Separation Logic Verification of C Programs with an SMT Solver

Matko Botinčan[1]

University of Zagreb, Croatia

Matthew Parkinson[2]

University of Cambridge, UK

Wolfram Schulte[3]

Microsoft Research, Redmond, USA

Abstract

This paper presents a methodology for automated modular verification of C programs against specifications written in separation logic. The distinguishing features of the approach are representation of the C memory model in separation logic by means of rewrite rules suitable for automation and the careful integration of an SMT solver behind the separation logic prover to guide the proof search.

Keywords: Separation logic, automated verification, automated theorem proving, C programming language.

1 Introduction

Deductive software verification has made great progress in recent years, with a number of automated tools emerging, such as e.g., ESC/Java[16], Spec# [3], Key-C [21], Caduceus [15], Havoc [2] or Verified C Compiler (VCC) [11]. One of the most interesting, but also the hardest, properties one wants to verify with such tools are those of heap allocated data. Specifying properties about the objects in the heap poses a major challenge.

Separation logic [17,24] is a promising new approach tailored for specifying and verifying properties of heap allocated data. It supports local reasoning by allowing small specifications that speak only about memory footprints. That is, in order

[1] Email: matko.botincan@math.hr

[2] Email: Matthew.Parkinson@cl.cam.ac.uk

[3] Email: schulte@microsoft.com

to specify how a piece of code works one only needs to deal with the part of the memory that the code actually accesses. This often leads to simpler specifications and proofs than one would have in other formalisms.

This paper presents a methodology for automated modular verification of C programs against specifications written in separation logic. Its distinguishing features are representation of the C memory model in separation logic by means of rewrite rules suitable for automation. We also show how to integrate the use of a separation logic prover with an SMT solver. The separation logic prover reduces validity of separation logic formulae to validity of formulae of classical logic, which can then be checked using an SMT solver. The SMT solver can also be called from the separation logic prover for computing a congruence closure and guiding proof search. This is needed, for instance, in case when the prover tries to establish equality between expressions that denote memory addresses involving arithmetic operations arising from the memory model representation. The SMT solver supports such reasoning within built-in background theories or those axiomatized by the user.

The presented methodology has been implemented in a prototype tool that uses the Z3 SMT solver [12] and a modified version of the separation logic prover from the jStar tool [14]. The tool has shown to be able to automatically verify a handful of small annotated example C programs involving dynamically allocated structures, one of which is described in the paper.

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

- A simple separation logic based representation of the C memory model and, in particular, composite C types (structures, unions and arrays), defined in terms of rewrite and sequent rules;
- An overview of CoreC*, a small and compact subset of the C language with contracts tailored for simple symbolic execution;
- The integration of an SMT solver with a separation logic prover.

1.1 Related Work

A number of tools such as Havoc [2], VCC [11] or Escher's C compiler [10] addressing automated deductive verification of low-level C programs have emerged in recent years. Many foundational issues regarding C memory model have been described in greater detail: Leroy and Blazy [19] describe a formal verification of a memory model for low-level imperative languages such as C; Moy [20] shows how unions and casts can be encoded as structure subtyping; and Norrish [23] deals with formalizing most aspects of C semantics in HOL.

Separation logic has already evidenced promising results in verifying high-level procedural programs [4,5], and object-oriented programs [7,14]. Design of our verification tool has been influenced by [14], while the programming language constructs for specifying contracts are adopted from Spec# [3] and VCC [11]. Conceptually similar work to ours is [1] on verifying Cminor programs using separation logic. Also related to our work is research in shape analysis based on separation logic [13].

A line of research about L4 kernel verification employs a separation logic based formalism capturing low-level features of the C memory model [28,29,30]. This approach deals with intricacies of structured C types in detail on a foundational

level, however, it is not as directly applicable for fully automated verification as our approach. Pelaska and Löding [25] developed a technique for abstract interpretation of C/C++ programs that has been applied for verification of embedded systems in the fields of avionics, railways and automotive control. Their approach is similar to ours with respect to the memory model treatment and the symbolic execution.

Use of SMT solvers for automated verification of heap properties is also not new, see e.g., [6,18] or [26]. A closely related work with respect to using an SMT solver for checking validity of separation logic formulae is [22]: the authors use an SMT solver for checking validity of pure formulae (they do not, however, use it for congruence closure computation).

The rest of this paper is structured as follows. We begin with motivating our approach on an example in Section 2. Section 3 explains how we represent the C memory model by use of rewrite rules. We give the architecture of our prototype tool in Section 4, and then describe its key components: the symbolic interpreter in Section 5, and the theorem prover in Section 6. The final Section 7 gives concluding remarks and discusses future work.

2 A Motivating Example

In this section, we motivate the methodology presented in the paper. We first give a brief introduction to the annotation language, and then explain how to use it to specify a queue data structure that is automatically verifiable with our tool.

2.1 A brief introduction to the syntax

Let S be a structure containing two **int** variables a and b:

```
struct S { int a; int b; };
```

and let f be a function that updates fields a and b of an object of type S, say p, to 42 and 28, respectively. The specification of f should require that p refers to a properly allocated object of type S and ensure that its fields a and b are indeed properly updated upon return from the call. In our syntax, the code for f together with the annotations for preconditions and postconditions would look as follows:

```
void f (struct S* p)
 logical (int x; int y;)
requires (p`->`S{x,y})
ensures (p`->`S{42,28})
{
  p->a = 42;
  p->b = 28;
}
```

The operator `->` in the concrete syntax represents the "points to" predicate symbol from separation logic with a meaning that p`->`S{x,y} holds iff p points to an object of type S such that its fields a and b have **int** values x and y, respectively. If we are not interested in specifying explicit values of object's fields, variables passed as parameters to object's constructor should be existentially quantified by declaring them as **logical**.

2.2 Specification of the queue

Let us now move on to a more interesting example. The goal is to specify, implement and verify operations of a queue data structure. Due to page limits, we show the final annotation for one operation only, but we explain all the ingredients.

The queue is implemented as a singly-linked list of elements of type Node:

```
typedef struct node {
  struct node* next;
  void* data;
} Node, *PNode;
```

Using the operator '−>', we can easily specify the contract of a function allocating a new queue node (variable **result** refers to the value returned from the function):

```
struct node* alloc_node()
 logical (void* d; PNode n;)
ensures (result‘−>‘Node{n,d});
```

The queue is assumed to always have a sentinel node allocated in the front. The first and the last node of the queue's linked list is accessible via the queue's first and last fields:

```
typedef struct queue {
  PNode first;
  PNode last;
} Queue;
```

For specification purposes, the queue can be seen as an abstract sequence of values stored in the queue. It is convenient to specify validity properties of the queue with respect to such abstract sequence. For instance, if first and last point to the same node then the sequence of values in the queue is empty. If first and last are different, then the queue nodes form a linked list with its values comprising a sequence of values stored in the queue. In the following, let predicate seq_empty() stand for the empty sequence; seq_cons(y,ys) for the sequence having y inserted before ys; and seq_append(ys,zs) for the sequence with ys being appended with zs. In our tool, such predicates for dealing with mathematical sequences are axiomatized as a background theory within the SMT theorem prover.

The validity properties of the queue are formally captured via the valid_queue predicate:

```
spec int valid_queue(Queue* q, sequence xs)
 logical (PNode f, l, n; void *d;)
ensures (result == q‘−>‘Queue{f,l} ‘*‘ f‘−>‘Node{n,d} &&
    ((f==l) ==> (xs==seq_empty())) && ((f!=l) ==> list(n, l, xs)));
```

Here the operator '*' represents the separating conjunction from separation logic which denotes that left and right conjuncts access disjoint parts of the heap. The meaning of list (h,t,xs) is that nodes between h and t form a linked list with the sequence of values xs:

```
spec int  list (PNode h, PNode t, sequence xs)
 logical (PNode n; void* d; sequence ys)
ensures (result == h‘−>‘Node{n,d} && ((h==t) ==> (xs==seq_cons(d, seq_empty()))) &&
    ((h!=t) ==> (xs==seq_cons(d, ys) && list(n, t, ys))));
```

Having everything prepared, the fully annotated enqueue operation of the queue looks as follows:

```
void enqueue(struct queue* q, void* d)
 logical (sequence xs)
requires (valid_queue(q, xs))
ensures (valid_queue(q, seq_append(xs, seq_cons(d, seq_empty()))))
{
  struct node* n = alloc_node();
  n->data = d;
  q->last->next = n;
  q->last = q->last->next;
}
```

3 The C Memory Model

This section describes the C memory model that we use in our approach. Here the memory model refers to the formal description of the memory layout of different C objects (in particular, primitive types, structures, unions and arrays). The key feature is determining how to provide enough aliasing and anti-aliasing information about objects in memory to make operations based on pointer arithmetic feasible.

The main difficulty with the C memory model defined by C standards in wide use (such as ISO/IEC 9899:1990) is that it is very permissive. The memory is seen as a sequence of bytes (actually, a collection of byte sequences), and interpretation of a chunk of memory depends on the type of the pointer used for accessing it.

Type-safe languages such as Java or C# see memory as a collection of objects, and the aliasing between two objects can happen only if two pointers (of the same type) are pointing to the same object. This approach is not sound for C since in C objects can overlap almost arbitrarily, however, it can be extended to become sound and complete with respect to the byte-level C memory model (see [9] for a detailed discussion). The key ingredient is having a way to represent when an object is a structural descendent of another, and this is the step we follow as well.

The C memory model in our approach represents memory locations as abstract symbolic values. We provide necessary aliasing and anti-aliasing information by unfolding denotations of locations with rewrite rules. The rewrite rules take into account the memory layout of objects (i.e., the offsets of object fields and array components). This is motivated by the fact that if two denotations refer to the same location, rewrite rules should eventually yield equivalent representations of them.

3.1 Structures

When a memory location is accessed via a sequence of object's fields (as in $\&(*\mathsf{p}).\mathsf{a}$), the rewrite rules provide a way to get rid of field accesses in exchange for adding corresponding field offsets to the memory location of the object. For instance, $\&(*\mathsf{p}).\mathsf{a}$ refers to the same memory location as $\mathsf{p} + \mathsf{offset}(\mathsf{S}.\mathsf{a})$, which equals to p since the field a in structure S is at offset 0. Here the function offset gives a field's offset which is calculated by summing up sizes of the preceding fields together with all intermediate padding (if any). The function Loc formalizes this idea:

$$\begin{aligned} \mathsf{Loc}(x, [\,]) &= x, \\ \mathsf{Loc}(x, [f_1, \ldots, f_n]) &= \mathsf{Loc}(x + \mathsf{offset}(f_1), [f_2, \ldots, f_n]). \end{aligned}$$

Loc is used whenever we want to deal with memory locations that are accessed via a sequence of object's fields.

To see how this works in practice, let us consider the case of a nested structure:

```
struct T { struct S s; int c; };
```

and let t be a pointer to an object of type T, and let S.a, S.b, T.s and T.c be the qualified field names of structures S and T, respectively. Then access to the memory location &(∗t).s.a is represented as Loc(t, [T.s, S.a]) which unfolds to t, establishing in this way aliasing between &(∗t).s.a and t. Similarly, &(∗t).c refers to a memory location different than t since Loc(t, [T.c]) equals to t + offset(T.c).

Loc participates in rewrite rules for the typed version of the "points to" relation which are unfolded whenever one is accessing an object's field (i.e., a memory location accessible by a field offset from the object). For the structure T, the definition would look as follows:

$$\mathsf{t} \mapsto_{\mathsf{T}} \mathsf{T}\{\mathsf{S}\{\mathsf{x},\mathsf{y}\},\mathsf{z}\} \;\Leftrightarrow\; \mathsf{Loc}(\mathsf{t},[\mathsf{T.s}]) \mapsto_{\mathsf{T}} \mathsf{S}\{\mathsf{x},\mathsf{y}\} * \mathsf{Loc}(\mathsf{t},[\mathsf{T.c}]) \mapsto_{\mathsf{T}} \mathsf{z}.$$

In general, if T is a structure with fields $f_1, \ldots, f_n$ of types $T_1, \ldots, T_n$ with values $x_1, \ldots, x_n$, then we have the following definition for an object p of type T:

$$p \mapsto_T T\{x_1, x_2, \ldots, x_n\} \;\Leftrightarrow\; \mathsf{Loc}(p,[T.f_1]) \mapsto_{T_1} x_1 * \ldots * \mathsf{Loc}(p,[T.f_n]) \mapsto_{T_n} x_n.$$

Note that we do not mention padding in this definition. We aim to apply our approach to the environment where padding fields are required to be introduced explicitly (and where a compiler issues a warning if it would introduce padding fields itself). Should account of padding be presented in the model, one would employ the following definition:

$$\begin{aligned} p \mapsto_T T\{x_1, x_2, \ldots, x_n\} \Leftrightarrow\, & \mathsf{Loc}(p,[T.f_1]) \mapsto_{T_1} x_1 * \mathsf{Pad}(p,[T.f_1]) * \ldots * \\ & \mathsf{Pad}(p,[T.f_{n-1}]) * \mathsf{Loc}(p,[T.f_n]) \mapsto_{T_n} x_n. \end{aligned}$$

Here the padding predicate Pad stands for the padding introduced for a particular field. It is not meant to be unrolled further, but is just passed around in the prover during the proof search.

3.2 Unions

The approach we used for structures can be applied for unions as well, the difference being the way field offsets are calculated. Each field of a union begins at the same memory location, thus the function offset yields the same value for all fields belonging to the same union. For instance, consider a union U:

```
union U { struct S s; int c; double d; };
```

and let u be an object of type U. Then Loc(u, [U.s]), Loc(u, [U.c]) and Loc(u, [U.d]) are all equal to u.

Unions in C are untagged, thus one can choose to read a union's field that was not most recently assigned to. Should types of the fields match, the field read access is to be considered safe. To see how this is supported within our model, consider again the object u. If u‘−>‘S{x,y}, then one can access (∗u).s (and thus

(∗u).s.a and (∗u).s.b), as well as (∗u).c (since it has the same offset and the type as U.s.a). Due to a different type, however, (∗u).d can not be accessed in this case. If u‘−>‘**int**{x}, then (∗u).c and (∗u).s.a are accessible, but (∗u).s.b and (∗u).d are not. Yet, if u‘−>‘**double**{x}, then one can access only (∗u).d. In our tool, if an inaccessible field gets encountered, the associated assertion cannot be verified and the inaccessibility is reported as the reason back to the user.

3.3 Bitfields

During the preprocessing phase, we compile bitfields away into unsigned integer(s) (respecting the order of appearance in the declaration) and manipulate them at field accesses explicitly with bitvector arithmetic. A related approach has been described in [8], where the correctness of such compilation is machine-checked with a theorem prover.

3.4 Arrays

An array of T's in C is a pointer to a contiguous block of memory encompassing a sequence of elements of type T numbered from 0. The memory location of the i-th element of the array equals to the memory location of the 0-th increased by i times the size of the array element type, as given by Ind_T:

$$\mathsf{Ind}_T(p, i) := p + i * \mathsf{sizeof}(T).$$

Arrays can be split in the following way:

$$\begin{aligned} p \mapsto_{T[n]} \{x_0, \ldots, x_{n-1}\} \Leftrightarrow p \mapsto_{T[l]} \{x_0, \ldots, x_{l-1}\} * \\ \mathsf{Ind}_T(p, l) \mapsto_{T[h-l]} \{x_l, \ldots, x_{h-1}\} * \\ \mathsf{Ind}_T(p, h) \mapsto_{T[n-h]} \{x_h, \ldots, x_{n-1}\}. \end{aligned}$$

Here $T[k]$ denotes an array of T's of length k, and $0 \leq l \leq h < n$. We equate the empty array with the empty predicate:

$$p \mapsto_{T[0]} \{\} \Leftrightarrow \mathsf{empty}$$

In the later part of the paper, we explain how we use guards to guide unfolding of the rules.

4 The Tool Architecture

The architecture of our prototype tool for automated separation logic verification of C programs is depicted in Figure 1. The tool takes a C program annotated with pre/post-conditions, assumptions and assertions as its input and transforms it into an equivalent program that uses only a compact subset of the C language, named CoreC*. This program representation is then sent to the symbolic interpreter.

The symbolic interpreter symbolically executes the control flow graph of each function in the CoreC*program. The state of the heap gets updated according to the symbolic execution rules. Since states of the heap are represented with separation logic formulae, a support from a theorem prover is needed.

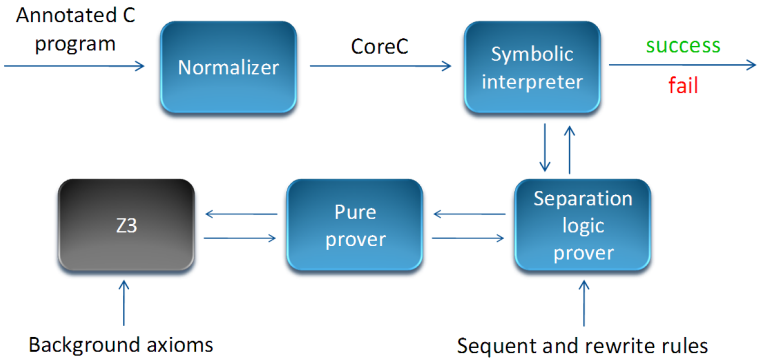

Fig. 1. The Tool Architecture

The theorem prover comprises a separation logic prover and an SMT solver. The separation logic prover is a modified version of the prover used in the jStar tool [14] and is used for frame inference, deciding spatial implication and rewriting modulo user provided rewrite rules. The key difference with respect to [14] is use of an SMT solver for deciding pure implication and congruence closure computation.

5 Symbolic Interpreter

In this section, we describe the symbolic interpreter. We first deal with the abstract syntax of its input language, CoreC*, and then with the symbolic execution rules defining the effects of CoreC* statements on symbolic states.

*5.1 CoreC**

The design of language CoreC* has been inspired with the design of CoreC, a small subset of the C language to which any C program can be transformed to, as described in [31]. The goal of CoreC* is to enrich CoreC with design by contract annotations, and reorganize its abstract syntax to make symbolic execution of C programs easy with respect to the separation logic based heap representation. A simplified abstract syntax of CoreC* function declarations, statements and expressions is shown on Figure 2. Here Var stands for program variables, $Type$ for types (including user defined ones), $Field$ for fields of composite types, and Fun for function identifiers.

A CoreC* program that is handed as an input to the symbolic interpreter consists of a list of top level function declarations. Before the symbolic execution begins, all type declarations are processed and necessary rules for the separation logic prover are created. The symbolic interpreter goes over each function declaration in the CoreC* program and runs symbolic execution rules over its control flow graph.

5.2 Symbolic execution

The goal of the symbolic execution is to prove correctness of each function with respect to its contract, i.e., the interpreter assumes preconditions on function entry, and asserts the function's postcondition on exit. On function call, the caller is required to establish the called function's precondition, and then assumes that the called function ensured its postcondition. This makes the approach modular.

$$
\begin{array}{rcl}
FunDecl & ::= & Fun(Var^*) \\
& & (\textbf{requires}\ Expr)^* \\
& & (\textbf{ensures}\ Expr)^* \\
& & Stmt^? \\
Expr & ::= & Constant \\
& | & Var \\
& | & Expr\ \textbf{op}\ Expr \\
& | & \exists Var\,.\,Expr \\
& | & Fun\,(Expr^*) \\
& | & \textbf{cast}\,(Type, Expr) \\
& | & (*Var).Field^* \\
& | & \&(*Var).Field^* \\
& | & Expr[Expr] \\
& | & \&Expr[Expr]
\end{array}
\qquad
\begin{array}{rcl}
Stmt & ::= & VarDecl \\
& | & Var = Expr \\
& | & (*Var).Field^* = Expr \\
& | & Expr[Expr] = Expr \\
& | & Var = Fun\,(Expr^*) \\
& | & Fun\,(Expr^*) \\
& | & \textbf{if}\,(Expr)\ \textbf{then goto}\ Label \\
& & \textbf{else goto}\ Label \\
& | & \textbf{label}\ Label \\
& | & \textbf{goto}\ Label \\
& | & \textbf{return}\ Expr^? \\
& | & \textbf{assert}\ Expr \\
& | & \textbf{assume}\ Expr \\
& | & \textbf{block}\ Stmt^+
\end{array}
$$

Fig. 2. Abstract syntax of CoreC*function declarations, statements and expressions

Symbolic execution is formally defined on a transition system with states of the form (pc, H), where pc is the index of the statement currently being executed, and H is the symbolic representation of the heap. Each statement increments pc by 1 (except for **if** and **goto**) and updates the state of the heap.

States of the heap are symbolically represented in terms of separation logic formulae. The basic ingredient of separation logic used here is the separating conjunction * which allows specifying properties about disjoint portions of the heap. Namely, the formula $H_1 * H_2$ asserts that the heap can be split into two parts, one of them being described only by H_1, and the other only by H_2.

The state of the heap gets updated according to the symbolic execution rules following the symbolic operational semantics. Some of the rules are nondeterministic since when splitting the heap with respect to H, one can find more than one remaining heap that conjoined with H gives back the original heap. Figure 3 shows the most interesting symbolic execution rules: (1) assignment; (2) field mutation; (3) field look-up; (4) array element mutation; (5) array element look-up. Each variable with a hat is assumed implicitly existentially quantified.

6 Theorem Prover

The key parts of the symbolic execution require deciding separation logic entailment, i.e., checking implication or performing frame inference. This is where the separation logic prover comes in. It performs judgements and rewrites terms modulo its internal and user-provided sets of rules until it completes the proof or gets stuck with a formula it does not know how to reason about further. We have extended the separation logic prover so that it can call an SMT solver on such events.

$$\frac{}{H \ \vdash \ x = E \rightsquigarrow H[\hat{x}/x] \ \wedge \ x = E[\hat{x}/x]} \ (1)$$

$$\frac{H \Rightarrow (\mathsf{Loc}(x, [f_1, \ldots, f_n]) \mapsto_T E_1) * H'}{H \ \vdash \ (*x).f_1 \ldots f_n = E_2 \rightsquigarrow (\mathsf{Loc}(x, [f_1, \ldots, f_n]) \mapsto_T E_2) * H'} \ (2)$$

$$\frac{H \Rightarrow (\mathsf{Loc}(E, [f_1, \ldots, f_n]) \mapsto_T F) * H'}{H \ \vdash \ x = (*E).f_1 \ldots f_n \rightsquigarrow ((\mathsf{Loc}(E, [f_1, \ldots, f_n]) \mapsto_T F) * H')[\hat{x}/x] \ \wedge \ x = F[\hat{x}/x]} \ (3)$$

$$\frac{H \Rightarrow (\mathsf{Ind}_T(x, i) \mapsto_T E_1) * H'}{H \ \vdash \ x[i] = E_2 \rightsquigarrow (\mathsf{Ind}_T(x, i) \mapsto_T E_2) * H'} \ (4)$$

$$\frac{H \Rightarrow (\mathsf{Ind}_T(E, i) \mapsto_T F) * H'}{H \ \vdash \ x = E[i] \rightsquigarrow ((\mathsf{Ind}_T(E, i) \mapsto_T F) * H')[\hat{x}/x] \ \wedge \ x = F[\hat{x}/x]} \ (5)$$

Fig. 3. Symbolic execution rules

6.1 Separation logic prover

The separation logic prover uses a restricted form of separation logic formulae of the form $\exists \hat{x}_1 \ldots \hat{x}_n \,.\, \Pi \wedge \Sigma$. Here Π is a pure formula not involving heap allocated objects, Σ is a spatial formula speaking about the heap, and $\hat{x}_1$, $\ldots$, $\hat{x}_n$ are existential variables occurring in $\Pi \wedge \Sigma$ (for brevity, the quantifier over existential variables is left implicit further in the text).

The set of expressions E used in separation logic formulae comprise program variables (denoted by x, y, $\ldots$), existential variables (denoted by $\hat{x}$, $\hat{y}$, $\ldots$), constants and function applications. Pure and spatial formulae are built according to the following grammar:

$$\Pi ::= \mathsf{true} \mid \mathsf{false} \mid E = E \mid E \neq E \mid p(\overline{E}) \mid \Pi \wedge \Pi$$
$$\Sigma ::= \mathsf{empty} \mid s(\overline{E}) \mid \Sigma * \Sigma.$$

The prover allows definition of arbitrary pure predicates p and spatial predicates s. The predicate empty asserts that the heap is empty. The meaning of the separating conjunction is that the formula $S_1 * S_2$ holds in a heap iff the heap can be split into two disjoint parts in which S_1 and S_2 hold, respectively. A more detailed introduction to separation logic can be found for example in [27].

The design of the separation logic prover is based on the entailment checker in the Smallfoot tool [5]. It works on sequents of the form

$$\Sigma_f \mid \Pi_1 \mid \Sigma_1 \ \vdash \ \Pi_2 \mid \Sigma_2$$

where Π_1 and Π_2 are pure formulae, and Σ_1 and Σ_2 are spatial formulae. Here $\Pi_1 \mid \Sigma_1$ is the assumed formula, $\Pi_2 \mid \Sigma_2$ the goal formula, and Σ_f the subtracted formula. The underlying semantics of such judgements is $\Pi_1 \wedge (\Sigma_1 * \Sigma_f)$ implies $\Pi_2 \wedge (\Sigma_1 * \Sigma_f)$.

The prover is built in such a way that it can be extended with arbitrary judgment and term rewriting rules. For instance, our typed version of the "points to" relation

is defined as a ternary spatial predicate $\mapsto$ taking as parameters a location, a type, and a value. This is how its basic match rule looks like:

$$\frac{p \mapsto_T x \mid \Pi_1 \mid \Sigma_1 \ \vdash \ \Pi_2 \wedge x = y \mid \Sigma_2}{\Sigma_f \mid \Pi_1 \mid \Sigma_1 * p \mapsto_T x \ \vdash \ \Pi_2 \mid \Sigma_2 * p \mapsto_T y}$$

To simplify the presentation, henceforth, we only present the parts of the rule that change:

$$\frac{p \mapsto_T x \mid \mathsf{true} \mid \mathsf{empty} \ \vdash \ x = y \mid \mathsf{empty}}{\mathsf{empty} \mid \mathsf{true} \mid p \mapsto_T x \ \vdash \ \mathsf{true} \mid p \mapsto_T y}$$

The C memory model is defined in terms of rewrite and sequent rules as described in Section 3. For instance, after one defines rewrite rules for accessing fields in the structure via the Loc function, the unfold and fold rules for a structure T with fields $f_1, \ldots, f_n$ of types $T_1, \ldots, T_n$ are in the prover given as:

$$\frac{\mathsf{empty} \mid x = T\{\hat{x}_1, \ldots, \hat{x}_n\} \mid \circledast_{i=1}^{n} \mathsf{Loc}(p, [T.f_i]) \mapsto_{T_i} \hat{x}_i \ \vdash \ \mathsf{true} \mid \mathsf{Loc}(p, fs) \mapsto_{T'} v'}{\mathsf{empty} \mid \mathsf{true} \mid p \mapsto_T x \ \vdash \ \mathsf{true} \mid \mathsf{Loc}(p, fs) \mapsto_{T'} v'}$$

$$\frac{\mathsf{empty} \mid \mathsf{true} \mid \mathsf{Loc}(p, fs) \mapsto_{T'} v' \ \vdash \ x = T\{\hat{x}_1, \ldots, \hat{x}_n\} \mid \circledast_{i=1}^{n} \mathsf{Loc}(p, [T.f_i]) \mapsto_{T_i} \hat{x}_i}{\mathsf{empty} \mid \mathsf{true} \mid \mathsf{Loc}(p, fs) \mapsto_{T'} v' \ \vdash \ \mathsf{true} \mid p \mapsto_T x}$$

The rule uses $\mathsf{Loc}(p, fs) \mapsto_{T'} v'$ to only allow unfolding and folding a data structure when a subcomponent is involved in the proof. Here fs is an arbitrary list of fields.

During the proof search, the prover applies internal and user-defined rules to sequents to generate new ones. The prover unfolds definitions only when they are actually needed and that does so to a level required for the proof. In order to prove implication or perform frame inference, it searches for a particular kind of sequent, namely:

- $\Sigma_f \mid \Pi_1 \mid \mathsf{empty} \ \vdash \ \Pi_2 \mid \mathsf{empty}$ for checking implication; and
- $\Sigma_f \mid \Pi_1 \mid F \ \vdash \ \Pi_2 \mid \mathsf{empty}$ for frame inference (here F is the frame).

If a sequent of this form is reached, then all that is left to prove is $\Pi_1 \Rightarrow \Pi_2$, and this is what gets sent to the SMT solver. If the SMT solver succeeds in checking the implication, then the proof search is complete. Otherwise, the separation logic prover tries any backtracking points that have remained, or admits defeat.

The separation logic prover can, however, also get stuck with determining equality between terms that occur in the spatial part. For instance, consider the situation when it has to prove that

$$\Sigma_f \mid \Pi_1 \mid p + 2 * 4 \mapsto_{\mathsf{int}} x \ \vdash \ \Sigma_f \mid \Pi_2 \mid 8 + p \mapsto_{\mathsf{int}} x.$$

If no rules for arithmetics are provided, the prover cannot know whether the terms $p + 2 * 4$ and $8 + p$ do match or not.

A possible solution for this problem is to employ the SMT solver for determining the equality between pairs of terms. More precisely, for every pair of terms t_1, t_2 in the sequent the SMT solver can be asked if $\Pi_1 \Rightarrow t_1 = t_2$ holds, and if so, then

add $t_1 = t_2$ to Π_1. In our example, one should ask if $\Pi_1 \Rightarrow p + 2 * 4 = 8 + p$, and hopefully obtain the wanted equality. We describe in more details this approach in the final part of this section.

Having an SMT solver at hand also allows us to add guards to rules involving complex predicates. That is, during the proof search, the prover can match Π_1, Σ_1, Π_2 and Σ_2 in $\Sigma_f \mid \Pi_1 \mid \Sigma_1 \vdash \Pi_2 \mid \Sigma_2$ not just syntactically, but can also employ the power of the SMT solver. We take this approach when dealing with arrays:

$$\frac{\mathsf{empty} \mid \mathsf{true} \mid \mathsf{Ind}_T(p,l) \mapsto_T \hat{x}_l * \ldots * \mathsf{Ind}_T(p,h-1) \mapsto_T \hat{x}_{h-1} \vdash \mathsf{true} \mid q \mapsto_{T'} \hat{x'}}{\mathsf{empty} \mid \mathsf{true} \mid \mathsf{Ind}_T(p,l) \mapsto_{T[h-l]} \{\hat{x}_l, \ldots, \hat{x}_{h-1}\} * \mathsf{guard} \vdash \mathsf{true} \mid q \mapsto_{T'} \hat{x'}}$$

$$\frac{\mathsf{empty} \mid \mathsf{true} \mid q \mapsto_{T'} \hat{x'} \vdash \mathsf{true} \mid \mathsf{Ind}_T(p,l) \mapsto_T \hat{x}_l * \ldots * \mathsf{Ind}_T(p,h-1) \mapsto_T \hat{x}_{h-1}}{\mathsf{empty} \mid \mathsf{true} \mid q \mapsto_{T'} \hat{x'} * \mathsf{guard} \vdash \mathsf{true} \mid \mathsf{Ind}_T(p,l) \mapsto_{T[h-l]} \{\hat{x}_l, \ldots, \hat{x}_{h-1}\}}$$

where guard equals

$$\mathsf{Ind}_T(p,l) <= q \wedge q <= \mathsf{Ind}_T(p,h-1) \wedge \mathsf{sizeof}(T') <= \mathsf{sizeof}(T[h-l]).$$

The guard predicate is sent to the SMT solver and helps guide the unfolding. Conjuncts together with the right hand side of the sequents allow rule to be fired for any object smaller than the array itself (e.g., even for subcomponents of a structure), while preventing unfolding the array when on the right there is an object containing the array.

6.2 Pure prover

The pure prover serves as a mediator between the separation logic prover and the SMT solver. It does all necessary translation of pure formulae into SMT solver formulae, deals with conversion of types, and communicates back the results obtained from the SMT solver. One can also attach different SMT solvers to the separation logic prover via the same interface.

The pure prover can be asked for one of the two tasks: to check implication between pure formulae or to determine equalities among terms in a given set of terms. The second task refers to what is commonly known as the congruence closure computation.

6.2.1 Checking implication.

Let Π_1 and Π_2 be pure formulae, where $\hat{x}_1, \ldots, \hat{x}_s$ and $\hat{y}_1, \ldots, \hat{y}_t$ are existential variables occurring in Π_1 and Π_2, respectively, and $v_1, \ldots, v_r$ program variables occurring in both Π_1 and Π_2. For checking implication, the SMT solver is asked to check satisfiability of the following formula:

$$\neg \forall v_1 \ldots v_r (\exists \hat{x}_1 \ldots \hat{x}_s \Pi_1 \rightarrow \exists \hat{y}_1 \ldots \hat{y}_t \Pi_2).$$

6.2.2 Congruence closure computation.

We present two algorithms used in the pure prover to compute congruence closure with an SMT solver. The first one computes congruence closure for uninterpreted

functions with Z3, while the second one is a general (and less efficient) approach for dealing with both uninterpreted and interpreted functions in any SMT solver.

The way we compute the congruence closure for uninterpreted functions relies on specific features of Z3 [12]. Once asked to check for satisfiability, Z3 can produce a model as a part of the output. The model assigns values to the constants in the input and generates partial function graphs for predicates and function symbols. The implementation of the congruence closure computation for uninterpreted functions in Z3 happens to compute a coarsest partition satisfying asserted equalities, so terms found equal in the generated model are guaranteed to be implied equal.

Algorithm 1 describes the process of computing the congruence closure for uninterpreted functions with Z3. The input to the algorithm is a set of terms $\{t_1, \ldots, t_n\}$ and a pure formula Π representing assumptions about terms (i.e., the knowledge about which congruence exists among terms). Only constants and uninterpreted functions can appear in terms and assumptions. The output of the algorithm is the congruence closure of the given set of terms, i.e., a partition $\{R_1, \ldots, R_k\}$ of resulting equivalence classes.

Algorithm 1 Congruence closure for uninterpreted functions with Z3

Input: Set of terms $\{t_1, \ldots, t_n\}$ and pure formula Π.
Output: Partition $\{R_1, \ldots, R_k\}$ of equivalence classes of terms.

procedure CongruenceClosureUF($\{t_1, \ldots, t_n\}, \Pi$)
  Assert constraints Π;
  Generate model $\mathcal{M}$;
  $$\mathrm{eq}(\mathcal{M}, t_i, t_j) := \begin{cases} \textbf{false}, \text{ if } [\![t_i]\!]_{\mathcal{M}} = \text{unknown or } [\![t_j]\!]_{\mathcal{M}} = \text{unknown}, \\ \textbf{true}, \text{ if } [\![t_i]\!]_{\mathcal{M}} = [\![t_j]\!]_{\mathcal{M}}. \end{cases};$$
  Define congruence relation $\sim$ with $t_i \sim t_j$ iff $\mathrm{eq}(\mathcal{M}, t_i, t_j) = \textbf{true}$;
  $\{R_1, \ldots, R_k\}$ = equivalence classes of $\sim$;
  return $\{R_1, \ldots, R_k\}$;
end procedure

The implementation of the congruence closure computation for uninterpreted functions in Z3 computes a coarsest partition satisfying asserted equalities. The terms that are not interpreted within the generated model (i.e., for which the value of $[\![\cdot]\!]_{\mathcal{M}}$ is unknown) are placed in singleton partitions. This preserves the minimality of the returned congruence relation, which ensures that returned is indeed the congruence closure of the congruence defined by Π over the set of terms $\{t_1, \ldots, t_n\}$.

In order to compute congruence closure for interpreted functions we cannot just inspect the model and check whether given terms are equal. A possible approach is shown in Algorithm 2. It does not rely on a model generation capability, nor specifics of SMT solver implementation, so it can be employed with any SMT solver. The algorithm assumes that the SMT solver supports Push/Pop backtracking mechanism (this mechanism provides that constraints asserted after a Push gets removed after a Pop). Should this not be the case, one can run each iteration of the loop separately (with constraints each time being asserted).

Algorithm 2 Congruence closure computation

Input: Set of terms $\{t_1, \ldots, t_n\}$ and pure formula Π representing assumptions.
Output: Partition $\{R_1, \ldots, R_k\}$ of equivalence classes of terms.

```
procedure CongruenceClosure({t_1, ..., t_n}, Π)
    Assert constraints Π;
    S := {1, ..., n};
    while S ≠ ∅ do
        Remove i from S;
        for all j ∈ S do
            Push context;
            Assert constraint t_i ≠ t_j;
            Check for satisfiability;
            if unsatisfiable then remove j from S and t_i ~ t_j else t_i ≁ t_j;
            Pop context;
        end for
    end while
    {R_1, ..., R_k} = equivalence classes of ~;
    return {R_1, ..., R_k};
end procedure
```

7 Conclusions and Future Work

This paper described a methodology for automated modular verification of C programs against specifications written in separation logic that we implemented in a prototype tool. The tool is in an early stage of development and has been tested just on a handful of small examples. It performs at a reasonable speed; for instance, the queue example from Section 2 takes about 8 seconds to verify.

We believe that there is a promising potential lying in the tight integration of a separation logic prover and an SMT solver. Larger case studies, however, are needed to explore whether such approach when applied to verification of C programs could be seen as better (at least in some aspects) than other approaches.

An important drawback is lack of termination guarantees. If the tool stops, then it has either proven the program to be correct against the specifications or has found a failing point. However, it may loop forever. In order to ensure termination, it would be sensible to apply abstraction in the spirit of abstract interpretation, as it has been done in the jStar tool [14].

Acknowledgement. We thank Nikolaj Bjørner for providing us with details about Z3 and suggesting us the approach in Algorithm 1, and anonymous reviewers for helpful suggestions and remarks on the earlier version of this paper.

References

[1] A. W. Appel and S. Blazy. Separation logic for small-step cminor. In *Proceedings of TPHOLs 2007*, volume 4732 of *Lecture Notes in Computer Science*, pages 5–21. Springer, 2007.

[2] T. Ball, B. Hackett, S. K. Lahiri, and S. Qadeer. Annotation-based property checking for systems software. Technical Report MSR-TR-2008-82, Microsoft Research, May 2008.

[3] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Proceedings of CASSIS 2004*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.

[4] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *Proceedings of APLAS 2005*, volume 3780 of *LNCS*, pages 52–68. Springer, 2005.

[5] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Proceedings of FMCO 2005*, volume 4111 of *LNCS*, pages 115–137. Springer, 2006.

[6] S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamaric. A reachability predicate for analyzing low-level software. In *Proceedings of TACAS 2007*, volume 4424 of *LNCS*, pages 19–33. Springer, 2007.

[7] W. N. Chin, C. David, H. H. Nguyen, and S. Qin. Enhancing modular OO verification with separation logic. In *Proceedings of POPL 2008*, pages 87–99. ACM, 2008.

[8] D. Cock. Bitfields and tagged unions in C: Verification through automatic generation. In *Proceedings of VERIFY 2008*, volume 372 of *CEUR Workshop Proceedings*, pages 44–55, 2008.

[9] E. Cohen, M. Moskal, W. Schulte, and S. Tobies. A precise yet efficient memory model for C. In *Proceedings of the 4th International Workshop on Systems Software Verification (SSV'09)*, 2009.

[10] David Crocker and Judith Carlton. Verification of C programs using automated reasoning. In *Proceedings of SEFM 2007*, pages 7–14. IEEE Computer Society, 2007.

[11] M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. VCC: Contract-based modular verification of concurrent C. In *ICSE 2009*. ACM, 2009.

[12] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of TACAS 2008*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[13] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *Proceedings of TACAS 2006*, volume 3920 of *LNCS*, pages 287–302. Springer, 2006.

[14] D. Distefano and M. J. Parkinson. jStar: Towards practical verification for java. In *Proceedings of OOPSLA 2008*, pages 213–226. ACM, 2008.

[15] J. C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *Proceedings of CAV 2007*, volume 4590 of *LNCS*, pages 173–177. Springer, 2007.

[16] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proceedings of PLDI 2002*, pages 234–245. ACM, 2002.

[17] S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Proceedings of POPL*, pages 14–26, 2001.

[18] S. K. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In *Proceedings of POPL 2008*, pages 171–182. ACM, 2008.

[19] X. Leroy and S. Blazy. Formal verification of a c-like memory model and its uses for verifying program transformations. *J. Autom. Reasoning*, 41(1):1–31, 2008.

[20] Y. Moy. Union and cast in deductive verification. In *Proceedings of C/C++ Verification Workshop*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[21] O. Mürk, D. Larsson, and R. Hähnle. KeY-C: A tool for verification of C programs. In *Proceedings of CADE 2007*, volume 4603 of *Lecture Notes in Computer Science*, pages 385–390. Springer, 2007.

[22] H. H. Nguyen and W.-N. Chin. Enhancing program verification with lemmas. In *Proceedings of CAV 2008*, volume 5123 of *Lecture Notes in Computer Science*, pages 355–369. Springer, 2008.

[23] M. Norrish. C formalised in HOL. Technical Report UCAM-CL-TR-453, University of Cambridge, December 1998.

[24] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of CSL*, pages 1–19, 2001.

[25] J. Peleska and H. Löding. Symbolic and abstract interpretation for C/C++ programs. In *Proceedings of the 3rd International Workshop on Systems Software Verification (SSV'08)*, 2008.

[26] Z. Rakamaric, R. Bruttomesso, A. J. Hu, and A. Cimatti. Verifying heap-manipulating programs in an SMT framework. In *Proceedings of ATVA 2007*, volume 4762 of *LNCS*, pages 237–252. Springer, 2007.

[27] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of LICS 2002*, pages 55–74. IEEE Computer Society, 2002.

[28] H. Tuch. *Formal Memory Models for Verifying C Systems Code*. PhD thesis, University of New South Wales, Australia, 2008.

[29] H. Tuch. Structured types and separation logic. *Electr. Notes Theor. Comput. Sci.*, 217:41–59, 2008.

[30] H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In *Proceedings of POPL 2007*, pages 97–108. ACM, 2007.

[31] G. Yorsh. Logical characterizations of heap abstractions. Master's thesis, School of Computer Science, Tel-Aviv University, Israel, 2003.