current version of the pointer a, then the assigned value $b_{V(b)}$ becomes the pointer value through which the array variable is mentioned. To indicate this the mapping for $a_{V(a)}$ is removed from M' and a new mapping from $b_{V(b)}$ to $M(a_{V(a)})$ is added. Finally the index variable must be handled: a new constraint is added that encodes the assignment of the index variable of b to the index variable of a and the index version number of a is decremented in I'.

As explained above, for statements of the form x `= load` a an array variable m is selected to represent the memory. The new constraint p' encodes assigning the value selected from m at index $\dot{a}_{I(a)}$ to $x_{V(x)}$. The version number of x is decremented in V' due to the modification from the assignment. The mentions map M' is also modified to record that m is mentioned through $a_{V(a)}$.

The rule for statements of the form `store` x a combines information from the mentions map M and the pointer assignments A at the point before the statement. First an aliasing constraint α is constructed by examining all pointer values currently in M. For each mentioning pointer value b we create a constraint of the form $(a_{V(a)} = b)$ if the pointers have the same value, i.e., $A(a_{V(a)})$ is equal to $A(b)$. If the pointers are not equal we instead create the constraint $(a_{V(a)} \neq b)$. The aliasing constraint α is a conjunction of all the created equalities and inequalities. Using α the constraint p' is then created. It is in this constraint that the form of the suitable predicate WP_α described in Section 2.3.1 can be seen. The created constraint is true if either α does not hold or if the weakest precondition from the `store` statement in the current aliasing situation holds. The weakest precondition part of the constraint is constructed by examining all mentioned arrays

m in M and creating a constraint of the form $(m = store(m', \dot{a}_{I(a)}, x_{V(x)}))$, where m' is a new array variable, if the pointer value b through which m is mentioned is equal to $a_{V(a)}$, i.e., $A(a_{V(a)})$ is equal to $A(b)$. Now, the weakest precondition part is a conjunction of p and any new constraints created for mentioned array variables. Finally, mentions that pointed to an array m are moved to point to the corresponding new array m', because all subsequent modifications (i.e., previous in the program order) must affect the new version.

The rule for statements of the form `a = allocate` selects a new integer s that has not been used by any previous `allocate` statement. The modifications to p' and V' from assigning s to $a_{V(a)}$ are the same as in the other assignment statements described above. However, an allocation always assigns to a pointer and as such the mentions map might also be modified. If $a_{V(a)}$ is in the mentions map, then the mapping can be removed because this `allocate` statement is the first point at which the value s is seen and as such cannot appear in any `store` statement that might be handled later (i.e., earlier in the program order). In addition to assigning the new address, an `allocate` statement initializes the index associated with the pointer a to 0 with the appropriate modification to I'.

Finally, the rules for the guard statements simply add the appropriate guard constraint to p'. These statements do not modify variable versions or the mentions map as they represent branching conditions and have no effect on the state of variables or memory. Note that the construction rules for guard statements on pointers are not shown here, but these are a simple extensions of the existing rules with the observation that for comparing equality the value of a pointer a can be taken to be $a_{V(a)} + \dot{a}_{I(a)}$.

To illustrate how the presented construction rules for statements involving pointers operate, consider the following program fragment written with statements from our program model:

```
a = allocate 5                                  1
b = gep a 2                                     2
a = b                                           3
store 7 b                                       4
x = load a                                      5
```

Assume a concrete execution has previously collected an assignment of pointers on each line. Given a predicate p^0 for the target region after line 5, the suitable predicate for each statement can be constructed by applying the rules in Table 3.13. The statements are processed starting with line 5 and the target predicate for the next statement processed is the suitable predicate from the previous statement. Assume that for the first statement the variable and index version maps and the mentions map are at their initial values. These get updated as the statements are processed.

For line 5 a new array constant m is created to represent the region of memory that is read and the mentions map is updated to $M[a_{-1} \mapsto m]$, which records that modifications to the memory pointed by a should be done through m. The variable version for x is decremented. The suitable predicate is as follows:

$$p^1 = p^0 \wedge (x_{-1} = select(m, \dot{a}_{-1}))$$

For line 4 the keys of the mentions map is examined to find pointers that are equal to b. The pointer assignment from a concrete execution will indicate that a is equal to b. A new array constant m' is created to represent the updated version m, which is the value for a in the mentions map. The mentions map is updated with $M[a_{-1} \mapsto m']$ to point to the new array constant. The suitable predicate is created with $(b_{-1} = a_{-1})$ as the aliasing constraint α:

$$p^2 = ((m' = store(m, \dot{b}, 7)) \wedge p^1) \vee \neg(b_{-1} = a_{-1})$$

For line 3 the mentions map is updated with $M[a_{-1} \not\mapsto][b_{-1} \mapsto M(a_{-1})]$ to move the entry for a to b. Note that the entries in the mentions map move in the opposite direction compared to the flow of data in concrete execution. The variable version and the index version for a is decremented. Both are now at -2 as we will see for the next statement. The suitable predicate encodes the assignment:

$$p^3 = p^2 \wedge (a_{-1} = b_{-1}) \wedge (\dot{a}_{-1} = \dot{b}_{-1})$$

For the `gep` instruction on line 2 the mentions map is updated similarly to the simple assignment with $M[b_{-1} \not\mapsto][a_{-2} \mapsto M(b_{-1})]$. The variable version and the index version for b is decremented. In the suitable predicate the value indexed with is added to the constraint for the index variables:

$$p^4 = p^3 \wedge (b_{-1} = a_{-2}) \wedge (\dot{b}_{-1} = \dot{a}_{-2} + 2)$$

For line 1 the mentions map entry for a_{-2} is cleared with $M[a_{-2} \not\mapsto]$. The variable version and the index version for a is decremented. The suitable predicate sets the initial values for a_{-2} and $\dot{a}_{-2}$:

$$p^5 = p^4 \wedge (a_{-2} = 0) \wedge (\dot{a}_{-2} = 0)$$

The suitable predicate p^5 is now a suitable predicate for the program fragment.

A situation not handled by the rules is when pointers are stored to or loaded from memory. Proper handling of these situations would involve also having the mentions map support entries where an array variable is mentioned from an

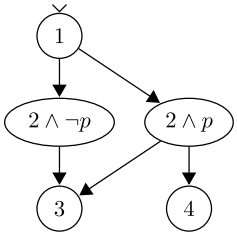


Fig. 4.14. Example abstraction for mapping traces.

element of another array variable that represents memory. We have left support for loading and storing pointers for further work.

In the DASH algorithm the constructed suitable predicates are combined with path constraints generated by dynamic symbolic execution. As was already mentioned in the running example in Section 2.3, if we have a path constraint ϕ_{path} and a predicate p for a target state then p can be added to the end of the path constraint with the help of some additional "glue" constraints. The additional constraints must ensure the following:

1. The earliest versions of variables in p must be equivalent to their latest versions found in the path constraint. If they are not found in the path constraint (i.e., they are not symbolic) then they must be assigned concrete values obtained from the test execution at the frontier.
2. Array variables for memory that are still in the mentions map M must be equivalent to their versions at the frontier.

Because concrete values from the execution are needed and storing the whole history of concrete values for all test executions could be memory intensive we have in our implementation chosen to re-execute the program to the frontier to obtain the necessary values. In the pseudocode of the DASH algorithm this re-execution is part of the `Gather-Constraints` procedure.

Programs in the LLVM IR are structured into basic blocks and control flow is transferred always to the beginning of a basic block. Non-terminator instructions do not transfer control flow (i.e., the next instruction listed in the block is the next to be executed), with the exception of the `call` instruction. In this work we omit support for programs with procedures and with this restriction we can treat reachability in the program on the level of basic blocks. Thus nodes in DASH's abstraction will represent basic blocks instead of single instructions.

4. Implementation

We have implemented the DASH algorithm as a modification to the Lime Concolic Tester (LCT) [19–21], which is an open source dynamic symbolic execution tool for C and Java programs. Our tool LCTD extends the LLVM based C support in LCT. LCT uses a model where transient client processes connect to a persistent server process to execute solver and test execution tasks. For DASH the client implements test execution and frontier extension, while the server maintains the abstraction and is responsible for the top-level control flow of the algorithm. For a full explanation of how DASH was implemented on LCT see [28].

The rest of this section presents solutions to several issues we encountered while implementing DASH.

4.1. Mapping traces to regions

In the pseudocode of the `GetCombinedConstraint` procedure in Fig. 2.8 the process of selecting a test that visits the frontier is presented as simply choosing a trace that intersects the frontier region. However, in an actual implementation of the algorithm it is not immediately clear what is the best way to decide to which regions the states of a concrete trace belong to. When a test execution enters a program location which due to splitting corresponds to multiple regions, we have to have a way to evaluate the predicates to decide which region the concrete state belongs to. A trivial way of doing this is to evaluate the relevant suitable predicates at each step of a test execution. In our implementation this would have meant the client would request predicates from the server during execution. However, it turns out that as long as (1) we know the region of the last state of a concrete trace and (2) we can evaluate the pointer constraints in each suitable predicate, then we can also resolve the regions of the previous states. For example, consider the abstraction shown in Fig. 4.14 and assume we wish to determine which regions a test with the trace $(1, 2, 3)$ belongs to. While regions 1 and 3 are clear, a selection must be made between $(2 \wedge p)$ and $(2 \wedge \neg p)$. Recall that p is a weakest precondition for the execution proceeding to region 4 projected down to a set of pointer constraints. If the test satisfies the pointer constraints then p must be false because otherwise the execution would have proceeded to region 4. With this we can evaluate p and make the selection between $(2 \wedge p)$ and $(2 \wedge \neg p)$. We will now present this selection process more formally. Assume we have the following:

- t_1 and t_2 as adjacent states in a concrete execution trace,

- R' as the abstract region t_2 belongs to and
- A as the assignment of pointer values at t_1.

We wish to determine the region that t_1 belongs to. If the region that was originally created for the program location of t_1 has not been split, then we know that t_1 belongs to that. Therefore we focus on the case that the region has been split. Now we have a set of candidate regions $R_0, R_1, \ldots, R_n$. Each region is associated with a conjunction of suitable predicates or their negations. Each predicate is of the form:

$$p = \neg\alpha \vee \mathrm{WP}{\downarrow_\alpha}(op, R_{target})$$

We can evaluate α using the pointer assignments in A. If α was false then we know that p is true and we are done. Otherwise, now that we know α is true then for $\mathrm{WP}{\downarrow_\alpha}(op, R_{target})$ to be true the state t_2 must belong to the region R_{target}. More succinctly p is true if and only if the following is true:

$$\neg\alpha(A) \vee (t_2 \in R_{target})$$

To evaluate $t_2 \in R_{target}$ we recognize that the region R' which t_2 belongs to and the region R_{target} are related in one of the following ways:

- The regions correspond to different program locations and thus $R' \cap R_{target} = \emptyset$.
- $R' = R_{target}$.
- Region R' is a result of splitting (or multiple splits of) R_{target} and thus $R' \subseteq R_{target}$.
- The regions are on different sides of a split and thus $R' \cap R_{target} = \emptyset$.

Note that R_{target} cannot be a result of splitting R' because R' is a region currently in the abstraction and therefore cannot have been split. So now it holds that either $R' \subseteq R_{target}$ or $R' \cap R_{target} = \emptyset$. Both cannot be true because we know that R' is not empty. If $R' \subseteq R_{target}$ then because $t_2 \in R'$ we also have $t_2 \in R_{target}$. Alternatively if $R' \cap R_{target} = \emptyset$ then we know that $t_2 \notin R_{target}$.

Using the procedure outlined above we can evaluate any suitable predicate associated with the candidate regions $R_0, R_1, \ldots, R_n$ in the state t_1 and, therefore, we can decide which region t_1 belongs to. If we have the whole concrete trace $T = (s_0, \ldots, s_{k-1}, s_k)$, know the pointer assignments for all states in the trace and know the region for s_k, we can now repeat this procedure to resolve s_{k-1} through s_0.

To see how the method described above works in practice consider the example abstraction in Fig. 4.14. Assume we have some execution trace that visits the program location 2 and that we wish to know whether the state belongs to $(2 \wedge p)$ or $(2 \wedge \neg p)$. First, we evaluate the aliasing constraint α from p using the pointer assignments at the program location 2. If α is false then we know that p is true and the correct region must be $(2 \wedge p)$. However, if α is true we must examine the subsequent control flow of the execution trace. For p to be true when α is true, the next program location must be 4. Since one of our assumptions was that the exact region of the next state in the execution trace is known, we have enough information to evaluate p. This gives the region for a single state and the reasoning can be iterated from the end of the trace to the beginning to map all states of the trace to their regions.

In our implementation the regions in the abstraction are stored as a forest, where each tree represents the regions for a single program location. The leaves in the trees are the regions currently in the abstraction while internal nodes represent regions which have been split. In the forest $R' \subseteq R_{target}$ holds if and only if the node for R' is in the subtree rooted at R_{target}. Therefore, with the regions stored as a forest we can check how any two regions are related by checking whether one is an ancestor of the other.

Our approach differs from the one taken in the YOGI tool [29], which stores the whole concrete state at each program location. We only store the concrete values for pointer variables and avoid storing other concrete state by exploiting our knowledge of the execution's subsequent control flow. Similarly to the YOGI tool, our tool LCTD only stores the changed pointers from one concrete state to another.

4.2. Handling phi instructions

Programs in the LLVM IR may contain `phi` instructions, which select a value to assign based on which control flow edge the basic block containing the `phi` instruction is entered. `phi` instructions are used to represent the program's control flow graph (CFG) in static single assignment (SSA) form, where each variable in the program is assigned in exactly one program location. For example, the body of a do-while loop could have a `phi` instruction that selects either the initial value if the body is entered for the first time or an updated value if the loop has already been executed once.

In LLVM `phi` instructions are always before any other types of instructions in a basic block (there may be multiple `phi` instructions). Each `phi` instruction in a basic block contains a list of *source-value pairs*, which specify a corresponding value (i.e., a variable or a constant) for every predecessor basic block. The following is a `phi` instruction presented similarly to statements in our program model:

$$r \texttt{ = phi } [b_1 \texttt{ } \mapsto \texttt{ } v_1]\texttt{, } [b_2 \texttt{ } \mapsto \texttt{ } v_2]\texttt{, ..., } [b_n \texttt{ } \mapsto \texttt{ } v_n]$$

The semantics are such that if the previous basic block was b_i where $i \in [1 \ldots n]$ then r gets the value v_i.

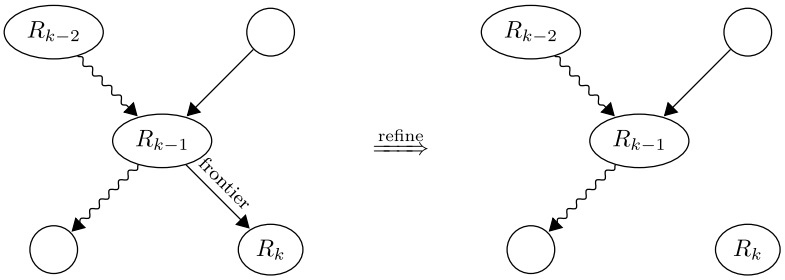


Fig. 4.15. Refinement when R_k is unsatisfiable.

In dynamic symbolic execution whenever a `phi` is encountered the source basic block is always known and therefore `phi` instructions simply reduce to simple assignments. For suitable predicate construction this is not the case. As a suitable predicate can be used in multiple contexts we cannot simplify `phi` instructions to simple assignments at construction time. However, the simplification can be done if we defer the construction as follows.

Assume we are constructing a suitable predicate $\mathrm{WP}_\alpha(op_{pred}, \mathrm{WP}_\alpha(op_{phi}, \phi))$, where op_{pred} is an instruction belonging to the previous basic block and op_{phi} is a `phi`. Because op_{pred} gives us the previous basic block we can now simplify the `phi` instruction to a simple assignment. This observation can be extended to situations with multiple sequential `phi` instructions. Now we can construct suitable predicates whenever the first (in terms of the program's execution order) instruction is not a `phi`. However, if we have a suitable predicate of the form $\mathrm{WP}_\alpha(op_{phi}, \phi)$ we are still stuck. To handle these we can observe that in Dash suitable predicates are always solved in the context of a specific execution path. Therefore, when we have a suitable predicate of the form $\mathrm{WP}_\alpha(op_{phi}, \phi)$ we can use this path information to finish the simplification for the solver call in question.

In conclusion, we can now always simplify `phi` instructions before we need to solve them due to the following: (1) `phi` instructions used to construct suitable predicates that are used as postconditions for other suitable predicates can be immediately simplified, and (2) the `phi` instruction in the outermost WP_α application can be simplified when combining with the path constraint.

There is one additional complication to constructing suitable predicates for `phi` instructions: in LLVM all `phi` instructions at the beginning of a basic block are executed atomically [30]. In other words a value assigned by a `phi` instruction is visible only after all `phi` instructions in a basic block have been executed. These semantics must be reflected in the suitable predicate construction. Consider a `phi` instruction that has a variable `x` in one of its source-value pairs. Assuming x_{-k} is the current version of `x` before the constraints for the `phi` instructions have been created, then the correct version to use in the constraint is as follows:

- x_{-k}, if `x` is not assigned to by another `phi` in the same basic block.
- x_{-k-1}, if `x` is assigned to by another `phi` in the same basic block. For the `phi` instruction that assigns to `x` the constraint should be written so that it assigns to x_{-k}.

4.3. Exploiting unsatisfiable regions

In the algorithm presented so far the only way to eliminate all paths to an error region is to continue refining the abstraction until the initial region is split. However, propagating the splitting all the way to the initial region can be expensive as it can involve many steps during which the constraints grow as more suitable predicates are constructed. Moreover, diamond structures in the CFG on the path from the initial region to the current frontier will during the splitting duplicate the frontier even if the code in the diamond structure is irrelevant to the verification task at hand. The duplicated frontiers would all have to be propagated to the initial region to eliminate them. The aggregate effect of this phenomenon is a combinatorial explosion as all reverse paths back from the frontier are explored through the splitting process. For an example of this see the experimental results on a program with sequenced diamond structures in Table 5.20.

To alleviate this problem we have modified how Dash attempts to extend and refine frontiers. In Fig. 2.5 the constraint passed to the solver is of the form $\phi_{path} \wedge \phi_{glue} \wedge R_k$. Observe that if the constraint R_k of the target region is unsatisfiable by itself, then no matter what the path constraint ϕ_{path} is the conjunction will still be unsatisfiable. Therefore, before attempting to generate a test to cross the frontier we make an additional solver call `Solve(`R_k`)`. If the constraint R_k is unsatisfiable, then we can apply the stronger refinement presented in Fig. 4.15 where we simply remove the frontier edge (R_{k-1}, R_k). Note that because of the way `phi` instructions are handled (see Section 4.2) the suitable predicates created for regions may be dependent on the edge a region is entered. Due to this we can only remove the frontier edge and not the whole target region.

5. Evaluation

In this section we present results from verifying a set of programs with our tool LCTD, which implements the DASH algorithm. We have evaluated LCTD with three different kinds of benchmarks: (1) programs from the 2015 Competition on Software Verification (SV-COMP), (2) a synthetic benchmark that illustrates the importance of exploiting unsatisfiable regions (see Section 4.3), and (3) synthetic benchmarks to evaluate array handling. The programs from SV-COMP are between 20 and 250 lines long. These benchmarks were run using two different versions of LCTD: for the array benchmarks we used a new version implementing the array handling support we have described in this paper, and for the rest of the benchmarks we used an older version without support for indexing arrays with the `getelementptr` instruction. Apart from the array handling the main difference between the two versions of LCTD is that the older one uses the Boolector SMT solver version 1.4 [31], while the newer one uses the Z3 SMT solver version 4.3.2 [32].

From the 2013 SV-COMP benchmark set we excluded programs that allocate memory from the heap (for example with `malloc()`). We also excluded programs that use structs due to struct member accesses compiling down to the `getelementptr` instruction, which we do not support for indexing structs. Finally, as our tool does not support procedure calls, we had to exclude some programs where not all procedures could be inlined.

One further limitation we encountered was that some of the benchmarks were such that all executions were infinite. These could be handled by implementing a bound on the depth of the test executions. However, the method by which we map test executions back to the abstraction (described in Section 4.1) requires that the region of the last state in the execution is known. This is not necessarily the case when an execution is stopped before termination. Instead of extending LCTD to support bounding the execution depth we chose to modify the benchmarks. For programs that consist of some initialization followed by an infinite loop with no code after, it is safe (i.e., reachability of error states is not altered) to replace the loop with one where a non-deterministic choice to continue is made. So in programs that have a top-level **`while`**`(1)` loop we would replace the loop with **`while`**`(lct_get_bool())`. The affected benchmarks are all the "Jain" and "Locks" programs.

The benchmarks from SV-COMP, the synthetic array benchmarks and both versions of LCTD used for them in the following experiments can be downloaded from: http://users.cse.aalto.fi/osaariki/lctd-jlap-2014/.

5.1. Experiment setup

We ran the SV-COMP benchmarks and the synthetic benchmark relating to the unsatisfiable regions on an Intel Core i5-2500 CPU @ 3.30 GHz with 8 GB of memory. The synthetic array benchmarks were run on an Intel Core 2 Quad Q9550 CPU @ 2.83 GHz with 4 GB of memory.

The SV-COMP benchmarks have varying conventions for indicating inputs and errors. We took the following steps to prepare the benchmark programs for verification:

- We identified the input variables and modified the program to initialize them with the `lct_get_*()` functions from our tool's runtime library. Some inputs were already marked with a call to `_ _VERIFIER_nondet_int()` or similar, while others were simply uninitialized variables.
- We replaced errors marked in the program with calls to the `assert` macro from the C standard library.
- To handle programs with multiple functions we added the `inline` keyword and `always_inline` attribute to all functions apart from `main`.

These modifications are such that the semantics of the benchmark was preserved.

We then compiled the prepared programs into LLVM IR, producing `.bc` files. Next we ran an optimization pass with the switches `-always-inline`, `-mem2reg` (moving variables to registers where possible) and `-lowerswitch` (lower `switch` statements to series of branches). Then we instrumented the programs with our instrumentation pass, which is implemented as a transformation pass that can be loaded into the LLVM `opt` tool. This transformation produces a `.bc` file containing the instrumented program as well as a `.bgd` file, which contains the program's CFG and simplified LLVM IR source code for initializing the abstraction on the server. Finally we compiled the instrumented `.bc` files into executables.

To run each test we started the server with the `.bgd` file of the program to verify. Once the server had started up we repeatedly ran the instrumented executable until the server reported the program to be safe or unsafe. The server then reported the following statistics:

- **Result**: Whether the program was SAFE or UNSAFE.
- **Time**: The time from the first client connection to either an error being reported or refinement of the abstraction removing all counterexamples.
- **Test runs**: The number of tests run including the initial arbitrary execution.
- **Solve runs**: The number of times DASH tried to extend a frontier.
- **SMT solver calls**: The number of calls made to the SMT solver. Each solve run makes one call if the target region's constraint is unsatisfiable and if not a second call is made for extending the frontier.
- **Unsat targets**: The number of unsatisfiable constraints encountered in target regions.

Table 5.16
Results for SV-COMP benchmarks.

Name	Result	Time (s)	Test runs	Solve runs	SMT calls	Unsat targets
Alias of return 1	SAFE	0.04	1	1	2	0
Alias of return 1.1	SAFE	0.04	1	1	2	0
Alias of return 2	SAFE	0.11	2	4	6	2
Alias of return 2.1	SAFE	0.04	1	1	2	0
Byte add safe 1	SAFE	5.91	1	78	126	30
Byte add safe 2	SAFE	5.86	1	79	127	31
Byte add unsafe	UNSAFE	0.05	2	1	2	0
GCD 1	SAFE	1.04	3	20	34	6
GCD 2	SAFE	0.52	1	11	18	4
GCD 3	SAFE	1.70	2	18	30	6
GCD 4	SAFE	2.29	1	32	55	9
Jain 1	SAFE	0.42	1	9	15	3
Jain 2	SAFE	0.45	1	9	15	3
Jain 4	SAFE	0.66	1	9	15	3
Jain 5	SAFE	0.42	1	9	15	3
Jain 6	SAFE	0.54	1	9	15	3
Jain 7	SAFE	0.53	1	9	15	3
Modulus	SAFE	2.02	2	9	15	3
Mutex lock int safe	SAFE	0.10	1	3	5	1
Mutex lock int unsafe	UNSAFE	0.01	1	0	0	0
Num conversion	SAFE	0.61	1	13	21	5
oomInt	SAFE	0.15	1	4	7	1
Parity	SAFE	1.16	1	18	29	7
Size of parameters	SAFE	0.04	1	1	2	0
Stateful check	UNSAFE	0.86	4	17	38	2
Locks 5	SAFE	3.87	6	87	148	26
Locks 6	SAFE	4.84	7	107	182	32
Locks 7	SAFE	8.63	9	174	287	61
Locks 8	SAFE	6.75	9	147	250	44
Locks 9	SAFE	12.45	11	226	373	79
Locks 10	SAFE	8.83	11	187	318	56
Locks 11	SAFE	17.48	13	278	459	97
Locks 12	SAFE	20.00	14	304	502	106
Locks 13	SAFE	12.27	14	247	420	74
Locks 14	SAFE	13.33	15	267	454	80
Locks 15	SAFE	14.67	16	287	488	86

We report the results of one verification run for each program as the time used did not vary significantly. However, in our tests we used a fixed random seed for generating the new inputs encountered during test runs. The random inputs generated affect which parts of the program are explored first and therefore have an effect on the time and iterations required for the verification task. In a comparative benchmark between verification tools this effect should be explored, but as we only wish to show the viability of our tool we chose not to. Thus the results we report only represent one data point from a distribution of possible results.

5.2. Results

The results for the SV-COMP benchmarks can be seen in Table 5.16. We will now go over some points of note in the results.

Looking at the ratio of test runs to solve runs, we can see that most programs require many more solve runs. For example, let us look at the “Byte add safe” programs number 1 and 2, which emulate a byte-wise carry adder and the specification for which is that the calculated addition matches C semantics. For these the only test run is the initial execution. After this no tests are generated and instead the abstraction is refined until the programs have been verified.

We can see two examples where a buggy version of a program is very fast to detect. In “Byte add unsafe” Dash needs only one solve run to find two values for which the emulated byte adder overflows and calculates a wrong result. A more extreme example is “Mutex lock int unsafe” where the faulty behavior does not depend on input and is found on the first arbitrary execution. On the other hand, not all of our unsafe programs were fast to falsify: in the “Stateful check” program the bug is found on one specific path and the program has many input dependent branches. As such Dash requires multiple iterations to discover the correct path.

The “Locks N” programs consist of an infinite loop, in which a random subset of N locks are first acquired and then the same locks are released in reverse order. As each lock is released, it is checked that the lock was actually previously acquired. Because the branches related to handling each lock are interleaved with each other, the verification time could potentially be multiplied whenever a new lock is introduced. However, looking at the running times from 5 to 15 locks we can see that a combinatorial explosion is avoided.

Table 5.17
A comparison of running times (in seconds) with leading tools from SV-COMP 2015.

Name	LCTD	CPAchecker	ESBMC 1.24.1	Predator
Alias of return 1	0.04	2.12	0.05	0.19
Alias of return 1.1	0.04	2.19	0.06	0.18
Alias of return 2	0.11	2.22	0.06	0.18
Alias of return 2.1	0.04	2.21	0.06	0.19
Byte add safe 1	5.91	34.56	0.08	–
Byte add safe 2	5.86	78.33	0.10	–
Byte add unsafe	0.05	63.58	0.36	–
GCD 1	1.04	2.29	0.59	–
GCD 2	0.52	2.31	18.54	–
GCD 3	1.70	2.26	18.45	–
GCD 4	2.29	1.38	0.06	–
Jain 1	0.42	2.52	0.07	–
Jain 2	0.45	2.61	0.07	–
Jain 4	0.66	2.60	0.06	–
Jain 5	0.42	TIMEOUT	0.04	–
Jain 6	0.54	2.65	0.05	–
Jain 7	0.53	4.97	0.06	–
Modulus	2.02	TIMEOUT	4.70	–
Mutex lock int safe	0.10	2.12	0.06	0.19
Mutex lock int unsafe	0.01	2.44	0.10	0.11
Num conversion 2	0.61	17.07	0.06	–
oomInt	0.15	1.41	0.06	0.18
Parity	1.16	TIMEOUT	0.19	–
Size of parameters	0.04	1.54	1.47	0.18
Stateful check	0.86	2.00	0.34	0.15
Locks 5	3.87	1.76	0.09	–
Locks 6	4.84	2.08	0.11	–
Locks 7	8.63	2.58	0.11	–
Locks 8	6.75	3.37	0.13	–
Locks 9	12.45	5.60	0.15	–
Locks 10	8.83	12.43	0.17	–
Locks 11	17.48	41.40	0.20	–
Locks 12	20.00	122.18	0.22	–
Locks 13	12.27	122.14	0.25	–
Locks 14	13.33	122.05	0.28	–
Locks 15	14.67	122.18	0.32	–

Table 5.18
Results for the synthetic array benchmarks.

Name	Result	Time (s)	Test runs	Solve runs	SMT calls	Unsat targets
Array store safe 10	SAFE	0.52	2	12	21	3
Array store safe 100	SAFE	1.69	2	12	21	3
Array store safe 1000	SAFE	163.53	2	12	21	3
Array store unsafe 10	UNSAFE	0.24	3	3	6	0
Array store unsafe 30	UNSAFE	7.65	3	3	6	0
Array store unsafe 50	UNSAFE	99.37	3	3	6	0

To place these results in context, a comparison of running times for LCTD and leading tools from the 2015 SV-COMP [33] is provided. We compare LCTD against the winning tools of the categories the benchmarks in Table 5.16 were drawn from: ESBMC [34] for the "BitVectors" category, CPAchecker [35] for the "ControlFlow" category and Predator [36] for the "Heap-Manipulation" category. For these tools Table 5.17 includes the running times from the results of the 2015 SV-COMP, which were obtained on an Intel Core i7-4770 @ 3.40 GHz with 33 GB of memory. Note that Predator did not take part in the "BitVectors" and "ControlFlow" categories. As these running times were achieved on different hardware, Table 5.17 should be used only to place our results in broad context. An entry TIMEOUT indicates the running time exceeded 900 seconds. LCTD achieves running times that are competitive with those of CPAchecker on all benchmarks. The running times of ESBMC 1.24.1 are very low for a large majority of the benchmarks, but LCTD achieves significantly smaller running times on some outliers.

The results for the synthetic array benchmarks can be seen in Table 5.18. The number in the program name indicates the length of the array used by the program. In both "Array store safe" and "Array store unsafe" there is a single path to the error in the CFG. Both programs initialize an array with zeros, then store a one at a symbolic index and finally assert that the element at index 7 is zero. In the unsafe version the symbolic index may be any valid index of the array and in the safe version it is any even index of the array. We can see that as the length of the array increases the number of iterations

```
int main() {                                                    1
    int lock = 1, x = 0, y = 0;                                 2
    int *p = &y;                                                3
    if (lct_get_bool())                                         4
        x = x + 1;                                              5
    else                                                        6
        *p = *p + 1;                                            7
    if (lct_get_bool())                                         8
        x = x + 1;                                              9
    else                                                        10
        *p = *p + 1;                                            11
    if (lct_get_bool())                                         12
        x = x + 1;                                              13
    else                                                        14
        *p = *p + 1;                                            15
    if (lct_get_bool())                                         16
        x = x + 1;                                              17
    else                                                        18
        *p = *p + 1;                                            19
    if (lct_get_bool())                                         20
        x = x + 1;                                              21
    else                                                        22
        *p = *p + 1;                                            23
    assert(lock == 1);                                          24
}                                                               25
```

Fig. 5.19. A program with diamond structures.

Table 5.20
Results for verifying the program in Fig. 5.19 with different algorithms.

Algorithm	Time (s)	Iterations	Solver calls	Unsat targets
DSE	6.883	32	31	–
Dash, plain	5.044	132	128	–
Dash, target unsat check	0.144	6	6	2

needed to prove the programs SAFE and UNSAFE, respectively, does not change. However, the constraints generated by the programs with longer arrays are more difficult for an SMT solver to solve, which can be seen in the rising verification times.

We also illustrate the effectiveness of the refinement for unsatisfiable constraints of target regions described in Section 4.3. For this we used the program in Fig. 5.19. The assertion on line 24 is never violated because the variable `lock` is initialized to 1 and never changed. The assertion is preceded by five input dependent if-else constructs, which are irrelevant to the verification task at hand. We compared verifying this program with DSE, plain Dash and a version of Dash with support for the additional refinement method. The results are shown in Table 5.20. We can see that DSE and plain Dash achieve similar running times. However, the version with the check for unsatisfiable target regions is some 35 times faster than plain Dash. Comparing the iterations used we can see that plain Dash has encountered a combinatorial explosion due to the diamond structures in the program, while the modified version has not.

6. Related work

This section discusses work related to the techniques described in this paper. For an extensive survey on automated software verification methods see [37].

6.1. Abstraction refinement tools

The SLAM software model checker [13] is one of the first tools for implementing practical automatic predicate abstraction of C programs. In SLAM a Boolean abstraction of a program is refined in a CEGAR loop to verify that an interface specification is not violated.

In 2005 Godefroid and Klarlund [38] predicted that testing and abstraction based verification methods could be combined in useful ways. Yorsh, Ball and Sagiv [39] propose one such approach, which aims to alleviate the cost of constructing the abstraction by abstracting from a set of concrete test executions. In their method a theorem prover is used to check that the concrete tests cover all the reachable abstract states instead of directly computing the abstract transition relation. This can in the best case reduce the number of required theorem prover calls when compared to previous methods. Kroening, Groce and Clarke [16] describe the CRunner tool, which also combines information from concrete executions with coun-

terexample guided abstraction refinement. Their approach combines concrete executions with partial SAT-based simulation of the program to make detecting spurious counterexamples more efficient.

The Synergy algorithm [17] is a predecessor of Dash. Similarly to Dash, Synergy combines a DSE based testing approach with abstraction refinement. However, Dash presents several improvements over Synergy. The refinement method in Dash for maintaining the abstraction avoids extra solver calls: while Synergy uses a solver call to refine the abstraction, Dash uses information from the failed test generation attempt to do the refinement. Dash also presents a technique for efficiently handling programs with pointers by using concrete pointer aliasing information gathered from tests. Finally, while the Synergy algorithm only supports single-procedure programs, in Dash programs with multiple procedures are supported via a recursive call to the algorithm for each procedure call. Both Synergy and Dash were originally implemented in Microsoft's Yogi tool [40].

The directed abstraction refinement approach of Dash is extended to verifying machine code in the tool McVeto [41]. Machine code presents several challenges that must be addressed, which include the absence of type information, no guarantees on aliasing, extensive use of pointer arithmetic, instruction aliasing and self-modifying code. To handle code with instruction aliasing (same bytes decoded as different instructions due to small offsets in jumps) McVeto constructs the control flow graph during the verification process. Because machine code can read the value of the program counter (PC) McVeto treats it as data. To handle self-modifying code the predicates for the regions in McVeto may also include constraints that require that the memory at the address held by the PC decodes into a particular instruction. Even though the LLVM IR is a fairly low level representation, it does include type information, has sufficient restrictions on aliasing and prohibits self-modifying code, which allow us to avoid the challenges addressed in McVeto.

Ultimate Automizer is a verification tool that represents refinement by generalizing infeasible abstract error traces to automata that describe sets of infeasible traces [42]. A program is shown to be correct when the language of possible traces of the target program can be shown to be included in the union of the languages of the automata describing infeasible traces. Ultimate Automizer differs from traditional predicate abstraction tools in that it models programs as automata on statements appearing in the programs.

Blast [15] software verification tool employs an abstraction refinement approach called *lazy abstraction*. Similarly to Dash, the abstraction in Blast is refined backwards along an infeasible abstract error trace. However, Blast represents the abstraction as a search tree of regions. Refinement is represented by backtracking to a new branch in this search tree with a refinement predicate for the new subtree. Instead of the suitable predicates used by Dash, Blast uses interpolants for refinement. CPAchecker [35] is another tool similar to Blast, which implements the lazy abstraction algorithm in a configurable and extensible framework.

6.2. Verification tools for concurrent software

Verification of concurrent programs presents some challenges that are not present when targeting sequential programs. Most importantly, the execution of a concurrent program is inherently nondeterministic, because the interleaving of operations from different threads is dependent on the environment, i.e., the scheduler. Concurrent programs also exhibit new types of bugs: ones arising from *data races*, where the same shared memory location is modified concurrently to another access, and *deadlocks*, where all threads of the program are simultaneously waiting for a resource held by another thread. These challenges must be handled in a verification tool targeting concurrent software.

Blast [15] has been extended to support concurrent software with the thread-modular abstraction refinement (TAR) algorithm [43]. The TAR algorithm extends the abstraction refinement algorithm in Blast to multithreaded programs. The algorithm is *thread-modular*, meaning that it only explores the state space of one thread at a time. Blast with TAR will fail to prove some concurrent programs correct, as thread-modular reasoning is incomplete [44].

Threader [45] is another verification tool that employs abstraction refinement to verify concurrent programs. Unlike Blast with TAR, Threader does not restrict itself to thread-modular reasoning. Instead it finds a thread-modular proof of correctness when one is available, but will fall back to monolithic reasoning when necessary. This way it exploits the scalability offered by thread-modular reasoning [46] when possible, while still retaining the possibility of verifying programs that do not admit a thread-modular proof.

Concurrent programs often execute the same code in multiple threads. For example a program might spawn a set of worker threads to execute a function in parallel. If during the abstraction refinement process the abstraction for one of these threads is refined, it would be useful to use the refinement also for the other threads. SatAbs [47] is a verification tool that exploits these symmetries in the structure of programs.

6.3. Other automated verification tools

Loopfrog is a verification tool which employs loop summarization to check program correctness [48]. Loopfrog finds loop invariants by checking which formulas from a set of heuristically generated candidates are invariant. Loops are then modeled with a "havoc" instruction, which nondeterministically sets all variables to values which satisfy the loop invariant. Summaries created for loops and other fragments of are combined into a loop-free abstraction, on which it is then checked that no assertion errors are reachable using the C Bounded Model Checker (CBMC) [49] as the underlying verification engine. Unlike abstraction refinement tools, Loopfrog performs no iterative refinement and instead tries to arrive at

sufficiently precise abstraction in a single step. Similarly to a single iteration in abstraction refinement, if LOOPFROG reports any assertion errors that are reachable in the abstraction as counterexamples. It is not, however, checked whether or not the counterexample is spurious.

IC3 is a verification technique that has been shown to be effective for hardware verification [50]. IC3 has been adapted to software verification by extending it to use SMT as the underlying theory and to exploit the control flow structure of the program [51].

SMACK [52] is a tool that also implements verification of programs in the LLVM IR by translating them to Boogie IVL [53], an intermediate language targeted at verification. One complication with the translation is that while LLVM IR supports dynamic allocation of memory, Boogie programs must operate on a static set of global and local variables (which can include arrays). SMACK handles this by partitioning statements with static analysis into sets that access disjoints regions of memory and creates an array variable for each of these. In contrast, DASH handles memory by only considering pointer aliasings encountered in concrete executions. See the suitable predicate construction rules described in Section 3 for a description of our strategy for creating the array variables representing memory.

Bounded model checking is a verification approach, where executions of a program up to a bound verified for correctness and when verification for a bound succeeds the bound is increased. This process is iterated until a bug is found or some predefined timeout is reached. CBMC [49] implements bounded model checking for ANSI-C programs. It supports dynamic memory allocation, pointer arithmetic and pointer type-casts (reinterpreting one type of pointer as a pointer of another type). CBMC uses a SAT-solver for satisfiability queries. The Efficient SMT-Based Context-Bounded Model Checker (ESBMC) [34] is another bounded model checking tool, but contrary to CMBC it encodes C programs as SMT formulas. ESBMC provides a set of encodings geared towards handling C structures and unions, pointer aliasing and floating point arithmetic efficiently.

6.4. Dynamic symbolic execution

DSE was originally developed as a testing approach in the tools DART [6] and EXE [9]. The approach was refined in Cute and jCute [7,8] with support for programs, whose data structures form object graphs. Pex is a DSE tool for the .NET framework, which employs a fitness based search strategy for guiding the exploration to achieve greater coverage with fewer iterations. DSE has also been employed in the context of security testing: the testing tool SAGE [1] implements a technique called “whitebox fuzzing” by extending fuzz testing with DSE. While in traditional fuzz testing the input file is randomly generated, SAGE uses DSE to find interesting inputs.

DSE on LLVM was implemented in the tool KLEE [54]. It has been successfully used for testing systems software on Unix: on a test suite for tools from the BUSYBOX coreutils collection KLEE achieved full line coverage on 31 of them, thus verifying them to be free from assertion errors (which KLEE checks for). The tool extended in this work, LCT [19], also implements DSE on LLVM. The most notable difference between the approaches used in KLEE and LCT is the method used for backtracking: KLEE creates a clone of the symbolic state when a branch occurs, while LCT re-executes the program.

6.5. Model-based testing tools

Model-based testing (MBT) is a testing approach where tests are generated from an abstract model of the system. In this setting symbolic execution can be applied to exploring the abstract model. One such approach was implemented in the tool Qtronic [55], which models target programs in a variant of the Scheme programming language. Tests are generated from the model using symbolic execution.

The model-based testing tool Spec Explorer has been extended with support for combinatorial interaction testing (CIT) [56]. The tool uses symbolic execution to explore a model and generate sets of parameter combinations that exercise the model.

For a recent survey on symbolic methods in testing see [57].

7. Conclusions

Combining automated test generation with counterexample guided abstraction refinement is a promising approach to software verification. In this work we have presented LCTD, which implements the DASH algorithm [18] for C programs on LLVM. As all of our instrumentation operates on the LLVM internal representation, with minor modifications LCTD can be used for any programming language that compiles down to our supported set of instructions. Our case study, while limited, shows the viability of our implementation. LCTD is open source and is available for download at: http://users.cse.aalto.fi/osaariki/lctd-jlap-2014/

As our tool targets the LLVM internal representation, we have in Section 3 presented a strategy for constructing splitting predicates for LLVM instructions in the presence of pointers and array indexing with the `getelementptr` instruction. One limitation in LCTD is that currently storing pointers to memory is not handled. We have left support for this for further work. We also have not implemented the interprocedural support for DASH described in [18]. While the feature itself is not conceptually very complex, we estimated that handling all of its implementation details would have been a significant undertaking.

Section 4.1 describes how our tool evaluates splitting predicates for concrete states using only concrete values of pointers and the execution path taken. In contrast, the YOGI tool from Microsoft stores the complete concrete state along the execution path. Due to the storage requirements of storing concrete states YOGI stores sets of values, called *delta states*, that contain only the values that change between states [29]. LCTD also stores delta states, but because only concrete values of pointers are stored our tool potentially uses less storage. One drawback with our approach is that the region of the final state of an execution must be known. Currently this is ensured in LCTD as the test executions are not bounded and therefore cannot terminate in a region that has been split. Also the pointer aliasing must not depend on input to the program along a fixed path, i.e., by assigning a pointer through a ternary `select` instruction. Note that symbolic indices into arrays are supported, because two pointers are considered equal if they have the same base pointer.

In the future we plan to extend our tool to support a wider range of programs. This includes adding support for C structures and implementing an interprocedural version of DASH.

On C programs we have not noticed cases where the translation to LLVM IR would lose information that would be useful for verification. However, it would be interesting to investigate if this approach works for higher level languages than C.

Once LCTD has sufficient support to allow verification of a larger variety of real-world programs, exploring optimizations for the implementation and algorithm will become of interest. Some potential optimizations have already been evaluated in the YOGI tool. We are also interested in extending our tool to support multiple concurrent clients, which would allow test execution and constraint solving to be distributed to multiple machines. For dynamic symbolic execution LCT scales well to at least 20 clients [22], but whether we can achieve similar results for DASH remains to be seen.

An issue we encountered in DASH during implementation was that the predicates in regions could grow very large from continual splitting along the same abstract error trace. This problem disappeared once we implemented the more powerful refinement operation described in Section 4.3. However, as we will attempt to verify larger programs we may encounter this problem again. One approach to alleviate this problem may be to employ interpolation, which can produce smaller predicates than weakest preconditions [29,58].

Another direction for future work is extending our tool to support verification of multithreaded programs. We have previously extended LCT to support testing multithreaded programs by combining dynamic symbolic execution with the dynamic partial order reduction algorithm [59]. Our research group also has made advancements in testing multithreaded software by applying Petri net unfoldings to achieve better partial order reductions [60]. It would be interesting to see whether some of these techniques could be combined with the counterexample guided abstraction refinement approach of DASH.

Acknowledgements

We would like to thankfully acknowledge the funding from the SARANA project in the SAFIR 2014 program and the Academy of Finland projects 139402 and 277522.

References

[1] P. Godefroid, M.Y. Levin, D.A. Molnar, SAGE: whitebox fuzzing for security testing, ACM Queue 10 (1) (2012) 20–27.
[2] J. Rushby, New challenges in certification for aircraft software, in: Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT 2011), ACM, 2011, pp. 211–218.
[3] N.G. Leveson, The role of software in spacecraft accidents, AIAA J. Spacecraft Rockets 41 (2004) 564–575.
[4] N.G. Leveson, Learning from the past to face the risks of today, Commun. ACM 56 (6) (2013) 38–42.
[5] A. Valmari, The state explosion problem, in: W. Reisig, G. Rozenberg (Eds.), Lectures on Petri Nets I: Basic Models, in: Lecture Notes in Computer Science, vol. 1491, Springer, 1996, pp. 429–528.
[6] P. Godefroid, N. Klarlund, K. Sen, DART: directed automated random testing, in: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI 2005), ACM, 2005, pp. 213–223.
[7] K. Sen, G. Agha, CUTE and jCUTE: concolic unit testing and explicit path model-checking tools, in: T. Ball, R.B. Jones (Eds.), Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006), in: Lecture Notes in Computer Science, vol. 4144, Springer, 2006, pp. 419–423.
[8] K. Sen, Scalable automated methods for dynamic program analysis, PhD thesis, University of Illinois, 2006.
[9] C. Cadar, V. Ganesh, P.M. Pawlowski, D.L. Dill, D.R. Engler, EXE: automatically generating inputs of death, ACM Trans. Inf. Syst. Secur. 12 (2) (2008) 322–335.
[10] P. Godefroid, M.Y. Levin, D.A. Molnar, Automated whitebox fuzz testing, in: Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS 2008), The Internet Society, 2008, pp. 151–166.
[11] S. Graf, H. Saïdi, Construction of abstract state graphs with PVS, in: O. Grumberg (Ed.), Proceedings of the 9th International Conference on Computer Aided Verification (CAV 1997), in: Lecture Notes in Computer Science, vol. 1254, Springer, 1997, pp. 72–83.
[12] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-guided abstraction refinement, in: E.A. Emerson, A.P. Sistla (Eds.), Proceedings of the 12th International Conference on Computer Aided Verification (CAV 2000), in: Lecture Notes in Computer Science, vol. 1855, Springer, 2000, pp. 154–169.
[13] T. Ball, R. Majumdar, T. Millstein, S.K. Rajamani, Automatic predicate abstraction of C programs, in: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI 2001), ACM, 2001, pp. 203–213.
[14] S. Chaki, E.M. Clarke, A. Groce, S. Jha, H. Veith, Modular verification of software components in C, IEEE Trans. Softw. Eng. 30 (6) (2004) 388–402.
[15] T.A. Henzinger, R. Jhala, R. Majumdar, G. Sutre, Lazy abstraction, in: Proceedings of the 29th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL 2002), ACM, 2002, pp. 58–70.
[16] D. Kroening, A. Groce, E.M. Clarke, Counterexample guided abstraction refinement via program execution, in: J. Davies, W. Schulte, M. Barnett (Eds.), Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM 2004), in: Lecture Notes in Computer Science, vol. 3308, Springer, 2004, pp. 224–238.

[17] B.S. Gulavani, T.A. Henzinger, Y. Kannan, A.V. Nori, S.K. Rajamani, SYNERGY: a new algorithm for property checking, in: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2006), ACM, 2006, pp. 117–127.
[18] N.E. Beckman, A.V. Nori, S.K. Rajamani, R.J. Simmons, S.D. Tetali, A.V. Thakur, Proofs from tests, IEEE Trans. Softw. Eng. 36 (4) (2010) 495–508.
[19] K. Kähkönen, Automated dynamic test generation for sequential Java programs, Master's thesis, Helsinki University of Technology, Department of Information and Computer Science, 2008.
[20] K. Kähkönen, T. Launiainen, O. Saarikivi, J. Kauttio, K. Heljanko, I. Niemelä, LCT: an open source concolic testing tool for Java programs, in: Proceedings of the 6th Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE 2011), 2011, pp. 75–80.
[21] K. Kähkönen, Automated test generation for software components, Technical report TKK-ICS-R26, Helsinki University of Technology, Department of Information and Computer Science, 2009.
[22] K. Kähkönen, O. Saarikivi, K. Heljanko, LCT: a parallel distributed testing tool for multithreaded Java programs, Electron. Notes Theor. Comput. Sci. 296 (2013) 253–259.
[23] C. Lattner, V.S. Adve, LLVM: a compilation framework for lifelong program analysis & transformation, in: Proceedings of the 2nd IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2004), IEEE Computer Society, 2004, pp. 75–88.
[24] C. Lattner, V. Adve, LLVM assembly language reference manual, http://llvm.org/docs/LangRef.html, 2014.
[25] T. Ball, S.K. Rajamani, Automatically validating temporal safety properties of interfaces, in: M.B. Dwyer (Ed.), Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN 2001), in: Lecture Notes in Computer Science, vol. 2057, Springer, 2001, pp. 103–122.
[26] E.W. Dijkstra, Guarded commands, nondeterminacy and formal derivation of programs, Commun. ACM 18 (8) (1975) 453–457.
[27] A. Stump, C.W. Barrett, D.L. Dill, J. Levitt, A decision procedure for an extensional theory of arrays, in: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS 2001), IEEE Computer Society, 2001, pp. 29–37.
[28] O. Saarikivi, Test-guided proofs for C programs on LLVM, Master's thesis Aalto University School of Science, 2013.
[29] A.V. Nori, S.K. Rajamani, An empirical study of optimizations in YOGI, in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering – vol. 1 (ICSE 2010), ACM, 2010, pp. 355–364.
[30] J. Zhao, S. Nagarakatte, M.M.K. Martin, S. Zdancewic, Formalizing the LLVM intermediate representation for verified program transformations, in: Proceedings of the 39th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL 2012), ACM, 2012, pp. 427–440.
[31] R. Brummayer, A. Biere, Boolector: an efficient SMT solver for bit-vectors and arrays, in: S. Kowalewski, A. Philippou (Eds.), Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009), in: Lecture Notes in Computer Science, vol. 5505, Springer, 2009, pp. 174–177.
[32] L. De Moura, N. Bjørner, Z3: an efficient SMT solver, in: C.R. Ramakrishnan, J. Rehof (Eds.), Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008), in: Lecture Notes in Computer Science, vol. 4963, Springer, 2008, pp. 337–340.
[33] D. Beyer, Software verification and verifiable witnesses (report on SV-COMP 2015), in: C. Baier, C. Tinelli (Eds.), Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and of Analysis Systems (TACAS 2015), in: Lecture Notes in Computer Science, vol. 9035, Springer, 2015, pp. 401–416.
[34] L. Cordeiro, B. Fischer, J. Marques-Silva, SMT-based bounded model checking for embedded ANSI-C software, in: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009), IEEE Computer Society, 2009, pp. 137–148.
[35] D. Beyer, M.E. Keremoglu, CPACHECKER: a tool for configurable software verification, in: G. Gopalakrishnan, S. Qadeer (Eds.), Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011), in: Lecture Notes in Computer Science, vol. 6806, Springer, 2011, pp. 184–190.
[36] K. Dudka, P. Peringer, T. Vojnar, Predator: a practical tool for checking manipulation of dynamic data structures using separation logic, in: G. Gopalakrishnan, S. Qadeer (Eds.), Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011), in: Lecture Notes in Computer Science, vol. 6806, Springer, 2011, pp. 372–378.
[37] R. Jhala, R. Majumdar, Software model checking, ACM Comput. Surv. 41 (4) (2009) 21:1–21:54.
[38] P. Godefroid, N. Klarlund, Software model checking: searching for computations in the abstract or the concrete, in: J. Romijn, G. Smith, J. van de Pol (Eds.), Proceedings of the 5th International Conference on Integrated Formal Methods (IFM 2005), in: Lecture Notes in Computer Science, vol. 3771, Springer, 2005, pp. 20–32.
[39] G. Yorsh, T. Ball, M. Sagiv, Testing, abstraction, theorem proving: better together!, in: Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2006), ACM, 2006, pp. 145–156.
[40] A.V. Nori, S.K. Rajamani, S. Tetali, A.V. Thakur, The YOGI project: software property checking via static analysis and testing, in: S. Kowalewski, A. Philippou (Eds.), Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009), in: Lecture Notes in Computer Science, vol. 5505, Springer, 2009, pp. 178–181.
[41] T. Reps, J. Lim, A. Thakur, G. Balakrishnan, A. Lal, There's plenty of room at the bottom: analyzing and verifying machine code, in: T. Touili, B. Cook, P. Jackson (Eds.), Proceedings of the 22nd International Conference on Computer Aided Verification (CAV 2010), in: Lecture Notes in Computer Science, vol. 6174, Springer, 2010, pp. 41–56.
[42] M. Heizmann, J. Hoenicke, A. Podelski, Software model checking for people who love automata, in: N. Sharygina, H. Veith (Eds.), Proceedings of the 25th International Conference on Computer Aided Verification (CAV 2013), in: Lecture Notes in Computer Science, vol. 8044, Springer, 2013, pp. 36–52.
[43] T.A. Henzinger, R. Jhala, R. Majumdar, S. Qadeer, Thread-modular abstraction refinement, in: W.A.J. Hunt, F. Somenzi (Eds.), Proceedings of the 15th International Conference on Computer Aided Verification (CAV 2003), in: Lecture Notes in Computer Science, vol. 2725, Springer, 2003, pp. 262–274.
[44] C. Flanagan, S. Qadeer, Thread-modular model checking, in: T. Ball, S.K. Rajamani (Eds.), Proceedings of the 10th International Conference on Model Checking Software (SPIN 2003), in: Lecture Notes in Computer Science, vol. 2648, Springer, 2003, pp. 213–224.
[45] A. Gupta, C. Popeea, A. Rybalchenko, Threader: a constraint-based verifier for multi-threaded programs, in: G. Gopalakrishnan, S. Qadeer (Eds.), Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011), in: Lecture Notes in Computer Science, vol. 6806, Springer, 2011, pp. 412–417.
[46] C. Flanagan, S. Qadeer, S. Seshia, A modular checker for multithreaded programs, in: E. Brinksma, K.G. Larsen (Eds.), Proceedings of the 14th International Conference on Computer Aided Verification (CAV 2002), in: Lecture Notes in Computer Science, vol. 2404, Springer, 2002, pp. 180–194.
[47] A. Donaldson, A. Kaiser, D. Kroening, T. Wahl, Symmetry-aware predicate abstraction for shared-variable concurrent programs, in: G. Gopalakrishnan, S. Qadeer (Eds.), Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011), in: Lecture Notes in Computer Science, vol. 6806, Springer, 2011, pp. 356–371.
[48] D. Kroening, N. Sharygina, S. Tonetta, A. Tsitovich, C.M. Wintersteiger, Loop summarization using state and transition invariants, Form. Methods Syst. Des. 42 (3) (2013) 221–261.
[49] E. Clarke, D. Kroening, F. Lerda, A tool for checking ANSI-C programs, in: K. Jensen, A. Podelski (Eds.), Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004), in: Lecture Notes in Computer Science, vol. 2988, Springer, 2004, pp. 168–176.
[50] A.R. Bradley, SAT-based model checking without unrolling, in: R. Jhala, D. Schmidt (Eds.), Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2011), in: Lecture Notes in Computer Science, vol. 6538, Springer, 2011, pp. 70–87.
[51] A. Cimatti, A. Griggio, Software model checking via IC3, in: P. Madhusudan, S.A. Seshia (Eds.), Proceedings of the 24th International Conference on Computer Aided Verification (CAV 2012), in: Lecture Notes in Computer Science, vol. 7358, Springer, 2012, pp. 277–293.

[52] Z. Rakamarić, M. Emmi, SMACK: decoupling source language details from verifier implementations, in: A. Biere, R. Bloem (Eds.), Proceedings of the 26th International Conference on Computer Aided Verification (CAV 2014), in: Lecture Notes in Computer Science, vol. 8559, Springer, 2014, pp. 106–113.
[53] R. DeLine, K.R.M. Leino, BoogiePL: a typed procedural language for checking object-oriented programs, Tech. rep. MSR-TR-2005-70, Microsoft Research, 2005.
[54] C. Cadar, D. Dunbar, D. Engler, KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs, in: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI 2008), USENIX Association, 2008, pp. 209–224.
[55] A. Huima, Implementing conformiq qtronic, in: A. Petrenko, M. Veanes, J. Tretmans, W. Grieskamp (Eds.), Proceedings of the 19th IFIP TC6/WG6.1 International Conference, and 7th International Conference on Testing of Software and Communicating Systems (TESTCOM/FATES 2007), in: Lecture Notes in Computer Science, vol. 4581, Springer, 2007, pp. 1–12.
[56] W. Grieskamp, X. Qu, X. Wei, N. Kicillof, M.B. Cohen, Interaction coverage meets path coverage by SMT constraint solving, in: M. Núñez, P. Baker, M.G. Merayo (Eds.), Proceedings of the 21st IFIP WG 6.1 International Conference on Testing of Software and Communication Systems and 9th International FATES Workshop (TESTCOM/FATES 2009), in: Lecture Notes in Computer Science, vol. 5826, Springer, 2009, pp. 97–112.
[57] T. Jéron, M. Veanes, B. Wolff, Symbolic methods in testing (Dagstuhl seminar 13021), Dagstuhl Rep. 3 (1) (2013) 1–29.
[58] T.A. Henzinger, R. Jhala, R. Majumdar, K.L. McMillan, Abstractions from proofs, in: Proceedings of the 31st ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL 2004), ACM, 2004, pp. 232–244.
[59] O. Saarikivi, K. Kähkönen, K. Heljanko, Improving dynamic partial order reductions for concolic testing, in: Proceedings of the 12th International Conference on Application of Concurrency to System Design (ACSD 2012), IEEE, 2012, pp. 132–141.
[60] K. Kähkönen, O. Saarikivi, K. Heljanko, Using unfoldings in automated testing of multithreaded programs, in: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012), ACM, 2012, pp. 150–159.