Diagnosing Abstraction Failure for Separation Logic–based Analyses

Josh Berdine[1], Arlen Cox[2,⋆], Samin Ishtiaq[1], and Christoph M. Wintersteiger[1]

[1] Microsoft Research, Cambridge
[2] University of Colorado, Boulder

Abstract. Abstraction refinement is an effective verification technique for automatically proving safety properties of software. Application of this technique in shape analyses has proved impractical as core components of existing refinement techniques such as backward analysis, general conjunction, and identification of unreachable but doomed states are computationally infeasible in such domains.
We propose a new method to diagnose proof failures to be used in a refinement scheme for Separation Logic–based shape analyses. To check feasibility of abstract error traces, we perform Bounded Model Checking over the traces using a novel encoding into SMT. A subsequent diagnosis finds discontinuities on infeasible traces, and identifies doomed states admitted by the abstraction. To construct doomed states, we give a model-finding algorithm for "symbolic heap" Separation Logic formulas, employing the execution machinery of the feasibility checker to search for concrete counter-examples. The diagnosis has been implemented in SLAyer, and we present a simple scheme for refining the abstraction of hierarchical data structures, and illustrate its effectiveness on benchmarks from the SLAyer test suite.

1 Introduction

Abstraction refinement has proven to be an effective technique for verification of safety properties of software. Iterative refinement of the abstraction allows the use of a coarse and computationally cheap abstraction that often suffices to prove the desired property. If the abstraction is not precise enough, it supports incremental shifting to a potentially very precise and computationally expensive analysis. This technique has been very successfully applied to predicate abstraction domains. Not so for shape analyses. The consequence is that the abstractions used in shape analyses must be very conservative, since any information that is abstracted away is forever irrecoverable. One solution is to simply choose the right abstraction in the first place, but while this can be computationally efficient, the choice is sensitive to the property and program, making this approach difficult to use in tools intended to be somewhat generally applicable.

To explain why a straightforward analogue of traditional counter-example guided abstract refinement (CEGAR) techniques used for predicate abstraction

⋆ This work was performed while an intern at Microsoft Research, Cambridge.

does not work for shape analyses, recall a basic CEGAR procedure. Suppose that the goal is to prove that some error state is not reachable, and that for a given abstraction this proof fails. In this context it is common to abstract traces rather than just states, and failing to find a proof amounts to finding an abstract trace to error, t. The first question is whether t constitutes a disproof of the property, or witnesses that the abstraction is too coarse to prove the property. This question can be answered by checking feasibility of t, that is, whether or not it represents at least one concrete trace. If so, there is nothing to do; the concrete trace witnesses that the program violates the property. If t is not feasible, then it must contain a *discontinuity* where concrete execution cannot follow the abstract trace. That is, for every concrete trace along t from an initial state to a state s at the discontinuity, execution would have to "leap sideways" to some unreachable state s' that the abstraction does not distinguish from s, before concrete execution from s' may proceed to reach error. The aim of abstraction refinement is to increase the precision of the abstraction in order to partition the *doomed* states such as s' from the others, and thereby avoid the introduction of t. To perform an effective refinement, a discontinuity and a characterization of doomed states is found, that is, the failure of abstraction is *diagnosed*. One way to do so is to search for a program point ℓ on t such that the over-approximation Q of the reachable states after executing along t to ℓ and the weakest precondition with respect to error of the command C along the suffix of t from ℓ to *error*, $wp(C, error)$, are consistent, i.e., $Q \wedge wp(C, error) \neq false$.

In this case ℓ is a discontinuity and the models of the formula $Q \wedge wp(C, error)$ are doomed states that need to be partitioned from others. There are various refinement techniques, but the use of precondition computation and conjunction or similar operations is ubiquitous.

The use of precondition computation and conjunction presents a serious problem in the context of shape analysis. To get an understanding of why backward shape analysis is very expensive, consider the weakest precondition of a command that swings a pointer stored at `x` from one object to another `p` resulting in a state satisfying Q: $P = wp(*\mathtt{x} = \mathtt{p},\ Q)$. In the states that satisfy P, there are many possible aliasing configurations for `x`, and $*\mathtt{x}$ might point to any object at all, or be any dangling pointer. There are very many such states, and they are not uniform in a way for which known shape analysis domains provide compact representations. Additionally, shape analyses based on separation logic use "symbolic heap" fragments of the logic similar to that introduced by Smallfoot [4], which do not include general conjunction. Reducing a general conjunction to a symbolic heap formula is theoretically possible, but computationally infeasible.

Therefore, an abstraction failure diagnosis that avoids precondition computation and general conjunction is a prerequisite for refinement of shape abstractions in a fashion similar to that applied for predicate abstraction. We propose to refine based on individual doomed states introduced by abstraction, rather than symbolic representations of all such states, and present a diagnosis technique that identifies discontinuities on abstract traces obtained from failed separation logic proofs and fabricates doomed states showing where the abstraction is too coarse.

Our procedure starts with a failed separation logic proof in the form of an abstract transition system and slices out an abstract counter-example. These abstract counter-examples generally contain loops and hence represent infinitely-many abstract error traces. A finite subset of these abstract traces is checked for feasibility using a very precise modeling of memory allocation and a new technique for encoding bounded model checking (BMC) as a single satisfiability modulo theories (SMT) problem, using quantified formulas with uninterpreted functions and bit-vectors.

If a concrete counter-example is not found, then a new algorithm is used to diagnose the failure of abstraction. This proceeds by searching through the points on the abstract counter-example for a discontinuity. For each point on the abstract counter-example, the prefix leading to the point is replaced with code that generates concrete states represented by the abstract state at that point. If this new abstract counter-example is feasible, then the program point under consideration contains a discontinuity, and the generated state is doomed. The diagnosis algorithm reports the input and output of abstraction and the doomed state witnessing that the abstraction was too coarse.

It should be emphasized that the state-of-the-art in refinement of shape abstractions is manual. When a shape analysis fails, the reason must be diagnosed by hand, and the definition of abstraction must be changed by hand. As the size of analyzed programs increases, the time and effort involved in diagnosing abstraction failure becomes a practical bottleneck. Therefore, automatic diagnosis of abstraction failure by itself represents a significant advance. Additionally, as a demonstration and quality check of the diagnosis, we present a simple automatic abstraction refinement scheme which uses the discontinuity and doomed state to select which "patterns" to use for abstracting hierarchical data structures.

2 Separation Logic–based Shape Analysis

Before presenting the material on abstraction failure diagnosis, we must provide some background on shape analysis using separation logic. In particular we introduce programs, abstract states, abstract transition systems, failed proofs, and give some description of pattern-based abstraction.

Programs. Assuming some language of pure *expressions* E, the language of state-transforming *commands* is generated by the following grammar:

$C ::= x = \texttt{malloc}(E) \mid \texttt{free}(x)$	allocate and delete heap memory
$\mid x = \texttt{nondet}() \mid x = E$	kill and move (register)
$\mid *x = y \mid x = *y$	store and load (heap)
$\mid \texttt{assume}(E) \mid \texttt{assert}(E)$	assumptions and assertions
$\mid \texttt{nop} \mid C; C$	sequential composition

A *program* is defined by its control-flow graph $\langle \mathbb{L}, \mathbb{E}, \ell_0, \kappa \rangle$, where the vertices $\mathbb{L}$ are program locations, the program entry point is the root $\ell_0 \in \mathbb{L}$, and the edges $\mathbb{E} \subseteq \mathbb{L} \times \mathbb{L}$ are labeled with commands by $\kappa : \mathbb{E} \to C$.

Abstract States. Separation logic–based shape analyses represent sets of program states using formulas in a "symbolic heaps" fragment [4] of separation logic's assertion language [24]. Our diagnosis algorithms are implemented in SLAyer [5], which uses the following language of *formulas*:

$$\begin{array}{lll} P, Q ::= F & & \text{first-order formulas} \\ \quad | \ \mathsf{emp} \mid l \mapsto r \mid \mathsf{ls}(\Lambda, k, \boldsymbol{p}, \boldsymbol{f}, \boldsymbol{b}, \boldsymbol{n}) & & \text{atomic heap formulas} \\ \quad | \ P * Q \mid P \vee Q \mid \exists \boldsymbol{x}. Q & & \end{array}$$

Apart from emp, which describes the empty part of a heap, atomic heap formulas are of two forms: points-to or list-segment. A points-to $l \mapsto r$ describes a single heap object at location l that contains a value described by record r. A list-segment $\mathsf{ls}(\Lambda, k, \boldsymbol{p}, \boldsymbol{f}, \boldsymbol{b}, \boldsymbol{n})$ describes a possibly-empty, possibly-cyclic, segment of a doubly-linked list, where the heap structure of each item of the list is given by Λ. In particular, $\mathsf{ls}(\Lambda, k, \boldsymbol{p}, \boldsymbol{f}, \boldsymbol{b}, \boldsymbol{n})$ is the least predicate satisfying

$$\begin{array}{l} \mathsf{ls}(\Lambda, k, \boldsymbol{p}, \boldsymbol{f}, \boldsymbol{b}, \boldsymbol{n}) \text{ iff } (k = 0 \wedge \boldsymbol{f} = \boldsymbol{n} \wedge \boldsymbol{p} = \boldsymbol{b}) \\ \qquad\qquad \vee \exists \boldsymbol{x'}, \boldsymbol{y'}. \, k > 0 \wedge \Lambda(\boldsymbol{p}, \boldsymbol{f}, \boldsymbol{x'}, \boldsymbol{y'}) * \mathsf{ls}(\Lambda, k{-}1, \boldsymbol{x'}, \boldsymbol{y'}, \boldsymbol{b}, \boldsymbol{n}) \ . \end{array}$$

See [3] for details on this predicate, but note that $\boldsymbol{p}, \boldsymbol{f}, \boldsymbol{b}, \boldsymbol{n}$ denote vectors of parameters, which are sometimes empty and written simply as a space.

The set of formulas is closed under separating conjunction $P * Q$, disjunction $P \vee Q$, and existential quantification $\exists \boldsymbol{x}. Q$. Note the absence of conjunction and negation of heap formulas. The pure, heap-independent, part of the logic (F) is essentially passed through to the Z3 SMT solver [17]. We assume that first-order formulas are among the expressions, $F \subset E$.

The set of *abstract states* is $Q^{\top}$, where $\top$ is the error state.

Pattern-based Abstraction. The abstraction performed by SLAyer is parameterized by "patterns", the Λ argument formulas of the ls predicate that describe the shape of hierarchical data structures. See [3] for more detail, but as an example, consider a pattern for simple singly-linked lists

$$\_\text{SLL\_ENTRY}(, \mathit{front}, , \mathit{next}) = (\mathit{front} \mapsto [\mathsf{Flink}\colon \mathit{next}])$$

and a pattern for singly-linked lists where each item carries a data object

$$\_\text{SLL\_OBJS}(, \mathit{front}, , \mathit{next}) = \exists d', r. (\mathit{front} \mapsto [\mathsf{Data}\colon d'; \mathsf{Flink}\colon \mathit{next}]) * (d' \mapsto r) \ .$$

Abstracting the formula, which represents a list of two items carrying data,

$$\begin{array}{l} \exists d'_0, d'_1, f', r_0, r_1. (\mathit{head} = \mathit{item}) \wedge (\mathit{nd} \neq 0) * \\ \qquad\qquad (\mathit{head} \mapsto [\mathsf{Data}\colon d'_0; \mathsf{Flink}\colon f']) * (d'_0 \mapsto r_0) * \\ \qquad\qquad (f' \mapsto [\mathsf{Data}\colon d'_1; \mathsf{Flink}\colon 0]) * (d'_1 \mapsto r_1) \end{array}$$

using _SLL_ENTRY results in (a warning about leaked memory and)

$$\exists l\,.\,(head = item) \wedge (nd \neq 0) * \mathsf{ls}(\text{\_SLL\_ENTRY}, l, , head, , 0)$$

while using _SLL_OBJS results in

$$\exists l\,.\,(head = item) \wedge (nd \neq 0) * \mathsf{ls}(\text{\_SLL\_OBJS}, l, , head, , 0)\ .$$

Note that abstracting using _SLL_OBJS produces a logically stronger result than abstracting using _SLL_ENTRY. The former preserves the fact that the Data fields point to valid objects, while the latter loses this information. As a result, adjusting the patterns used for abstraction provides an effective mechanism for abstraction refinement, analogous to the set of predicates used to control precision of predicate abstraction.

Abstract Transition Systems and Failed Proofs. SLAYER abstracts a program to an *abstract transition system* (ATS). An ATS is a graph $\langle \mathbb{L}, \mathbb{E}, \ell_0, \kappa, \delta \rangle$, which is a program where program points are labeled with abstract states by $\delta : \mathbb{L} \to Q^\top$.

An ATS is constructed by the analysis while exploring the computation tree of the program under the abstract semantics, creating cycles when abstract states are covered by existing ones. A fully-expanded ATS where no vertex is labeled with $\top$ induces a *proof* in separation logic, where for each edge $e = (\ell_i, \ell_j) \in \mathbb{E}$, the triple $\{\delta\,\ell_i\}\ \kappa\,e\ \{\delta\,\ell_j\}$ is valid.

An ATS where some vertex ℓ_e is labeled with $\top$ constitutes a *failed proof*. If $\delta\,\ell_e = \top$, then ℓ_e is an *error* vertex, and the ATS restricted to the transitive predecessors of ℓ_e is an *abstract counter-example*. An abstract counter-example is either concretely feasible, or it witnesses that the abstraction is too coarse.

Abstract Programs. The CEGAR approach to model checking commonly involves construction of an abstract program. If the abstract program contains an error, subsequent analysis finds an abstract trace that shows it. If this trace is infeasible in the abstract program, then it is also infeasible in the concrete program, and refinement may be performed based on the explanation for abstract infeasibility. If it is feasible in the abstract program, then it is checked for feasibility in the concrete program to determine whether it corresponds to a concrete error or should be refined.

We do not use such a two-staged approach. While we do employ abstraction functions to obtain an ATS, they are used to abstract sets of program states, producing *abstract states*, instead of directly abstracting program transitions, producing *abstract transitions*. The ATS is therefore a relation over abstract states, *not* an abstracted relation over concrete states. An abstract program could be obtained from the ATS, however, *all* error traces in the ATS will be feasible in the resulting system.

In short, since we do not use a postcondition computation that loses more precision than required by the abstraction,[3] there is no need to check if a potential counter-example is due to imprecision in the postcondition computation.

[Aside: Some theoretical results regarding the complexity of adding arbitrary Boolean connectives to the fragments of separation logic used in analyzers are known [10]. For the simple propositional case with no inductive definitions, the model checking problem is NP-complete and the validity problem is Π_2^{P}-complete. Adding general Boolean conjunction preserves these bounds. General negation is more problematic, both problems become PSPACE-complete even in this simple case. Furthermore, performing backward analysis in the known way requires $-\!\!*$ [24], which also brings both problems up to PSPACE-complete.*]*

3 Abstraction Failure Diagnosis

Our approach to failure diagnosis is meant to be employed in the context of abstraction refinement. We therefore give a brief overview using a typical abstraction refinement algorithm for presentation purposes. Algorithm 1 first runs an abstract interpreter in `analyze` and if it succeeds, it returns `Safe`. If not, we simplify ATS using `slice` and then search for a concrete counter-example via `feasible`, which is described in Section 4. If it has a concrete counter-example, then we report `Unsafe`. If it does not have a concrete counter-example, we try to refine the abstraction. The `diagnose` procedure searches for doomed states and is described in Section 5. It returns a description of the discontinuity at which the abstract state was identified as doomed. If such a state is not found, or if the refinement fails for other reasons, the algorithm terminates with a result of `PossiblyUnsafe`. Otherwise it repeats the process using the new abstraction.

We illustrate the behavior of our algorithm on the simple linked list program depicted in Figure 1. This program creates a list of non-deterministic length with a heap allocated data object in every element and then deletes the list. This program is safe.

SLAyer initially fails to prove that the program is safe, the corresponding ATS is shown in Figure 2(a). At the transition from vertex 4 to vertex 3, the abstract interpreter explored the first while loop twice, creating and explicitly tracking a list of length 2 with points-to predicates. At the third iteration of the loop, it widens at vertex 2. In doing so, it selects an _SLL_ENTRY shape, thereby discarding information that is required to complete the proof. It still has a d_0 data object, but it has lost d_1 and it has lost any connection between data elements like d_0 and the list itself. When the abstract interpreter reaches the last command through the transition from vertex 1 to 0, it no longer knows if the particular list element points to the beginning of allocated memory or not. As a result, the proof attempt fails.

Once `analyze` terminates with a failed proof attempt, `feasible` attempts to find a concrete counter-example in the abstract counter-example. Since this program is safe, it does not find one, and the algorithm then runs `diagnose` which

[3] With the exception of losing some disequations between deallocated addresses.

Algorithm 1 Abstraction refinement algorithm

```
let abstraction_refinement prog abstraction =
  let ats = analyze prog abstraction in
  if safe ats
    return Safe
  else
    let abstract_cex = slice ats in
    let concrete_cex = feasible abstract_cex bound in
    if concrete_cex != None
      return Unsafe
    else
      let failure = diagnose abstract_cex bound in
      if failure = None
        return PossiblyUnsafe
      else
        let abstraction' = refine failure abstraction in
        if abstraction' = None
          return PossiblyUnsafe
        else
          return abstraction_refinement prog abstraction'
```

searches for a concrete counter-example starting from each widened state in the abstract counter-example. In this example, the state at vertex 2 is the only widened state. It then synthesizes a new, temporary ATS shown in Figure 2(b) which is constructed to generate all models of the separation logic formula on the vertex (within bounds). It then continues to check feasibility of counter-examples in this new ATS, which, in this example, yields a counter-example that constructs a single element list, where the data pointer is invalid.

Now that a doomed state has been found, the `refine` procedure attempts to construct a more precise abstraction. It succeeds only if it is able to find a new abstraction in which the doomed state is no longer included at the discontinuity. In this example, the `refine` procedure implemented in SLAyer (see Section 6) activates the previously inactive _SLL_OBJS pattern which preserves information about the `Data` objects. Finally, it restarts the abstract interpreter with the new abstraction, which, in this example, is successful in proving safety of the program.

4 Feasibility Checking

When the abstract interpretation is unable to show that a program is safe, we obtain an ATS which represents the relevant parts of the program together with an abstract model (abstract values for every variable at every control location). To distinguish between actual errors and abstraction failures, we check feasibility of error traces in the ATS. Note that this is a general verification problem and that we may employ any of a multitude of Model Checking algorithms to solve

```
 1 typedef struct _SLL_ENTRY {
 2  void* Data;
 3  struct _SLL_ENTRY *Flink;
 4 } SLL_ENTRY, *PSLL_ENTRY;
 5
 6 void main(void) {
 7  SLL_OBJS *head = NULL, *item;
 8  while (nondet()) {
 9   item = (PSLL_ENTRY)malloc(sizeof(SLL_ENTRY));
10   item->Data = (int*)malloc(sizeof(int));
11   item->Flink = head;
12   head = item;
13  }
14  while (head) {
15   item = head;
16   head = item->Flink;
17   free(item->Data);
18   free(item);
19  }
20 }
```

Fig. 1. An example program

this problem. Here, we propose a Bounded Model Checker (BMC). For any fixed unrolling depth, this represents an under-approximation of the ATS. The trade-off between precision and efficiency is of paramount importance in practice and we propose to use BMC because it conveniently offers fine-grained control over the precision through a single parameter.

Recent advances in SMT solving have made it possible to encode BMC instances through a single query to the theorem prover [25] and to solve them by providing efficient quantifier instantiation and elimination procedures. In particular, the theory of bit-vectors with uninterpreted functions and quantifiers (SMT UFBV) has been shown to be a very effective means of analyzing BMC instances [33]. This theory allows for an encoding that does not require a predetermined unrolling depth for every loop, but for the whole system, i.e., the unrolling bound corresponds to the number of nodes visited in the ATS, but the SMT solver may freely chose a different bound for each loop in the ATS. This simplifies the analysis and allows the utilization of powerful heuristics employed by SMT solvers to increase performance.

4.1 A Memory Model

To encode an ATS into SMT UFBV, a memory model is required. To achieve maximum precision, we use a flat memory model that implements accurate execution semantics. A segmented model might be easier to analyze, but would introduce unsoundness [18].

This choice is motivated by the particular interest in detecting four specific classes of errors: 1) Array out of bounds errors; 2) Dereferencing NULL pointers; 3) Double frees; and 4) Frees of unallocated memory. In a flat memory model, these four errors can be reduced to two: out of bounds errors and NULL pointer errors can both be treated as dereferencing unallocated memory; a double free error corresponds to an attempt to free unallocated memory.

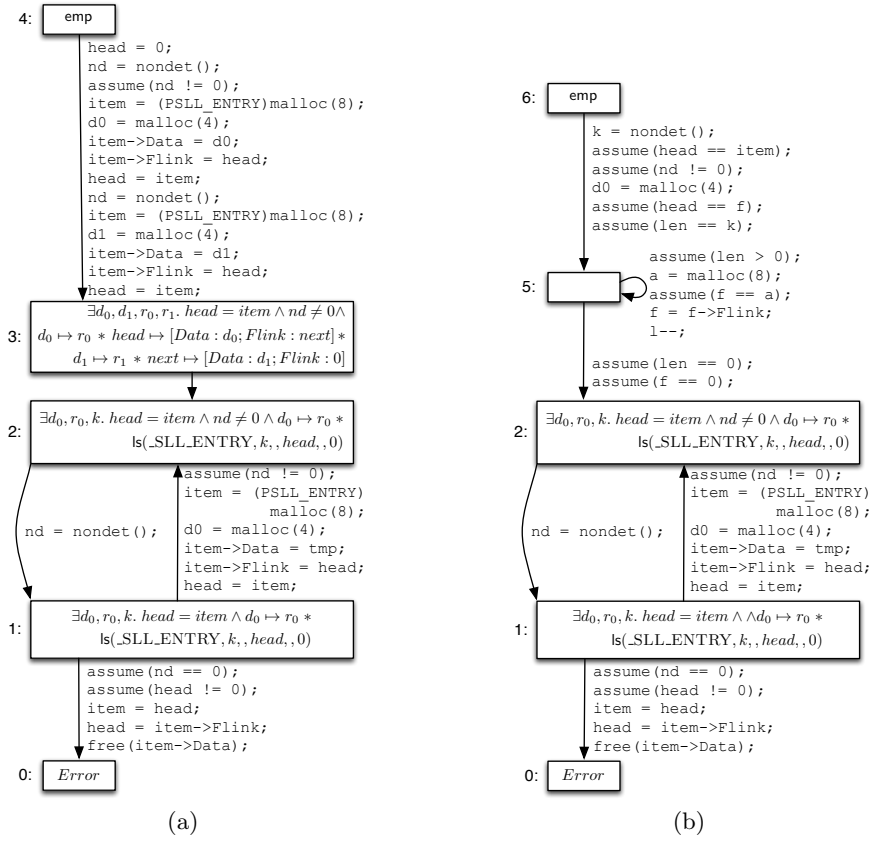

Fig. 2. (a) Abstract counter-example prior to refinement and (b) with prefix of vertex 2 replaced

Memory allocation must be modeled accurately for a flat model to be able to find errors. If a strategy is chosen similar to a real memory allocator (first fit, best fit, etc), the objects are packed together and will likely not cause errors when accessing out of bounds array elements. For this reason we allow the SMT solver to place the allocated objects. We existentially quantify the starting location for each allocation, such that, if objects can be rearranged to cause an error to occur, they will be.

In our encoding, memory is modeled by three arrays: *heap*, *alloc*, and $objsize$. The first contains a representation of the heap at a given time (execution step):

$$heap : \text{Time} \rightarrow \text{Address} \rightarrow Value$$

The *alloc* array is used to track whether a memory address is allocated or not:

$$alloc : \text{Time} \rightarrow \text{Address} \rightarrow Bool$$

If some address is not allocated at the time of being accessed, this corresponds to a segmentation fault. The *objsize* array is used to track the size of allocated objects at a given time and memory address:

$$objsize : \text{Time} \to \text{Address} \to Nat \cup \{\bot\}$$

Note that this array contains $\bot$ (encoded as -1) for memory locations that are allocated, but not at the beginning of an object.

4.2 Encoding to SMT

The procedure `feasible` *ats bound* checks for the feasibility of counter-examples of bounded length in an ATS. If such a counter-example exists, it returns a mapping $\mu : \mathbb{L} \to$ Structures which associates each vertex with a Kripke structure that provides a concrete interpretation for each variable, function symbol, and the heap memory. If no such counter-example exists, `feasible` returns `None`.

In order to pose the bounded model checking problem as a single SMT problem, we make use of quantifiers. We constrain the solver to start at some symbolic set of initial states constructed by *init* and then for some bounded number of steps, unroll the transition relation of the ATS. The function $\text{tr}(ats, t)$ corresponds to the encoding of the transition relation of the ATS *ats* from time $t-1$ to time t. The top-level check is encoded as

$$init() \wedge \forall t.\ 0 < t < bound \to \text{tr}(ats, t)\ .$$

Our encoding makes use of semantic functions $[\![\cdot]\!]$, which take a state and continuations (to work out what to do next in the translation). In what follows, the encoding of commands is denoted by

$$[\![\cdot]\!]^{\text{C}} \quad : St \to (St \to SMT) \to (St \to SMT) \to SMT\ ,$$

which takes a state and two continuations, one for successful transitions and one for transitions to error. The transition relation $tr(ats, t)$ is encoded as

$$tr(ats, t) = (at(t-1) = \ell_e \to at(t) = \ell_e) \wedge \left\{ \bigwedge_{\ell \in \mathbb{L}} at(t-1) = \ell \to \Big(\bigvee_{(\ell,\ell') \in \mathbb{E}} [\![\kappa(\ell, \ell')]\!]^{\text{C}}\ \sigma\ sk\ ek \Big) \right\}\ ,$$

where $\mathbb{L}$ and $\mathbb{E}$ are the sets of vertices and edges of the ATS and the function $at(t)$ encodes the control vertex at time t.

We use the 4-tuple $\langle vars, heap, alloc, objsize \rangle$ to represent a state of the system, where *vars* is the set of variables in the ATS. This is used as the source of generating the corresponding time-stamped variables in the encoding. For efficiency reasons, the implementation also keeps flags for if and when the state was last updated. The arrays in the initial environment σ_0 are empty.

The top-level command encoding takes the two continuations, one to signify a successful transition $sk = \lambda\sigma.\ STEP(t, \ell, \sigma)$ and another for transitions to

the error $ek = \lambda\sigma.\ ERR(t, \sigma)$. If the command completes without error, the command threads the modified state to sk, otherwise it threads the modified state to the error continuation. Once the error continuation is followed, the top-level encoding tr ensures that the system will stay in the error state. These continuations allow for a clean representation of the ATS that maximizes the use of if-then-else structures and minimizes general disjunctions. Threading the state also allowed us to reduce the number of quantifiers used in the problem by using if-then-else constructs instead of quantified uninterpreted functions, so long as we are in the same block as previous heap updates.

The initial continuations end with the $STEP$ and ERR predicates which are defined as follows. $STEP(t, \ell, \sigma)$ asserts that $at(t) = \ell$ to ensure the transition of the vertex to the next time step. (A transition to ℓ_e is explicitly disallowed.) Furthermore, it preserves all the values from the current block that must be preserved (*heap* if modified, *alloc* and $objsize$ if modified, as well as all variables). Lastly, $STEP(t, \ell, \sigma)$ asserts that $pure(\delta\ \ell)$, the pure consequences of the Separation Logic assertion at ℓ, hold at time t. The $ERR(t, \sigma)$ predicate is like $STEP(t, \ell, \sigma)$ except that the transition must be to ℓ_e and $pure(\delta\ \ell) = true$.

We now describe the encoding of commands, concentrating on the memory-related commands `malloc`, `free` and `store`. A forthcoming tech report [6] gives a full definition of the encoding for the other commands.

The `malloc` command produces a new function for the *alloc* array. It uses a fresh variable to store the location. We cannot simply constrain the target variable x, because it may already have been assigned a value and thus is not unconstrained. By introducing a fresh variable, f, constraining it and updating x to be equal to f, we achieve the desired effect. The seemingly odd constraint that $f \leq f + s$, given that $s \geq 0$ exists because of the modular behavior of arithmetic in the bit-vector theory. Without this constraint, memory would be allowed to wrap around past zero. While this behavior should be prohibited by the constraint from *init* that location 0 is always deallocated, adding this constraint provides performance benefits. Formally, the encoding of `malloc` is defined by

$$
\begin{aligned}
[\![\texttt{x := malloc(s)}]\!]^{\mathrm{C}}\ \sigma\ sk\ ek = {} & \textbf{let}\ \langle vars, heap, alloc, objsize\rangle\ =\ \sigma\ \textbf{in} \\
& \textbf{let}\ f\ =\ gensym()\ \textbf{in} \\
& \textbf{let}\ z\ =\ ([\![s]\!]^{\mathrm{Exp}}\ \sigma)\ \textbf{in} \\
& \forall i. f \leq i < f + z \rightarrow alloc(i) = false \wedge \\
& \forall i. f \leq i < f + z \rightarrow objsize(i) = -1 \wedge \\
& \textbf{let}\ a'\ =\ \lambda a.\ ite(f \leq a < f{+}z\ ,\ true\ ,\ alloc\ a)\ \textbf{in} \\
& \textbf{let}\ s'\ =\ \lambda a.\ ite(f = a\ ,\ s\ ,\ objsize\ a)\ \textbf{in} \\
& (sk\ \langle vars \oplus [x \mapsto f], heap, a', s'\ \rangle)\ ,
\end{aligned}
$$

where $gensym()$ represents the introduction of a fresh symbol.

The `free` command is similar to `malloc`, except that it relies upon the values in the $objsize$ array instead of the *alloc* array. It requires that the $objsize$ of the freed address have a value other than -1, whose value indicates no value in the

size array. This value in $objsize$ indicates how many successive entries in $alloc$, starting at address x, need to be set back to $false$. Formally,

$$\begin{aligned}[\![\texttt{free(x)}]\!]^{\mathrm{C}}\ \sigma\ sk\ ek = {} & \textbf{let}\ \langle vars, heap, alloc, objsize\rangle\ =\ \sigma\ \textbf{in}\\ & \textbf{let}\ f\ =\ gensym()\ \textbf{in}\\ & \textbf{let}\ s\ =\ (objsize\ x)\ \textbf{in}\\ & \textbf{let}\ a'\ =\ \lambda a.\ ite(f \le a < f + s\,,\ false\,,\ alloc\ a)\ \textbf{in}\\ & \textbf{let}\ s'\ =\ \lambda a.\ ite(f = a\,,\ -1\,,\ objsize\ a)\ \textbf{in}\\ & \textbf{let}\ \sigma'\ =\ \langle vars, heap, a', s'\ \rangle\ \textbf{in}\\ & ite(x = 0\,,\ (sk\ \sigma)\,,\ ite(s \neq -1\,,\ (sk\ \sigma')\,,\ (ek\ \sigma)))\ .\end{aligned}$$

The `store` command first checks the precondition $(alloc\ x)$, which is that the memory at the target address is in fact allocated. If this precondition holds, the execution is allowed to continue with the updated state where $heap$ has been assigned to a new function. Conversely, the execution continues at ℓ_e, assuming that the state was not updated as required by the command. Formally, we have

$$\begin{aligned}[\![\texttt{*x = y}]\!]^{\mathrm{C}}\ \sigma\ sk\ ek = {} & \textbf{let}\ \langle vars, heap, alloc, objsize\rangle\ =\ \sigma\ \textbf{in}\\ & \textbf{let}\ heap'\ =\ \lambda a.\ ite(a = x\,,\ y\,,\ (heap\ a))\ \textbf{in}\\ & ite((alloc\ x)\,,\ (sk\ \langle vars, heap', alloc, objsize\rangle)\,,\ (ek\ \sigma))\ .\end{aligned}$$

5 Doomed State Synthesis

We define the `diagnose` procedure for identifying doomed states, i.e., for states for which abstraction was too aggressive, and so can be passed to a refinement procedure. Our procedure works as follows: It iterates through the edges of the abstract counter-example, to determine at which of them the widening operator has abstracted too coarsely. It does this by analyzing a new, temporary ATS in which the prefix of the cutpoint ℓ' has been replaced with a program fragment that constructs states which satisfy $\delta\,\ell'$, the formula at that cutpoint. We then use the `feasible` procedure to search for a concrete counter-example in this new ATS. If a counter-example is found, then it returns the discontinuity (ℓ, ℓ') together with the doomed state obtained by looking up ℓ' in the concrete trace μ. The `diagnose` procedure is depicted in Algorithm 2.

Executing the code generated by `prefix` Q produces states that satisfy Q. Algorithm 3 defines `prefix`, where the generated pseudo-code is shorthand for standard control-flow graph construction, and the **local** `v` **in** C form is short for C[`v'`/`v`]; `v'`= `nondet`() where `v'` is fresh.

The model generation assumes a model finder for first-order logic, so first-order formulas F are simply assumed. Existential quantification is synthesized using non-deterministic assignment, reverse engineering Floyd's assignment axiom. Disjunction is translated into a non-deterministic branch, that is, disjunction of commands. Nothing need be done to synthesize `emp` since it is a sub-heap

Algorithm 2 Doomed state search

```
let diagnose ⟨𝕃, 𝔼, ℓ0, κ, δ⟩ bound =
  for (ℓ, ℓ′) ∈ 𝔼 do
    if widened (ℓ, ℓ′)
      let ⟨𝕃n, 𝔼n, ℓn, κn⟩ = cfg_of ((prefix δℓ′); goto ℓ′) in
      let mod_ats = ⟨𝕃 ∪ 𝕃n, 𝔼 ∪ 𝔼n, ℓn, κ ∪ κn, δ ∪ (𝕃n × {emp})⟩ in
      match feasible mod_ats bound with
      | None ->
          continue
      | μ ->
          return ((ℓ, ℓ′), μ(ℓ′))
   return None
```

of any heap. Points-to formulas are synthesized by a `malloc()` call, and separating conjunction is mapped to sequential composition. This has the effect of encoding the core partiality in the semantics of $*$ into the freshness guarantee and non-determinism provided by allocation, meaning that correctly generating models relies on an accurate treatment of allocation. Lastly, lists are synthesized by using a loop to realize induction on the list length. As an example, `prefix` $\mathsf{ls}(\_\text{SLL\_ENTRY}, k, , p, , q)$ is realized by, after slight simplification:

```
local l,f,a;
l = k; f = p;
for (; l>0; --l) {
  a = malloc(sizeof(sll));
  assume(f == a);
  f = f->Flink;
}
assume(f = q ∧ l = 0);
```

Lemma 1. *Every reachable state of* `prefix` Q *satisfies* Q*.*

Theorem 1. *The* `abstraction_refinement` *procedure is a sound analysis.*

Proof. The procedure only returns `Safe` when the abstract interpreter in `analyze` did in fact find a proof; this result is correct as long as the `refine` procedure maintains the fact that the abstraction is in fact a valid abstraction. In case the procedure returns `Unsafe`, it has found a concrete counter-example which witnesses the fact that the program is in fact unsafe. In all other cases, the procedure returns `PossiblyUnsafe`, which does not harm the soundness of the analysis. □

Note that this approach for doomed state synthesis has the effect of translating separation logic formulas to code, and then in the feasibility checker, to first-order logic formulas. It would be possible to compose these two translations and translate separation logic formulas to first-order logic directly, but the result would be more difficult to understand, and would impede reuse in the implementation. Note that the separation logic formulas are not precisely expressible in

Algorithm 3 Prefix synthesis

```
let prefix Q =
  match Q with
  | F ->
      assume(F)
  | ∃x. Q ->
      local x in (prefix Q)
  | Q_0 ∨ ... ∨ Q_N ->
      if nondet() then (prefix Q_0)
      else (prefix Q_1 ∨ ... ∨ Q_N)
  | emp ->
      nop
  | Q ∗ R ->
      (prefix Q); (prefix R)
  | l ↦ [r; o_1 : e_1; ...; o_N : e_N] ->
      local a in
      a = malloc(sizeof(typeof(r)));
      assume(a = l);
      ∗l.o_1 = e_1; ...; ∗l.o_N = e_N
  | ls(Λ, k, p, f, b, n) ->
      local l, w, x in
      l = k; w = p; x = f;
      for (; l > 0; l = l − 1) {
        local y, z in
        (prefix Λ(w, x, y, z));
        w = y; x = z
      }
      assume(l = 0 ∧ w = b ∧ x = n)
```

first-order logic due to transitive closure used to interpret the list predicate and second-order quantification implicit in the semantics of the $*$ connective. So a direct translation must under-approximate, and the ways that the transitive closure and second-order quantification interact make this nontrivial. The translation via the model-construction code avoids eagerly constructing formulas of size exponential in the bound, unlike a naive "blasting" approach. Additionally, the translation via code approach potentially allows the solver to unroll loops in the generation of a model of the separation logic formula guided by the path to error, where a direct translation would blindly generate the first order logic formula without any guidance.

6 Experimental Evaluation

There are two motivations in undertaking the work described in this paper. One is to make precise the notion of abstraction failure diagnosis in separation logic shape analyses. The other, a more practical one, is to use this understanding to

improve the quality of results of SLAyer runs. We implemented feasibility checking and diagnosis in SLAyer. This alone has improved SLAyer regression tests, in particular turning around two dozen known unsafe tests from `PossiblyUnsafe` to definitely `Unsafe`.

We also implemented a simple pattern `refine` procedure. SLAyer keeps a set of active and inactive abstraction patterns. When widening admits a doomed state s, this diagnosis is fed into SLAyer's shape discovery module in order to select a pattern to eliminate the doomed state. The basic algorithm for refinement is to enumerate the inactive patterns, for each one widen to s' using the active patterns plus the chosen one, and then check if s entails s'. If not, keep the chosen pattern active; otherwise it keeps looking. This is a simple automatic refinement procedure, and we can imagine more sophisticated schemes. For instance, to deal with more complex programs, we could try with all the inactive patterns and then minimize akin to unsat core minimization in MaxSAT.

Table 1 presents some experimental results. The programs are taken from the SLAyer test suite, and so are biased towards control (rather than data), traversal through linked lists, pointer arithmetic, etc. The table gives the results for SLAyer without and with this simple pattern refinement scheme. The second column indicates that these are all tests where SLAyer previously reported an inconclusive result, in the time indicated in the third column. The fourth column reports the result using the techniques described here, either Unsafe indicating a concrete counter-example of memory safety was found, or Safe indicating that a memory safety proof was found after abstraction refinement, or PossUnsafe indicating a result that remains inconclusive. The fifth column reports the additional time taken either for feasibility checking or for diagnosis and refinement, indicated as the sum of shape analysis and feasibility checking times.

7 Related Work

Counter-Example Guided Abstraction Refinement (CEGAR) inspired this work. SLAyer's implementation attempts to mirror the primary steps of the algorithm without requiring weakest precondition or general conjunction. There have been many implementations of CEGAR, though it is most popularly used with predicate abstraction as in the SLAM tool [1, 2]. Other implementations include one by Clarke et al [13] applied to hardware, the BLAST project [23], MAGIC [11] and SATABS [15]. Obtaining the initial abstraction is not addressed by CEGAR, but there are several techniques, including existential abstraction [14] and predicate abstraction [16, 22].

Our feasibility checking algorithm is an implementation of bounded model checking [7] and is most closely related to the CBMC [12] bounded model checker for C programs. We implement bounded model checking as a single large problem and leave the task of determining unrolling to the SMT solver. This differs from CBMC in that CBMC does explicit unrolling.

Instead of bounding the depth of the search, it is possible to bound the breadth of the search by using a symbolic or concolic testing technique. Tools like EXE [9],

Table 1. SLAyer versus SLAyer + Feasibility Checking and Pattern Refinement

Test	SLAyer		SLAyer + Diagnosis	
			Disproved/Refined	
	Result	Time	Result	Time
T2_n-19	PossUnsafe	0.031	Unsafe	+0.078
T2_n-1b	PossUnsafe	0.016	Unsafe	+0.062
T2_n-34	PossUnsafe	0.031	Unsafe	+0.140
T2_n-38	PossUnsafe	0.078	Unsafe	+0.421
T2_p-38	PossUnsafe	0.515	Unsafe	+12.230
T2_p-50	PossUnsafe	0.062	Unsafe	+0.562
T2_p-62	PossUnsafe	0.078	Unsafe	+0.546
changing_truth_value	PossUnsafe	0.062	Unsafe	+1.373
complicated_safe	PossUnsafe	0.140	PossUnsafe	+89.279
complicated_unsafe	PossUnsafe	0.156	Unsafe	+2.309
no_loops_unsafe	PossUnsafe	0.016	Unsafe	+0.140
simple_loop_unsafe	PossUnsafe	0.109	Unsafe	+0.078
very_simple_unsafe	PossUnsafe	0.000	Unsafe	+0.016
csll_remove_unsafe	PossUnsafe	0.499	Unsafe	+1.342
cleanup_isochresourcedata	PossUnsafe	0.796	Unsafe	+23.306
array_in_formal	PossUnsafe	0.016	Unsafe	+0.094
deref_NULL	PossUnsafe	0.016	Unsafe	+0.031
free_free	PossUnsafe	0.000	Unsafe	+0.016
free_local	PossUnsafe	0.000	Unsafe	+0.047
if_pointer	PossUnsafe	0.000	Unsafe	+0.016
sized_arrays	PossUnsafe	0.016	Unsafe	+0.078
store_to_0	PossUnsafe	0.000	Unsafe	+0.031
sll_copy_unsafe	PossUnsafe	0.218	Unsafe	+2.293
list_of_objects	PossUnsafe	0.140	Safe	+0.230+5.975

KLEE [8], DART [20], CUTE [32] and SAGE [21] are well tuned to rapidly search large code bases looking for memory violations, assertion violations and arithmetic bugs. They could benefit from the reduction in state-space that searching an abstract transition system provides, but we preferred the guarantee that all paths were searched up to a specific depth and thus did not use these techniques.

Instead of bounding the search space using an abstract transition system, a symbolic testing engine could use a heuristic that guides it to reach certain program points. Ma, et al [28] explore this approach but do not attempt to use it to guide another analysis.

Our approach to refinement is complementary to the pattern discovery and synthesis such as [3], but refinement is not the only way to improve the widening. By adding more information to the abstraction, such as numerics [29, 30], the proofs will become more likely to succeed. This does not preclude a refinement phase, however. This would reduce the number of times refinement was needed, but widening still loses data values and thus might lose information such as sortedness of a fixed length list that our counter-example generation would know.

An abstraction refinement technique applicable to shape analysis has been proposed [27]. This technique refines the generalization of predicate abstraction [31] used by TVLA [26]. Rather than being guided by counter-examples, refinement is directed by either the syntax of an asserted formula with an indeterminate valuation, or by detection of precision loss by the abstraction during the proof attempt, without an indication that the lost precision is relevant to the proof failure.

A problem similar to feasibility checking has been investigated in the context of TVLA [19]. There, a bounded breadth-first search through program paths and a bounded model finder for first-order logic is used, in contrast to out single search encoded into SMT. It seems likely that this feasibility checker could be combined with our diagnosis technique to also obtain an abstraction failure diagnosis for shape analyses based on 3-valued logic.

8 Conclusion

We have presented a method for diagnosing abstraction failure in separation logic-based analyses. To do this, we use a new algorithm to pinpoint where abstraction failed based on a concrete counter-example. We generate this concrete counter-example with a bounded model checker that precisely analyzes abstract transition systems. These techniques have been implemented and evaluated using a pattern-based abstraction refinement scheme in SLAyer, a tool for automated analysis of low-level C programs, and have become an invaluable aid in debugging failed SLAyer runs and refining the definition of abstraction. With this contribution, we look forward to finding new automatic refinement algorithms that significantly improve the capacity and precision of shape analyses.

References

1. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with SLAM. Commun. ACM 54(7) (2011)
2. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In: SPIN (2001)
3. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W., Wies, T., Yang, H.: Shape analysis for composite data structures. In: CAV (2007)
4. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In: APLAS (2005)
5. Berdine, J., Cook, B., Ishtiaq, S.: SLAyer: Memory safety for systems-level code. In: CAV (2011)
6. Berdine, J., Cox, A., Ishtiaq, S., Wintersteiger, C.: Diagnosing abstraction failure for separation logic–based analyses. Tech. Rep. MSR-TR-2012-44, Microsoft Research, Cambridge (April 2012)
7. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: TACAS (1999)
8. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI (2008)

9. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: Automatically generating inputs of death. In: CCS (2006)
10. Calcagno, C., Yang, H., O'Hearn, P.W.: Computability and complexity results for a spatial assertion language for data structures. In: FSTTCS (2001)
11. Chaki, S., Clarke, E.M., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. IEEE Trans. Software Eng. 30(6) (2004)
12. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS (2004)
13. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV (2000)
14. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. 16 (1994)
15. Clarke, E.M., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: TACAS (2005)
16. Colón, M., Uribe, T.E.: Generating finite-state abstractions of reactive systems using decision procedures. In: CAV (1998)
17. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS (2008)
18. Elkarablieh, B., Godefroid, P., Levin, M.Y.: Precise pointer reasoning for dynamic test generation. In: ISSTA (2009)
19. Erez, G.: Generating concrete counterexamples for sound abstract interpretation. Master's thesis, School of Computer Science, Tel-Aviv University, Israel (2004)
20. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: PLDI (2005)
21. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: NDSS (2008)
22. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: CAV (1997)
23. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software verification with BLAST. In: SPIN (2003)
24. Ishtiaq, S., O'Hearn, P.W.: BI as an assertion language for mutable data structures. In: POPL (2001)
25. Jussila, T., Biere, A.: Compressing BMC encodings with QBF. Electr. Notes Theor. Comput. Sci. 174(3), 45–56 (2007)
26. Lev-Ami, T., Sagiv, S.: TVLA: A system for implementing static analyses. In: SAS (2000)
27. Loginov, A., Reps, T.W., Sagiv, S.: Abstraction refinement via inductive learning. In: CAV (2005)
28. Ma, K.K., Yit Phang, K., Foster, J., Hicks, M.: Directed symbolic execution. In: SAS (2011)
29. Magill, S., Berdine, J., Clarke, E.M., Cook, B.: Arithmetic strengthening for shape analysis. In: SAS (2007)
30. Magill, S., Tsai, M.H., Lee, P., Tsay, Y.K.: Automatic numeric abstractions for heap-manipulating programs. In: POPL (2010)
31. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3) (2002)
32. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. SIGSOFT Softw. Eng. Notes 30 (2005)
33. Wintersteiger, C.M., Hamadi, Y., de Moura, L.M.: Efficiently solving quantified bit-vector formulas. In: FMCAD (2010)

A Encoding details

Consider the command sequence $C_1; C_2$ first. We first translate C_1, passing it a continuation representing the translation of C_2. The state passed to command C_1 is the input state σ. The execution of this command produces a state σ', which is passed to the second command by constructing the appropriate continuation, i.e,

$$[\![c_1; c_2]\!]^{\mathrm{C}}\ \sigma\ sk\ ek = [\![c_1]\!]^{\mathrm{C}}\ \sigma\ (\lambda\sigma'.\ [\![c_2]\!]^{\mathrm{C}}\ \sigma'\ sk\ ek)\ ek\ .$$

The encoding of an `assume` command adds the translation of the assumed (pure) expression E in the current state σ to the rest of the constraints. The encoding of `assert` is similar to `assume`, except that it uses both the error continuation ek and the normal continuation sk. It transitions to error if the expression asserted is not true, otherwise it behaves like an `assume` command.

$$\begin{aligned}
[\![\texttt{assume (E)}]\!]^{\mathrm{C}}\ \sigma\ sk\ ek &= [\![E]\!]^{\mathrm{Exp}}\ \sigma \wedge (sk\ \sigma) \\
[\![\texttt{assert (E)}]\!]^{\mathrm{C}}\ \sigma\ sk\ ek &= ite([\![E]\!]^{\mathrm{Exp}}\ \sigma\,,\ (sk\ \sigma)\,,\ (ek\ \sigma)) \\
[\![\texttt{x = nondet ()}]\!]^{\mathrm{C}}\ \sigma\ sk\ ek &= \mathbf{let}\ \langle vars, heap, alloc, objsize\rangle\ =\ \sigma\ \mathbf{in} \\
&\qquad (sk\ \langle vars \oplus [v \mapsto (gensym\ ())], heap, alloc, objsize\rangle) \\
[\![\texttt{x = E}]\!]^{\mathrm{C}}\ \sigma\ sk\ ek &= \mathbf{let}\ \langle vars, heap, alloc, objsize\rangle\ =\ \sigma\ \mathbf{in} \\
&\qquad (sk\ \langle vars \oplus [v \mapsto [\![E]\!]^{\mathrm{Exp}}\ \sigma], heap, alloc, objsize\rangle)
\end{aligned}$$

Fig. 3. Encoding of Pure Commands

Effectively, other commands are similar except that they typically modify the threaded state as well as emit constraints. The most interesting of these are the `malloc`, and `free` and `store` rules. `malloc` produces a new function for $\sigma(alloc_l)$ and it sets $\sigma(alloc_u pd_l)$ to $true$. It makes use of a fresh variable to store the location. We cannot simply constrain the target variable x the way we constrain f because x may have already been assigned a value and thus is not unconstrained. By introducing a fresh variable, f, constraining it and updating x to be equal to f, we can achieve the desired effect. The seemingly odd constraint that $f \leq f + s$, given that $s \geq 0$ exists because of the modular behavior of arithmetic in the bit-vector theory. Without this constraint, memory would be allowed to wrap around past zero. While this behavior should be prohibited by the constraint from init that location 0 is always deallocated, adding this constraint provides performance benefits.

`free` is similar to `malloc` except that it relies upon the values in the $objsize$ array instead of the $alloc$ array. It requires that the freed address have a value other than negative one, whose value indicates no value in the size array. This value in $size$ indicates how many successive entries in $alloc$, starting at address x, need to be set back to $false$.

`store` checks the precondition v, which is that the memory at the target address is allocated. If the precondition holds, continue with the updated state where *heap* has been assigned to a new function. If the precondition does not hold, continue to error assuming that the state was not updated.

$$\begin{aligned}[\![\texttt{*x = y}]\!]^{\mathrm{C}}\ \sigma\ sk\ ek = {} & \textbf{let}\ \langle vars, heap, alloc, objsize\rangle\ =\ \sigma\ \textbf{in} \\ & \textbf{let}\ heap'\ =\ \lambda a.\ ite(a = x\,,\ y\,,\ (heap\ a))\ \textbf{in} \\ & ite((alloc\ x)\,,\ (sk\ \langle vars, heap', alloc, objsize\rangle)\,,\ (ek\ \sigma))\end{aligned}$$

The corresponding `load` command is encoded as

$$\begin{aligned}[\![\texttt{x = *y}]\!]^{\mathrm{C}}\ \sigma\ sk\ ek = {} & \textbf{let}\ \langle vars, heap, alloc, objsize\rangle\ =\ \sigma\ \textbf{in} \\ & \textbf{let}\ \sigma'\ =\ \langle vars \oplus [x \mapsto (heap\ y)], heap, alloc, objsize\rangle\ \textbf{in} \\ & ite((alloc\ y)\,,\ (sk\ \sigma')\,,\ (ek\ \sigma))\ .\end{aligned}$$