Heap-Dependent Expressions in Separation Logic

Jan Smans, Bart Jacobs, Frank Piessens

Heap-dependent Expressions in Separation Logic

Jan Smans, Bart Jacobs*, and Frank Piessens

Katholieke Universiteit Leuven, Belgium
{jan.smans,bart.jacobs,frank.piessens}@cs.kuleuven.be

Abstract. Separation logic is a popular specification language for imperative programs where the heap can only be mentioned through points-to assertions. However, separation logic's take on assertions does not match well with the classical view of assertions as boolean, side effect-free, potentially heap-dependent expressions from the host programming language familiar to many developers.
In this paper, we propose a variant of separation logic where side effect-free expressions from the host programming language, such as pointer dereferences and invocations of pure methods, can be used in assertions. We modify the symbolic execution-based verification algorithm used in Smallfoot to support mechanized checking of our variant of separation logic. We have implemented this algorithm in a tool and used the tool to verify some interesting programming patterns.

1 Introduction

The design of many specification languages centers around the idea that specifications should resemble the host programming language, in order to make it easy for developers to adopt and learn the specification language and to provide a straightforward semantics for run-time checking. Examples of such specification languages include the Java Modeling Language [1] and Spec# [2], where field dereferences and certain method calls can be used freely within contracts.

Over the past couple of years, separation logic [3] has proven to be a promising, powerful alternative to traditional specification formalisms. However, contrary to for instance JML and Spec#, separation logic assertions are quite different from the host programming language, in particular because the heap can only be mentioned through points-to assertions and expressions cannot mention field dereferences.

In this paper, we try to achieve the best of both worlds, by combining the power of separation logic with the programmer-friendly notation offered by traditional specification languages. In particular, we propose a variant of separation logic where side effect-free, potentially heap-dependent expressions from the host programming language can freely be mentioned inside specifications. In addition, we port the symbolic execution-based verification algorithm used in Smallfoot [4] to our variant of separation logic such that program correctness can be checked mechanically.

* Bart Jacobs is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO).

Supporting heap-dependent expressions in separation logic is challenging for a number of reasons. First of all, as pure methods can be used in specifications, the question arises of how to frame their return values. Secondly, assertions can be ill-defined, for example because the assertion dereferences a pointer while it does not have permission to do so. In this paper, we solve the former challenge by encoding the fact that a pure method's return value depends only on values stored in the heap covered by the precondition. We solve the latter challenge by performing additional checks when assuming and producing assertions.

In summary, the contributions of this paper are as follows:

- We propose a variant of separation logic where heap-dependent expressions from the host programming language, in particular pointer dereferences and invocations of pure methods, can be used in specifications.
- We modify Smallfoot's symbolic execution-based verification algorithm to support mechanized checking of our variant of separation logic.
- We implemented our algorithm in a tool and report on experience in verifying some interesting programming patterns.

The remainder of this paper is structured as follows. Section 2 introduces our variant of separation logic and applies it in an example. In Section 3, we propose a verification algorithm for the specification language introduced in Section 2. Finally, we discuss experience with a verifier prototype, compare with related work and conclude in Sections 4, 5 and 6.

2 Separation Logic with Side effect-free Expressions

We describe our variant of separation logic in the context of the imperative language of Figure 1. In this figure, overlining indicates repetition; annotations are highlighted by a gray background.

$$
\begin{array}{lll}
\mathit{program} & ::= & \overline{\mathit{decl}}\ \overline{s} \\
\mathit{decl} & ::= & \mathit{func} \mid \mathit{purefunc} \mid \mathit{predicate} \\
\mathit{func} & ::= & \textbf{func}\ f(\overline{x})\ \ \textbf{requires}\ A;\ \textbf{ensures}\ A;\ \{\ \overline{s}\ \textbf{return}\ e;\ \} \\
\mathit{purefunc} & ::= & \ \textbf{pure}\ \ \textbf{func}\ p(\overline{x})\ \ \textbf{requires}\ A;\ \{\ \textbf{return}\ e;\ \} \\
\mathit{predicate} & ::= & \ \textbf{predicate}\ q(\overline{x}) = A; \\
s & ::= & x := \textbf{cons}(\overline{e});\ \mid x := e;\ \mid [e] := e;\ \mid x := f(\overline{e});\ \mid \textbf{free}\ e;\ \mid \\
 & & \quad \textbf{assert}\ e = e;\ \mid \textbf{if}(e = e)\ \{\ \overline{s}\ \}\ \textbf{else}\ \{\ \overline{s}\ \} \mid\ \textbf{open}\ q(\overline{e});\ \mid \textbf{close}\ q(\overline{e}); \\
e & ::= & x \mid c \mid [e] \mid p(\overline{e}) \mid\ \textbf{open}\ q(\overline{e})\ \textbf{in}\ e \mid e\ \mathit{op}\ e \mid\ \textbf{old}(e) \\
A & ::= & \ q(\overline{e}) \mid \textbf{acc}(e) \mid A * A \mid e = e \mid \textbf{untouched}(A)
\end{array}
$$

Fig. 1. A C-like language with side effect-free, heap-dependent expressions and separation logic annotations.

A program consists of a number of declarations and a main routine $\overline{s}$. A declaration is either a mutator function, a pure function or a predicate definition. Each mutator function has a corresponding contract, consisting of a pre- and postcondition, and a method body, consisting of a number of statements followed by a return statement. Each pure function has a corresponding precondition and a method body, returning a side effect-free expression (that can potentially call the pure function itself). A predicate definition assigns a name to an assertion. A statement is either a memory allocation, a variable update, a heap update, a mutator function call, a free statement, an assert statement, an if-then-else statement, or an open or close statement. Note that the last two statements are ghost statements: they have no runtime effect and are only needed to indicate to the program verifier when folding and unfolding of predicates is required (discussed in Section 3). An expression is a variable, a constant, a pointer dereference, a pure method call, an open expression, an operator expression or an old expression. Note that all expressions are side effect-free; however some of them depend on the heap. An assertion is either a predicate assertion, an access assertion, a separating conjunction, an equality between expressions or an untouched assertion. An access assertion $\mathbf{acc}(e)$ denotes the permission to dereference e. In classical separation logic, $\mathbf{acc}(e)$ would be denoted by $e \mapsto \_$. An untouched assertion $\mathbf{untouched}(A)$ is a two-state assertion that holds if the values in the heap covered by A are the same in both the pre- and post-state. In the remainder of this paper, we use **true** as syntactic sugar for the assertion $0 = 0$, and **false** as syntactic sugar for $1 = 0$. In the remainder of this paper, we consider only well-formed programs (Definition 1). In our implementation, well-formedness is checked using a simple syntactic analysis.

Definition 1. *A program is* well-formed *if all of the following hold:*

- *Predicate, mutator function and pure function names are unique within the program. Parameter names are unique within a declaration.*
- *The free variables of a function's body and contract are the function parameters. Postconditions can additionally mention the variable* **result**. *The free variables of a predicate's body are the predicate's parameters. The main routine has no free variables.*
- *The program only mentions functions and predicates declared in the program text. The number of actual arguments in a function call or predicate assertion is equal to the number of formal parameters in the corresponding declaration.*
- *Old expressions and untouched assertions only appear in postconditions.*

Let us take a look at the example program of Figure 2. This program declares a predicate, a pure function and a number of mutator functions for dealing with cells, together with a main routine that uses the aforementioned functions to create and interact with cells. The pure function *get* returns the value of the cell referred to by c. Its precondition demands that c is a valid cell data structure. Note that pure functions do not have postconditions, as they cannot modify the program state. *create_cell* creates a new cell data structure with value 0. *create_cell*'s postcondition not only states that the result is a valid cell, but also

expresses that resulting cell holds 0 via the pure function *get*. *inc* increments the value of a cell by one. Note that an old expression is used to relate the pre- and post-state. *copy* copies the value of cell d to cell c. The untouched assertion in its postcondition expresses that the heap covered by $cell(d)$ is not modified by *copy*. Client code can use the latter information to frame pure methods that depend on $cell(d)$. For example, clients can prove that $get(d)$'s return value is the same before and after calling *copy*.

```
predicate cell(c) = acc(c);

pure func get(c)
    requires cell(c);
{ return open cell(c) in [c]; }

func create_cell()
    requires true; ensures cell(result) * get(result) = 0;
{ c := cons(0); close cell(c); return c; }

func inc(c)
    requires cell(c);
    ensures cell(c) * get(c) = old(get(c)) + 1;
{ open cell(c); [c] := [c] + 1; close cell(c); return 0; }

func copy(c, d)
    requires cell(c) * cell(d);
    ensures cell(c) * cell(d) * get(c) = get(d) * untouched(cell(d));
{ open cell(c); [c] := get(d); close cell(c); return 0; }

func dispose(c)
    requires cell(c); ensures true;
{ open cell(c); free(c); }

c1 := create_cell(); inc(c1);
c2 := create_cell(); inc(c2);
assert get(c1) = 1;
dispose(c1); dispose(c2);
```

Fig. 2. An annotated program written in the language of Figure 1. Annotations are highlighted by a gray background.

The main routine creates two cells, c_1 and c_2, updates their values, checks that the value of c_1 is 1, and disposes both cells. Note that in order to prove that

the assertion never fails, one must show that creating and modifying c_2 does not affect the return value of $get(c_1)$.

The function bodies in Figure 2 contain open and close ghost statements. These statements instruct the program verifier to respectively unfold and fold predicates during symbolic execution. For example, the close statement in the body of *create_cell* removes the body of the predicate *cell* from the symbolic heap and replaces it with the predicate itself, thereby establishing the postcondition. A more detailed description of symbolic execution and open and close statements will be given in Section 3.

3 Verification

In this section, we describe the symbolic execution-based verification algorithm for our variant of separation logic. After defining how we represent program states symbolically (section 3.1), we define symbolic evaluation and execution of expressions and statements (section 3.2). Based on these definitions, we define what it means for a program to be valid (section 3.3).

3.1 Symbolic State

A symbolic state is a four-tuple (h, g, γ, π) consisting of a symbolic heap h, a symbolic pre-state heap g, a symbolic store γ and a path condition π. A symbolic heap is a multiset of heap chunks, where each heap chunk $q[s](\bar{t})$ consists of a predicate name q, a first-order term s and a list of first-order terms $\bar{t}$. We refer to the term s as the snapshot of the heap chunk. A symbolic store is a partial function from variables to first-order terms. Finally, a path condition is a set of first-order formulas, describing the conditions that hold on the current execution path.

In the remainder of this section, we describe the symbolic execution algorithm itself. The core of this algorithm consists of 4 functions: eval, produce, consume and exec. These functions respectively represent symbolic evaluation of expressions, assuming and checking assertions and symbolic execution of statements. All aforementioned functions are written in continuation passing style. That is, each function takes a continuation parameter (typically called Q) that represents the work to be done on the current path. Their signatures are as follows:

$$\begin{aligned}
&\mathsf{eval} : H \times H \times \Gamma \times \Pi \times e \times (\mathcal{T} \times \Pi \to \mathbb{B}) \to \mathbb{B} \\
&\mathsf{produce} : H \times H \times \Gamma \times \Pi \times \mathcal{T} \times A \times (H \times \Pi \to \mathbb{B}) \to \mathbb{B} \\
&\mathsf{consume} : H \times H \times \Gamma \times \Pi \times A \times (H \times \Pi \times \mathcal{T} \to \mathbb{B}) \to \mathbb{B} \\
&\mathsf{exec} : H \times H \times \Gamma \times \Pi \times s \times (H \times \Gamma \times \Pi \to \mathbb{B}) \to \mathbb{B}
\end{aligned}$$

In the above signatures, H stands for the set of symbolic heaps, Γ for the set of symbolic stores, Π for the set of path conditions, $\mathcal{T}$ for the set of first-order terms and $\mathbb{B}$ for the set of booleans. Note that each of the aforementioned functions returns a boolean, indicating whether symbolic execution was successful.

3.2 Symbolic Execution

Preliminaries The symbolic state represents symbolic values as first-order terms and information about those values as first-order formulas. In the algorithm, we use a first-order logic with equality. The signature of the logic contains a number of built-in functions, including *unit*, *pair*, *fst* and *snd*, which are used to create snapshots. A snapshot uniquely determines the values in the heap covered by a predicate. *unit* is the empty snapshot, while $pair(a, b)$ combines snapshots a and b. The functions are axiomatized as follows:

$$\forall a, b \bullet fst(pair(a, b)) = a \qquad \forall a, b \bullet snd(pair(a, b)) = b$$

We do not explicitly mention these axioms in our algorithm. Instead, we write $\pi \vdash \phi$ (denoting that formula ϕ is provable from π) as a shorthand for $\pi \cup T \vdash \phi$, where T is the theory containing the two axioms described above. Our implementation relies on the Z3 SMT solver [5] to discharge such proof tasks.

A key question our approach has to answer is how to encode pure functions during verification in a way that allows us to frame their return values. Like other verification approaches [6–11], we encode a pure function as a first-order function in the verification logic and encode a call of a pure function as an application of the corresponding first-order function. The key to solving the issue of framing is the fact that a pure function's return value can only depend on memory locations covered by its precondition. This dependence on part of the heap is encoded as an additional function parameter. As we will show in Figure 3, this parameter is the snapshot of the function's precondition. For example, the signature of the function symbol for the pure function *get* of Figure 2 is $get : \mathcal{T} \times \mathcal{T} \rightarrow \mathcal{T}$. The first parameter represents the values in the heap covered by the precondition, while the second parameter corresponds to the pure function's parameter c.

Note that the functions produce and consume do not contain cases for $\mathbf{acc}(e)$. Instead, the algorithm considers **acc** to be just another predicate with one parameter.

If at a certain point during symbolic execution the path condition is inconsistent, then that point is not reachable during a concrete execution of the program. In our implementation, we check consistency of the path condition whenever a new formula is added to it. If adding a new formula makes the path condition inconsistent, we simply stop symbolic execution and return *true* (indicating that symbolic execution succeeded). To avoid cluttering the rules, we do not explicitly show consistency checks in the definitions of eval, consume, produce and exec.

Symbolic evaluation of expressions $\mathsf{eval}(h, g, \gamma, \pi, e, Q)$ (defined in Figure 3) evaluates the expression e in symbolic state (h, g, γ, π) and passes both the resulting term and a potentially updated path condition to the continuation Q. More specifically, symbolic evaluation of a variable x corresponds to looking up x in the symbolic store and passing the resulting term to the continuation. A constant evaluates to itself. Dereferencing a pointer is allowed only if the thread has permission to do so. To check whether the thread has permission,

the algorithm looks in the symbolic heap for a matching chunk. If a matching chunk is found, that chunk's snapshot is passed to the continuation; otherwise, $\mathsf{eval}(h, g, \gamma, \pi, [e], Q)$ fails.

$$
\begin{array}{l}
\mathsf{eval}(h, g, \gamma, \pi, x, Q) \equiv Q(\gamma(x), \pi) \\
\\
\mathsf{eval}(h, g, \gamma, \pi, c, Q) \equiv Q(c, \pi) \\
\\
\mathsf{eval}(h, g, \gamma, \pi, [e], Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e, (\lambda t, \pi' \bullet \\
\qquad \textbf{let } \mathit{matches} = \{\ \mathit{acc}[t_s](t_1) \in h \mid \pi \vdash t_1 = t\ \} \textbf{ in} \\
\qquad \exists \mathit{acc}[t_s](t_1) \in \mathit{matches} \bullet Q(t_s, \pi'))) \\
\\
\mathsf{eval}(h, g, \gamma, \pi, p(e_1, \ldots, e_n), Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e_1, (\lambda t_1, \pi_1 \bullet \ldots \mathsf{eval}(h, g, \gamma, \pi_{n-1}, e_n, (\lambda t_n, \pi_n \bullet \\
\qquad \mathsf{consume}(h, g, \{(x_1, t_1), \ldots, (x_n, t_n)\}, \pi_n, \mathsf{precondition}(p), (\lambda h', \pi', s \bullet \\
\qquad\quad \mathsf{if}\ p\text{'s body visible } \mathsf{then} \\
\qquad\qquad \mathsf{produce}(h', g, \{(x_1, t_1), \ldots, (x_n, t_n)\}, \pi', s, \mathsf{precondition}(p), (\lambda h'', \pi'' \bullet \\
\qquad\qquad\quad \mathsf{eval}(h'', g, \{(x_1, t_1), \ldots, (x_n, t_n)\}, \pi'', \mathsf{body}(p), (\lambda t, \pi''' \bullet \\
\qquad\qquad\qquad Q(p(s, t_1, \ldots, t_n), \pi''' \cup \{p(s, t_1, \ldots, t_n) = t\})))))) \\
\qquad\quad \mathsf{else} \\
\qquad\qquad Q(p(s, t_1, \ldots, t_n), \pi'))))))) \\
\\
\mathsf{eval}(h, g, \gamma, \pi, \textbf{open } q(e_1, \ldots, e_n) \textbf{ in } e, Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e_1, (\lambda t_1, \pi_1 \bullet \ldots (\lambda t_n, \pi_n \bullet \\
\qquad \mathsf{consume}(h, g, \gamma, \pi_n, q(e_1, \ldots, e_n), (\lambda h', \pi', s \bullet \\
\qquad\quad \mathsf{produce}(h', g, \{(x_1, t_1), \ldots, (x_n, t_n)\}, \pi', s, \mathsf{definition}(q), (\lambda h'', \pi'' \bullet \\
\qquad\qquad \mathsf{eval}(h', g, \gamma, \pi'', e, Q)))))))) \\
\\
\mathsf{eval}(h, g, \gamma, \pi, e_1\ \mathit{op}\ e_2, Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e_1, (\lambda t_1, \pi_1 \bullet \mathsf{eval}(h, g, \gamma, \pi_1, e_2, (\lambda t_2, \pi_2 \bullet \\
\qquad Q(t_1\ \mathit{op}\ t_2, \pi_2))))) \\
\\
\mathsf{eval}(h, g, \gamma, \pi, \textbf{old}(e), Q) \equiv \mathsf{eval}(g, g, \gamma, \pi, e, Q)
\end{array}
$$

Fig. 3. Symbolic evaluation of expressions.

Calling a pure function $p(e_1, \ldots, e_n)$ is allowed only if its precondition holds. Our algorithm checks whether the precondition holds by consuming it. Note that consuming the precondition not only returns an updated heap and path condition, but also a snapshot s. This snapshot is used as the first parameter in the application of the corresponding first-order function. Note that the algorithm branches on the fact whether p's body is visible. That is, if the body is visible, an assumption stating that evaluation of $p(e_1, \ldots, e_n)$ equals evaluation of its body is added to the path condition passed to the continuation. An opening expression **open** $q(e_1, \ldots, e_n)$ **in** e evaluates the expression e in a context

where the predicate $q(e_1, \ldots, e_n)$ is replaced by its body. To symbolically evaluate an binary operation $e_1 \, op \, e_2$, the algorithm applies the operation to the corresponding symbolic values, t_1 and t_2. Evaluation of an old expression $\mathbf{old}(e)$ corresponds to evaluating e in the pre-state heap g.

Symbolic production of assertions $\mathsf{produce}(h, g, \gamma, \pi, s, A, Q)$ assumes the assertion A in the symbolic state (h, g, γ, π) based on snapshot s and passes a potentially updated heap and path condition to its continuation Q. The parameter s is used to determine the snapshots of heap chunks created during production. The function $\mathsf{produce}$ is defined in terms of the helper function $\mathsf{produce}'$ as follows:

$$\mathsf{produce}(h, g, \gamma, \pi, s, A, Q) \equiv \mathsf{produce}'(\emptyset, g, \gamma, \pi, s, A, (\lambda h', \pi' \bullet Q(h \uplus h', \pi')))$$

$\mathsf{produce}'$ starts with an empty heap to ensure that assertions are self-framing. That is, the assertion A should only dereference a pointer if A itself demands access to that pointer.

Figure 4 shows the definition of $\mathsf{produce}'$. To produce a predicate assertion $q(e_1, \ldots, e_n)$, we add a predicate chunk for q to the symbolic store with snapshot s. To produce a separating conjunction $A_1 * A_2$, we must first produce A_1 and afterwards produce A_2 in the resulting symbolic state. Note that the snapshot s is split up into two pieces using the functions *fst* and *snd*. Producing an equality $e_1 = e_2$ comes down to adding the assumption that the values of both expressions are equal to the path condition. Finally, to produce $\mathbf{untouched}(A)$, we consume A in both the current symbolic heap h and the pre-state heap g, and assume that the resulting snapshots are equal.

$$\begin{array}{l}
\mathsf{produce}'(h, g, \gamma, \pi, s, q(e_1, \ldots, e_n), Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e_1, (\lambda t_1, \pi_1 \bullet \ldots \mathsf{eval}(h, g, \gamma, \pi_{n-1}, e_n, (\lambda t_n, \pi_n \bullet \\
\qquad Q(h \uplus \{q[s](t_1, \ldots, t_n)\}, \pi_n)))))
\end{array}$$

$$\begin{array}{l}
\mathsf{produce}'(h, g, \gamma, \pi, s, A_1 * A_2, Q) \equiv \\
\quad \mathsf{produce}'(h, g, \gamma, \pi, \mathit{fst}(s), A_1, (\lambda h', \pi' \bullet \\
\qquad \mathsf{produce}'(h', g, \gamma, \pi', \mathit{snd}(s), A_2, Q)))
\end{array}$$

$$\begin{array}{l}
\mathsf{produce}'(h, g, \gamma, \pi, s, e_1 = e_2, Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e_1, (\lambda t_1, \pi_1 \bullet \mathsf{eval}(h, g, \gamma, \pi_1, e_2, (\lambda t_2, \pi_2 \bullet \\
\qquad Q(h, \pi_2 \cup \{t_1 = t_2\})))))
\end{array}$$

$$\begin{array}{l}
\mathsf{produce}'(h, g, \gamma, \pi, s, \mathbf{untouched}(A), Q) \equiv \\
\quad \mathsf{consume}(h, g, \gamma, \pi, A, (\lambda_, _, s_1 \bullet \mathsf{consume}(g, g, \gamma, \pi, A, (\lambda_, _, s_2 \bullet \\
\qquad Q(h, \pi \cup \{s_1 = s_2\})))))
\end{array}$$

Fig. 4. Production of assertions.

Symbolic consumption of assertions Consumption is the reverse of production. $\mathsf{consume}(h, g, \gamma, \pi, A, Q)$ checks if A holds in the symbolic state (h, g, γ, π) and passes a potentially updated heap, path condition and the snapshot of the consumed heap chunks to the continuation Q. Note that "checking" of spatial assertions causes heap chunks to be removed from the symbolic heap. consume is defined in terms of the helper function $\mathsf{consume}'$ as follows:

$$\mathsf{consume}(h, g, \gamma, \pi, A, Q) \equiv \mathsf{consume}'(h, h, g, \gamma, \pi, A, Q)$$

The first symbolic heap passed to $\mathsf{consume}'$ is used for evaluating expressions, while the second represents the remainder of the original heap which is not consumed yet by the assertion.

Figure 5 shows the definition of $\mathsf{consume}'$. Consumption of a predicate assertion $q(e_1, \ldots, e_n)$ succeeds only if a heap chunk matching $q[\_](t_1, \ldots, t_n)$ exists. This heap chunk is removed from the symbolic heap and its snapshot is passed to the continuation. To consume $A_1 * A_2$, one must first consume A_1 and afterwards consume A_2. A pair containing the snapshots of A_1 and A_2 is passed to the continuation. Consumption of $e_1 = e_2$ succeeds only if both expressions are provably (from the path condition) equal and the continuation Q succeeds. Finally, consumption of $\mathbf{untouched}(A)$ succeeds only if the snapshots obtained by consuming A in both the pre- and post-state are provably equal.

$$\begin{array}{l}
\mathsf{consume}'(h, h', g, \gamma, \pi, q(e_1, \ldots, e_n), Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e_1, (\lambda t_1, \pi_1 \bullet \ldots \mathsf{eval}(h, g, \gamma, \pi_{n-1}, e_n, (\lambda t_n, \pi_n \bullet \\
\qquad \mathbf{let}\ \mathit{matches} = \{q[s](t_1', \ldots, t_1') \in h' \mid \pi_n \vdash t_1 = t_1' \wedge \ldots \wedge t_n = t_n'\}\ \mathbf{in} \\
\qquad \exists q[s](t_1', \ldots, t_1') \in \mathit{matches} \bullet Q(h' - \{q[s](t_1', \ldots, t_1')\}, \pi_n, s)))))
\end{array}$$

$$\begin{array}{l}
\mathsf{consume}'(h, h', g, \gamma, \pi, A_1 * A_2, Q) \equiv \\
\quad \mathsf{consume}'(h, h', g, \gamma, \pi, A_1, (\lambda h'', \pi', s_1 \bullet \\
\qquad \mathsf{consume}'(h, h'', g, \gamma, \pi', A_2, (\lambda h''', \pi'', s_2 \bullet Q(h''', \pi'', \mathit{pair}(s_1, s_2))))))
\end{array}$$

$$\begin{array}{l}
\mathsf{consume}'(h, h', g, \gamma, \pi, e_1 = e_2, Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e_1, (\lambda t_1, \pi_1 \bullet \mathsf{eval}(h, g, \gamma, \pi_1, e_2, (\lambda t_2, \pi_2 \bullet \\
\qquad (\pi_2 \vdash t_1 = t_2) \wedge Q(h', \pi_2)))))
\end{array}$$

$$\begin{array}{l}
\mathsf{consume}'(h, h', g, \gamma, \pi, s, \mathbf{untouched}(A), Q) \equiv \\
\quad \mathsf{consume}'(h, h, h, \gamma, \pi, A, (\lambda_, _, s_1 \bullet \mathsf{consume}'(g, g, g, \gamma, \pi, A, (\lambda_, _, s_2 \bullet \\
\qquad (\pi \vdash s_1 = s_2) \wedge Q(h', \pi)))))
\end{array}$$

Fig. 5. Consumption of assertions.

Symbolic execution of statements $\mathsf{exec}(h, g, \gamma, \pi, s, Q)$ (Figure 6) symbolically executes statement s in symbolic state (h, g, γ, π) and passes a potentially updated heap, store and path condition to the continuation Q. More specifically, a memory allocation $x := \mathbf{cons}(e_1, \ldots, e_n)$; is modeled by creating a fresh

term representing the address of the newly allocated memory. The heap is extended with n new memory locations starting at address l containing the terms t_1 to t_n. The variable x is modified to l. A variable update $x := e;$ modifies the value of the variable x in the symbolic store γ to the symbolic value of e. A heap update $[e_1] = e_2;$ is allowed only if the symbolic heap contains a chunk that matches $acc[\_](t_1)$. If such a match exists, that heap chunk's snapshot is changed to t_2; otherwise, symbolic execution fails. Symbolic execution of a mutator call $x := f(e_1, \ldots, e_n);$ consists of consuming f's precondition and producing f's postcondition afterwards. Note that a fresh term is used as the snapshot for producing the postcondition. A free statement **free** $e;$ is allowed only if the heap contains a chunk matching $acc[\_](t)$. If it does, then that chunk is removed from the heap; otherwise, symbolic execution fails. Note that the chunks produced by $\mathbf{cons}(e_1, \ldots, e_n)$ need to be freed separately. As assert statement **assert** $e_1 = e_2;$ fails if the values of e_1 and e_2 are not provably equal; otherwise, the assert statement is equivalent to skip. An if-then-else statement $\mathbf{if}(e_1 = e_2)\ \{\ \overline{s_1}\ \}\ \mathbf{else}\ \{\ \overline{s_2}\ \}$ splits symbolic execution into two branches. An open statement **open** $q(e_1, \ldots, e_n);$ removes a chunk matching $q[\_](t_1, \ldots, t_n)$ from the symbolic state and produces q's body with the snapshot of the removed chunk; if no such chunk exists, symbolic execution fails. Finally, a close statement **close** $q(e_1, \ldots, e_n);$ consumes q's body and replaces the consumed chunks with a predicate chunk for q. The snapshot obtained by consuming q's body is used as the snapshot of this new heap chunk.

3.3 Valid Program

We say that a mutator function is valid (Definition 2) if after producing the precondition for arbitrary values of the parameters and executing the mutator's body in the resulting state, consumption of the postcondition succeeds and the symbolic heap is empty. A pure function is valid (Definition 3) if evaluation of its body does not fail in the symbolic state resulting from producing the precondition for arbitrary values of the parameters. A predicate is valid (Definition 4) if production of its body does not fail for arbitrary parameters. Finally, the main routine is valid (Definition 5) if its symbolic execution succeeds in an empty heap, pre-state heap, store and path condition and the resulting heap is empty.

A program is valid if all functions and the main routine are valid. The program of Figure 2 is valid.

Definition 2. *A mutator function*

$$\mathbf{func}\ f(x_1, \ldots, x_n)\ \mathbf{requires}\ A_1;\ \mathbf{ensures}\ A_2;\ \{\ \overline{s};\ \mathbf{return}\ e;\ \}$$

is valid if the following holds:

$$\begin{array}{l}
\mathbf{let}\ (t_1, \ldots, t_n) = (\mathsf{fresh}, \ldots, \mathsf{fresh})\ \mathbf{in} \\
\mathsf{produce}(\emptyset, \emptyset, \{(x_1, t_1), \ldots, (x_n, t_n)\}, \emptyset, \mathsf{fresh}, A_1, (\lambda h, \pi\bullet \\
\quad \mathsf{exec}(h, h, \{(x_1, t_1), \ldots, (x_n, t_n)\}, \pi, \overline{s}, (\lambda h', \gamma, \pi'\bullet \\
\quad\quad \mathsf{eval}(h', h, \gamma, \pi', e, (\lambda t, \pi''\bullet \\
\quad\quad\quad \mathsf{consume}(h', h, \{(x_1, t_1), \ldots, (x_n, t_n), (\mathbf{result}, t)\}, \pi'', A_2, \\
\quad\quad\quad\quad (\lambda h'', _, _\bullet\ h'' = \emptyset)))))))
\end{array}$$

$$
\begin{array}{l}
\mathsf{exec}(h, g, \gamma, \pi, x := \mathbf{cons}(e_1, \ldots, e_n);, Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e_1, (\lambda t_1, \pi_1 \bullet \ldots \mathsf{eval}(h, g, \gamma, \pi_{n-1}, e_n, (\lambda t_n, \pi_n \bullet \\
\qquad \mathbf{let}\ l = \mathsf{fresh}\ \mathbf{in} \\
\qquad Q(h \uplus \{acc[t_1](l), \ldots acc[t_n](l+n-1)\}, \gamma[x \mapsto l], \pi_n)))))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{exec}(h, g, \gamma, \pi, x := e;, Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e, (\lambda t, \pi_1 \bullet Q(h, \gamma[x \mapsto t], \pi_1)))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{exec}(h, g, \gamma, \pi, [e_1] := e_2;, Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e_1, (\lambda t_1, \pi_1 \bullet \mathsf{eval}(h, g, \gamma, \pi_1, e_2, (\lambda t_2, \pi_2 \bullet \\
\qquad \mathbf{let}\ matches = \{acc[s](t_1') \mid \pi_2 \vdash t_1 = t_1'\}\ \mathbf{in} \\
\qquad \exists acc[s](t_1') \in matches \bullet Q(h - \{acc[s](t_1')\} \uplus \{acc[t_2](t_1)\}, \gamma, \pi_2)))))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{exec}(h, g, \gamma, \pi, x := f(e_1, \ldots, e_n), Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e_1, (\lambda t_1, \pi_1 \bullet \ldots \mathsf{eval}(h, g, \gamma, \pi_{n-1}, e_n, (\lambda t_n, \pi_n \bullet \\
\qquad \mathsf{consume}(h, g, \{(x_1, t_1), \ldots, (x_n, t_n)\}, \pi_n, \mathsf{precondition}(f), (\lambda h', \pi', \_ \bullet \\
\qquad\quad \mathbf{let}\ (s, r) = (\mathsf{fresh}, \mathsf{fresh})\ \mathbf{in} \\
\qquad\quad \mathsf{produce}(h', h, \{(x_1, t_1), \ldots, (x_n, t_n), (\mathbf{result}, r)\}, \pi', s, \mathsf{postcondition}(f), (\lambda h'', \pi'' \bullet \\
\qquad\qquad Q(h'', \gamma[x \mapsto r], \pi'')))))))))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{exec}(h, g, \gamma, \pi, \mathbf{free}\ e;, Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e, (\lambda t, \pi' \bullet \\
\qquad \mathbf{let}\ matches = \{acc[s](t') \mid \pi' \vdash t = t'\}\ \mathbf{in} \\
\qquad \exists acc[s](t') \in matches \bullet Q(h - \{acc[s](t')\}, \gamma, \pi')))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{exec}(h, g, \gamma, \pi, \mathbf{assert}\ e_1 = e_2;, Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e_1, (\lambda t_1, \pi_1 \bullet \mathsf{eval}(h, g, \gamma, \pi_1, e_2, (\lambda t_2, \pi_2 \bullet \\
\qquad (\pi_2 \vdash t_1 = t_2) \wedge Q(h, \gamma, \pi_2)))))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{exec}(h, g, \gamma, \pi, \mathbf{if}(e_1 = e_2)\ \{\ \overline{s_1}\ \}\ \mathbf{else}\ \{\ \overline{s_2}\ \}, Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e_1, (\lambda t_1, \pi_1 \bullet \mathsf{eval}(h, g, \gamma, \pi_1, e_2, (\lambda t_2, \pi_2 \bullet \\
\qquad \mathsf{exec}(h, g, \gamma, \pi_2 \cup \{t_1 = t_2\}, \overline{s_1}, Q) \wedge \mathsf{exec}(h, g, \gamma, \pi_2 \cup \{t_1 \neq t_2\}, \overline{s_2}, Q)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{exec}(h, g, \gamma, \pi, \mathbf{open}\ q(e_1, \ldots, e_n);, Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e_1, (\lambda t_1, \pi_1 \bullet \ldots \mathsf{eval}(h, g, \gamma, \pi_{n-1}, e_n, (\lambda t_n, \pi_n \bullet \\
\qquad \mathsf{consume}(h, g, \gamma, \pi_n, q(e_1, \ldots, e_n), (\lambda h', \pi', s \bullet \\
\qquad\quad \mathsf{produce}(h', g, \{(x_1, t_1), \ldots, (x_n, t_n)\}, \pi, s, \mathsf{definition}(q), (\lambda h'', \pi'' \bullet \\
\qquad\qquad Q(h'', \gamma, \pi'')))))))))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{exec}(h, g, \gamma, \pi, \mathbf{close}\ q(e_1, \ldots, e_n);, Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e_1, (\lambda t_1, \pi_1 \bullet \ldots \mathsf{eval}(h, g, \gamma, \pi_{n-1}, e_n, (\lambda t_n, \pi_n \bullet \\
\qquad \mathsf{consume}(h, g, \{(x_1, t_1), \ldots, (x_n, t_n)\}, \pi_n, \mathsf{definition}(q), (\lambda h', \pi', s \bullet \\
\qquad\quad Q(h' \uplus \{q[s](t_1, \ldots t_n)\}, \gamma, \pi')))))))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{exec}(h, g, \gamma, \pi, s_0\ \overline{s}, Q) \equiv \\
\quad \mathsf{exec}(h, g, \gamma, \pi, s_0, (\lambda h', \gamma', \pi' \bullet \mathsf{exec}(h', g, \gamma', \pi', \overline{s}, Q)))
\end{array}
$$

Fig. 6. Symbolic execution of statements.

Definition 3. *A pure function*

$$\textbf{pure func } p(x_1, \ldots, x_n) \textbf{ requires } A;\ \{\ \textbf{return } e;\ \}$$

is valid if the following holds:

$$\begin{array}{l}\textbf{let } (t_1, \ldots, t_n) = (\mathsf{fresh}, \ldots, \mathsf{fresh}) \textbf{ in}\\ \mathsf{produce}(\emptyset, \emptyset, \{(x_1, t_1), \ldots, (x_n, t_n)\}, \emptyset, \mathsf{fresh}, A, (\lambda h, \pi \bullet\\ \quad \mathsf{eval}(h, h, \{(x_1, t_1), \ldots, (x_n, t_n)\}, \pi, e, (\lambda \_, \_ \bullet \mathit{true}))))\end{array}$$

Definition 4. *A predicate*

$$\textbf{predicate } q(x_1, \ldots, x_n) = A;$$

is valid if the following holds:

$$\mathsf{produce}(\emptyset, \emptyset, \{(x_1, \mathsf{fresh}), \ldots, (x_n, \mathsf{fresh})\}, \emptyset, \mathsf{fresh}, A, (\lambda h, \pi \bullet \mathit{true}))$$

Definition 5. *A main routine $\overline{s}$ is valid if the following holds:*

$$\mathsf{exec}(\emptyset, \emptyset, \emptyset, \emptyset, \overline{s}, (\lambda h, \_, \_ \bullet h = \emptyset))$$

Pure Method Termination It is essential for the soundness of our approach that pure methods terminate. Verification therefore includes a phase that checks sufficient conditions for pure method termination. Specifically, it is checked for each pure method call in a pure method body that either (1) the callee is defined earlier in the program text, or (2) the call is in the body of an **open** expression, or (3) there is some symbolic heap chunk that is not consumed by the precondition of the call. This ensures that at each call, either the size of the symbolic heap decreases, or the *derivation depth* decreases (i.e. the number of close operations required to construct the heap from one that contains only field chunks), or the position in the program text decreases. Since the size and the derivation depth are always finite and a pure method cannot increase the size or the derivation depth of the symbolic heap, this ensures termination. Contrary to pure functions, mutator functions are not required to terminate.

4 Implementation and Experience

We have implemented the algorithm described in Section 3 in a tool. The source code (F#), binaries and a number of examples are available from the author's website `http://www.cs.kuleuven.be/~jans/speccheck`. Instead of the C-like language used in the paper, the tool supports a larger assertion language (e.g. conditional assertions) for a small subset of C#. To check whether a first-order formula is derivable from the path condition, we use the Z3 SMT solver [5]. Our verifier prototype has been used to verify a number of small programming patterns, including aggregate objects and iterator. These programs together with their verification times are shown in Table 1. To help developers diagnose verification errors, our verifier includes a symbolic debugger (shown in Figure 7). When verification fails, the developer can inspect the components of the symbolic states encountered during symbolic execution on the path to the failure.

example	cell	abstract cell	cell50	interval	iterator
time taken (seconds)	0.005	0.008	0.18	0.03	0.01

Table 1. Programs verified using the verifier prototype together with the verification time (seconds). The experiments were executed on a standard desktop machine with a 2.66 Ghz processor and 4 GB of RAM running Windows Vista. cell50 is the similar to the main routine of Figure 2, except that 50 intermediate cells are created and updated instead of 1.

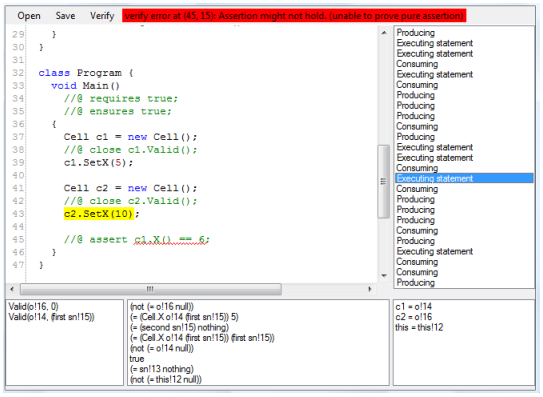

Fig. 7. A screenshot of the verifier prototype. Developers can use the symbolic debugger in the IDE to diagnose verification errors and inspect the symbolic state at each program point. The box on the right of the screen contains a list of symbolic states encountered during symbolic execution. The boxes at the bottom of the screen show the symbolic state (from left to right: the symbolic heap, the path condition and the symbolic store) at a particular program point.

5 Related Work

Separation logic [3] extends Hoare logic with three new assertions: **emp**, separating conjunction and points-to. Our assertion **acc**(e) is similar to separation logic's points-to assertion $e \mapsto \_$, which denotes that the current thread has permission to access the memory at address e (without constraining the value at that address). The key difference between classical separation logic and our variant is that we allow expressions that are used inside assertions to depend on the heap. Examples of heap-dependent expressions include pointer dereferences and function invocations. Note that we do not claim that our variant has additional expressive power. Instead, the difference leads to a different style of writing specifications. More specifically, where we use pure methods to express the state of an object, classical separation logic relies on predicate parameters. For example, the contract of *inc* of Figure 2 would be written in separation logic as: **requires** $cell(c, X)$; **ensures** $cell(c, X + 1)$;. Here, X is a logical variable.

Berdine *et al.* [4] have proposed a symbolic execution-based verification algorithm for programs annotated with separation logic specifications and have implemented this algorithm for a small imperative language in Smallfoot. The algorithm described in Section 3 is a variant of the aforementioned algorithm. In particular, the idea of dividing the symbolic state in spatial and pure assertions (the symbolic heap and the path condition respectively) and the rules dealing with heap access, non-heap-dependent expressions and assertions are largely similar to those in [4]. The novel aspect of our approach is the treatment of pure methods and pointer dereferences inside assertions via snapshots. Smallfoot only supports a limited number of predicates, but the developer does not need to write open and close statements as the tool has hard-coded, built-in rules for reasoning about those predicates.

jStar [12] and VeriFast [13] extend the basic ideas of Berdine *et al.* to full-fledged programming languages like Java and C. While loop invariants must be provided by the developer in our approach, jStar infers certain loop invariants automatically, provided the developer inputs the necessary abstraction rules. Using an SMT solver [5] for discharging pure queries and supporting symbolic debugging via an IDE that shows the components of the symbolic state are ideas taken from VeriFast.

As an alternative to the Smallfoot's symbolic execution-based verification algorithm, Leino and Müller [14] and Smans *et al.* [15] have proposed an approach based on verification condition generation and automated theorem proving for a variant of separation logic. However, experience has shown that approaches based on verification condition generation and automated theorem proving in general, and [14, 15] in particular, have 3 disadvantages: (1) they are slow, (2) they are unpredictable, as small changes in the specification can have a significant impact on verification time and (3) verification errors are hard to diagnose, as it is hard to determine whether the specification is flawed or whether the theorem prover is unable to prove a particular part of the verification condition. As the experiments with our verifier prototype indicate, verification times are consistently low. For example, even if we increase the number of intermediate statements in the main routine of Figure 2, the verification time only increases marginally. The main reason why verification is fast is that we reason about the heap outside of the theorem prover, and hence do not need to send quantifier-heavy formulas (in particular quantifiers about the heap) to the automated prover. These experiments thus confirm earlier results with similar algorithms [4, 12, 13]. Moreover, the developer can diagnose verification errors by inspecting the symbolic states on the path to the failure. A disadvantage of the approach presented in this paper with respect to [14, 15] is that non-separating conjunction is not supported.

Reasoning about method calls in specifications and framing their return values in particular was posed as a challenge for verification by Leavens, Leino and Müller [16]. In the context of approaches based on verification condition generation and automated theorem proving, researchers have attacked well-formedness of pure method specifications [6, 7], framing of return values [8–10] and allowing certain side effects in pure methods [11, 9]. The approach proposed in this paper

is similar to existing techniques in the sense that we also encode pure methods as functions and invocations of pure methods as function applications. Moreover, the idea of using snapshots is similar to the snapshots used in [8, 9]. However, to the best of our knowledge, this is the first paper that discusses the use of pure methods and framing of their return values in the context of separation logic (and in the context of its symbolic execution-based verification algorithm).

6 Conclusion

In this paper, we combined the expressive power of separation logic with the programmer-friendly notation of specification languages such as JML where heap-dependent expressions can be used in annotations. We proposed an algorithm to support mechanized checking for this variant of separation logic and implemented this algorithm in a tool.

References

1. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. *Behavioral Specifications of Businesses and Systems*, 1999.
2. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS*, 2004.
3. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.
4. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, 2005.
5. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.
6. Arsenii Rudich, Ádám Darvas, and Peter Müller. Checking well-formedness of pure-method specifications. In *FM*, 2008.
7. K. Rustan M. Leino and Ronald Middelkoop. Practical reasoning about invocations and implementations of pure methods. In *FASE*, 2009.
8. Bart Jacobs and Frank Piessens. Inspector methods for state abstraction. *Journal of Object Technology*, 6(5), 2007.
9. Ádám Darvas and K. Rustan M. Leino. Practical reasoning about invocations and implementations of pure methods. In *FASE*, 2007.
10. Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. Automatic verification of java programs with dynamic frames. In *FASE*, 2008.
11. Ádám Darvas and Peter Müller. Reasoning about method calls in interface specifications. *2006*, 5(5), Journal of Object Technology.
12. Dino Distefano and Matthew Parkinson. jStar: Towards practical verification for Java. In *OOPSLA*, 2008.
13. Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, 2008.
14. K. Rustan M. Leino and Peter Müller. A basis for verifying multi-threaded programs. In *ESOP*, 2009.
15. Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP*, 2009.
16. Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *FAC*, 2007.