Cascade 2.0

Wei Wang, Clark Barrett, and Thomas Wies

New York University

Abstract. Cascade is a program static analysis tool developed at New York University. Cascade takes as input a program and a control file. The control file specifies one or more assertions to be checked together with restrictions on program behaviors. The tool generates verification conditions for the specified assertions and checks them using an SMT solver which either produces a proof or gives a concrete trace showing how an assertion can fail. Version 2.0 supports the majority of standard C features except for floating point. It can be used to verify both memory safety as well as user-defined assertions. In this paper, we describe the Cascade system including some of its distinguishing features such as its support for different memory models (trading off precision for scalability) and its ability to reason about linked data structures.

1 Introduction

Automatic verification using SMT solvers is an active area of research, with a number of tools emerging, such as ESC/Java [16], Caduceus [15], LLBMC [30], Spec# [1], HAVOC [8], VCC [10], LAV [29], and Frama-C [13]. Increasingly, SMT solvers are used as back-end checkers because of their speed, automation, and ability to model programs and assertions using built-in theory constructs.

Cascade[1] is an open-source tool developed at New York University for automatically reasoning about programs. An initial prototype of the system was described in [24]. This paper describes version 2.0, a from-scratch reimplementation which provides a number of new features, including support for nearly all of C (with the exception of floating point), support for loops and recursion via unrolling, support for loop invariants and deductive reasoning, and a new back-end interface supporting both CVC4 [2] and Z3 [14].[2] It is easy to add additional back-end plugins as long as they support the SMT-LIB input format.

In addition to describing the overall system, this paper focuses on two distinguishing features of Cascade: its support for multiple memory models and its extensibility for specific domains.

The paper is organized as follows. Section 2 gives an overview of the system and its features. Section 3 describes the three memory models supported by Cascade, and reports the results of an empirical evaluation of these models on

[1] Available at `http://cims.nyu.edu/~wwang1109/cascade/index.html`

[2] Another important feature of version 2.0 is that it has a very permissive license and if CVC4 is used, it does not depend on any code with restrictive licenses.

the NECLA suite of static analysis benchmarks [23]. Section 4 describes a case study in extending the system to reason about linked data structures, Section 5 describes related work, and Section 6 concludes.

2 System Design

Cascade is implemented in Java. The overall framework is illustrated in Figure 1. This version of Cascade focuses on C, but the system is designed to be able to accommodate multiple front-end languages. The C front-end converts a C program into an abstract syntax tree using a parser built using the Rats parser generator [17]. The core module takes an abstract syntax tree and a control file as input. The control file specifies one or more paths through the program, assumptions that should be made along the path, and assertions that should be checked along the path. The core module uses symbolic execution over the abstract syntax tree to build verification conditions corresponding to the assertions specified in the control file. Currently, it takes the approach of simple forward execution [3,5,18]. The core module converts paths through the abstract syntax tree into logical formulas.

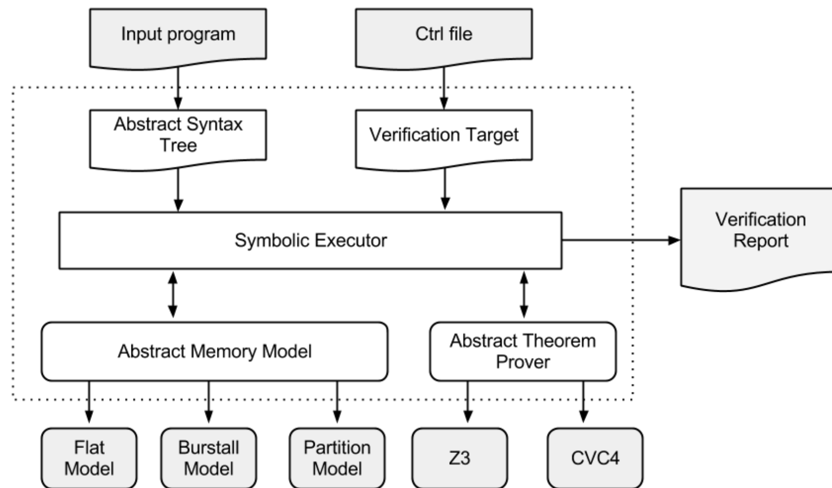

Fig. 1. Cascade framework

2.1 The Control File

Unlike some systems (e.g., [10], [8]), Cascade does not rely on annotated C code. Rather, a separate control file is used to guide the symbolic execution. Control

files use XML and support the constructs detailed below. The rationale for the control file is that we want to be able to use Cascade on large existing code bases without having to modify the code itself.

Basic structure. Every control file begins with a `sourceFile` section that gives the paths to the source files. This is followed by one or more `Run` sections, each defining a constrained (symbolic) run of the program. Each run starts with a single `startPosition` command and ends with a single `endPosition` command that give respectively the start point and end point of the run. If the source code contains branches, Cascade will consider both branches by default (merging them when they meet again). If users wish to execute one branch in particular, they may include one or more `wayPoint` commands in `run`, to indicate the positions that the considered run should pass through. A simple example is shown in Figure 2.

```
int abs(int x) {
  int result;
  if(x>=0)
        result = x;
  else
        result = -x;
  return result;
}
```

```
<controlFile>
  <sourceFile name="abs.c" id="1" />
   <run>
    <startPosition fileId="1" line="1" />
    <wayPoint fileId="1" line="4" />
    <endPosition fileId="1" line="8" />
  </run>
  <run>
    <startPosition fileId="1" line="1" />
    <wayPoint fileId="1" line="6" />
    <endPosition fileId="1" line="8" />
  </run>
</controlFile>
```

Fig. 2. abs.c and abs.ctrl

Function Calls. Cascade supports procedure calls via inlining. Note that Cascade always assigns a unique name for each dynamically encountered variable declaration, so name clashes between caller and callee functions are not an issue. By default, Cascade can perform inlining and parameter passing automatically, as shown in Figure 3 (the body of function `pow2` is inlined at the call sites in `main`). If users wish to specify a particular path in the function, a `wayPoint` command must be used to specify the line on which the function is called. Then, a `function` section can be embedded within the `wayPoint` command which provides an attribute `funcName`, as well as the `wayPoints` of the desired path inside the function. Even if multiple functions are called at the same line of code, this can be handled by nesting multiple function sections under the `wayPoint` command for that line of code. These function sections will constrain the function calls on that line in the order that the function calls appear (from left to right). Figure 4 gives an example.

Loops. By default, loops are eliminated using bounded loop unrolling [3]. A default number of unrolls can be specified on the command line, and a specific

```
int pow2(int x) {
  return x*x;
}

int main() {
  int a, b, result;
  a = 2;
  b = 3;
  result = pow2(a) + pow2(b);
  return result;
}
```

```
<controlFile>
  <sourceFile name="pow2.c" id="1" />
  <run>
    <startPosition fileId="1" line="5" />
    <endPosition fileId="1" line="11" />
  </run>
</controlFile>
```

Fig. 3. pow2.c and pow2.ctrl

```
int abs(int x) {
  int result;
  if(x>=0)
        result = x;
  else
        result = -x;
  return result;
}

int main() {
  int a, result;
  a = -4;
  result = abs(a) - abs(-a);
  return result;
}
```

```
<controlFile>
  <sourceFile name="absext2.c" id="1" />
  <run>
    <startPosition fileId="1" line="10" />
    <wayPoint fileId="1" line="13" >
      <function funcName="abs" funcId="1" >
        <wayPoint fileId="1" line="6" />
      </function>
      <function funcName="abs" funcId="2" >
        <wayPoint fileId="1" line="4" />
      </function>
    </wayPoint>
    <endPosition fileId="1" line="15" />
  </run>
</controlFile>
```

Fig. 4. absext2.c and absext2.ctrl

```
int log2(int num) {
  int result, i;
  result = 0;
  for(i=num; i>1; i=i/2) {
    result++;
  }
  return result;
}

int main() {
  int num, result;
  num = 1024;
  result = log2(num);
  return result;
}
```

```
<controlFile>
  <sourceFile name="log2.c" id="1" />
  <run>
    <startPosition fileId="1" line="10" />
    <wayPoint fileId="1" line="13" >
      <function funcName="log2" >
        <wayPoint fileId="1" line="4" >
          <loop iterTimes="10" />
        </wayPoint>
      </function>
    </wayPoint>
    <endPosition fileId="1" line="15" />
  </run>
</controlFile>
```

Fig. 5. log2.c and log2.ctrl

number of iterations for a particular loop can be specified using a `loop` command, as shown in Figure 5. Alternatively, a loop invariant can be specified using the `invariant` command as shown in Figure 6. If a loop invariant is provided, Cascade will simply check that the loop invariant holds when the loop is entered, and that it is preserved by a single iteration of the loop (for this second check, any variables updated in the loop body are assumed to be unconstrained so that the check is valid for all iterations of the loop). Then, the loop invariant is assumed going forward (this is in contrast to the default behavior which is to symbolically execute the loop a fixed number of times). As with assumptions and assertions (see below), invariants are specified using C expressions. Note that quantified loop invariants are acceptable, and Cascade's ability to solve them is limited only by the quantifier reasoning capabilities of the back-end solver.

```
int main() {
  int sum = 0;

  for (int i = 0; i<=10; i++) {
    sum = sum + i;
  }
  return sum;
}
```

```
<controlFile>
  <sourceFile name="forLoop_test.c" id="1" />
  <run>
    <startPosition fileId="1" line="1" />
    <wayPoint fileId="1" line="4" >
      <loop>
        <invariant><![CDATA[
        sum == (i-1) * i / 2 && i >= 0 && i <= 11
        ]]>
        </invariant>
      </loop>
    </wayPoint>
    <endPosition fileId="1" line="7" />
  </run>
</controlFile>
```

Fig. 6. sum.c and sum.ctrl

Commands. Two commands `cascade_assume` and `cascade_check` are provided, each of which takes a C expression as an argument. `cascade_assume` is used to constrain the set of possible states being considered to those satisfying the argument provided. `cascade_check` generates a verification condition to check that the symbolic execution up to this point satisfies the argument provided. Commands are allowed as part of a `startPosition`, `wayPoint`, or `endPosition` directive (see Figure 7).

As a new feature of Cascade 2.0, commands *can* also be included as annotations in the source code, as shown in Figure 8. To use this feature, the "`--inline-anno`" option must be enabled[3]. Cascade provides a number of extensions that can be embedded within C expressions to enable more expressive reasoning. Some are listed here:

– Logic symbols: $implies(P, Q)$, $forall(v, u, E)$ and $exists(v, u, E)$.

[3] The control file style annotation is designed to keep the source code clean, while the inline style is available for those users who prefer it.

```c
int abs(int x) {
  int result;
  if(x>=0)
        result = x;
  else
        result = -x;
  return result;
}
```

```xml
<controlFile>
  <sourceFile name="absext.c" id="1" />
  <run>
    <startPosition fileId="1" line="1" />
    <endPosition fileId="1" line="7" >
      <command>
        <cascadeFunction> cascade_check
        </cascadeFunction>
        <argument><![CDATA[
        result >= 0
        ]]>
        </argument>
      </command>
    </endPosition>
  </run>
</controlFile>
```

Fig. 7. absext.c and absext.ctrl. The assertion is invalid due to the possibility of signed overflow.

- Memory checks:
  - *valid*(p): denotes that p is guaranteed to point to a memory address within a region allocated by the program.
  - *valid*(p, *size*): denotes that the addresses from $p, ..., p + size - 1$ are valid (in the sense described above).
  - *valid_malloc*(p, *size*): denotes the assumptions that can be made on the pointer p after a `malloc` instruction.
  - *valid_free*(p): denotes that a `free` instruction on pointer p is admissible.

```c
int strlen(const char* str){
  ASSUME(valid_malloc(str,
                    4*sizeof(char)));

  int i=0;
  while(str[i] != '\0')
    ++i;

  ASSERT(forall(j,
         implies(j >= 0 && j <= i,
                 valid(&str[i]))));
  return i;
}
```

```xml
<controlFile>
  <sourceFile name="strlen.c"
              id="1" />
  <run>
    <startPosition fileId="1"
                   line="1" />
    <wayPoint fileId="1" line="7" >
      <loop iterTimes="3" />
    </wayPoint>
    <endPosition fileId="1"
                 line="13" />
  </run>
</controlFile>
```

Fig. 8. strlen.c and strlen.ctrl. Because we specify that the loop should be executed exactly 3 times, no errors are found.

3 Memory Models

One goal of Cascade is to support the analysis of systems software such as device drivers and operating systems code. These programs make heavy use of pointer

manipulation and require a fairly precise memory model. A complementary goal of Cascade is to scale to large programs that are not as pointer-intensive. To achieve these complementary goals, Cascade provides three different memory models, with different trade-offs in terms of precision and scalability: (1) the *flat* model, in which all of memory is modeled as a single array; (2) the *Burstall* model [6] which uses an array for every different structure field; and (3) the *partition* model which divides up memory into several partitions, using a pointer analysis to ensure that variables that may alias end up in the same partition. In this section, we discuss these models in detail, including their semantics, implementation details, advantages and restrictions.

3.1 Flat Memory Model

The flat memory model is essentially the standard conceptual memory model for C: the entire memory is represented as a single flat array M mapping addresses to values[4] (by default, both addresses and values are represented as fixed-width bit-vectors). Memory operations are modeled with the array operations *store* and *select*. Each program variable is modeled as the content of some address in memory. For example, the variable x is associated with a memory address $addr_x$, and all reads from and writes to x are done by accessing M at address $addr_x$. This model can soundly support all type-unsafe operations including union types, pointer arithmetic and pointer casts.

Concretely, we model the memory with two arrays M and $Size$ of types

$$M : BitVec(n) \rightarrow BitVec(m)$$
$$Size : BitVec(n) \rightarrow BitVec(m)$$

The constants m and n can be assigned on the command line via the options "`--mem-cell-size`" and "`--mem-addr-size`". [5]

To model dynamic memory allocation operations such as $x = malloc(size)$, a fresh region variable $region_x$ of type $BitVec(n)$ is created and stored at $addr_x$ of M. To keep track of the size of the allocated region, the auxiliary array variable $Size$ is used to map $region_x$ to $size$. Deallocation, $free(x)$, is modeled by selecting the region variable $M[addr_x]$ corresponding to x, and updating $Size$ to 0 at $M[addr_x]$. In the initial state, the array $Size$ is assumed to map all indices to 0. The following table gives the formal semantics of *malloc* and *free*.

Statement	Interpretation
$x = malloc(size)$	$M' = store(M, addr_x, region_x)$ $Size' = store(Size, region_x, size)$
$free(x)$	$Size' = store(Size, M[addr_x], 0)$

[4] Some tools use separate arrays for the stack and the heap. However, this is not always a sound assumption, so Cascade uses a single mapping to represent both.

[5] Integers are represented as fixed-size bit vectors, and thus integer arithmetic is arithmetic modulo 2^k where k is the number of bits. Cascade also allows the user to select unbounded integers to represent integers in the program. This is activated with the option "`--non-overflow`" .

Note that the value of $store(a, i, v)$ is a new array equivalent to a except at index i where its value is now v [27]. The array M is the symbolic value of memory before the operation and M' is the symbolic value of memory afterwards. Having these definitions, memory checks can be formalized as follows:

$$\begin{aligned}
&valid(p, size) \equiv \\
&\quad \exists region : BitVec(n).\ Size[region] > 0 \implies \\
&\qquad M[addr_p] \geq region \ \wedge M[addr_p] + size \leq region + Size[region]
\end{aligned}$$

The predicate $valid(p, size)$ is inserted as an assertion before each memory access.

Allocation and deallocation have associated guard predicates, *valid_malloc*, respectively, *valid_free*. The predicate *valid_free* is used to detect errors related to deallocation of invalid pointers. It is inserted as an assertion before each free instruction and is defined as follows:

$$valid_free(x) \equiv M[addr_x] = 0 \vee Size[M[addr_x]] > 0$$

The predicate *valid_malloc* is used to ensure that the new region $region_x$ is indeed fresh and does not overlap with previously allocated regions. It is inserted as an assumption after each malloc instruction. Cascade provides two modes for the flat memory model that differ in how they interpret this predicate: an *unordered* and an *ordered* mode.

Unordered mode. The unordered mode can be selected with the command line option "`--sound`". In this mode, *valid_malloc* is interpreted as follows:

$$\begin{aligned}
&valid_malloc(p, size) \equiv \\
&\quad M[addr_p] \neq 0 \implies M[addr_p] > 0 \ \wedge M[addr_p] \leq M[addr_p] + size \ \wedge \\
&\qquad (\forall region : BitVec(n).\ Size[region] > 0 \ \wedge region \neq M[addr_p] \implies \\
&\qquad\quad M[addr_p] + size \leq region \vee region + Size[region] \leq M[addr_p])
\end{aligned}$$

This interpretation accurately reflects the C semantics. However, the size of the allocation guards grow quadratically with the number of allocations encountered during symbolic execution (after instantiating the universal quantifiers with the actual regions). This places a high burden on the back-end SMT solvers.

Ordered mode. To obtain a more efficient SMT encoding, Cascade provides an additional ordered mode, which sacrifices precision for scalability without overly constraining the memory model. In this mode, Cascade assumes that every freshly allocated memory address is larger than the largest address in the latest allocated region. In order to track the latest allocated region, a new auxiliary variable *last_region* is introduced. This variable is updated appropriately after each allocation operation. The predicate *valid_malloc* is then interpreted as follows:

$$\begin{aligned}
&valid_malloc(p, size) \equiv \\
&\quad M[addr_p] \neq 0 \implies M[addr_p] > 0 \wedge M[addr_p] \leq M[addr_p] + size \ \wedge \\
&\qquad (last_region = 0 \vee last_region + Size[last_region] \leq M[addr_p])
\end{aligned}$$

Hence, in the ordered mode, the memory model does not capture memory management strategies in which freed addresses will be reallocated. However, this mode greatly reduces the size of the generated SMT solver queries without sacrificing much precision. In particular, many errors due to imprecise reasoning about pointer arithmetic between fields and objects can still be detected. Note that during symbolic execution, a data structure “Regions” is maintained to keep track of all allocated regions along the current path. Using this auxiliary data structure, we can completely instantiate the quantifiers in the guard predicates and memory checks.

3.2 Burstall Memory Model

The main idea of the Burstall memory model [6] is to split the memory according to the types of allocated objects, making the assumption that pointers with different types will never alias. Apart from common scalar types, each struct field is also represented as a unique type. This model guarantees that updates to different fields of a struct will not interfere with each other. Consequently, it cannot capture union types or pointer arithmetic on fields inside a struct object. Cascade has a preprocessor that detects such operations and gives a warning when using the Burstall model.

To encode the Burstall memory model in Cascade, M is encoded as a record instead of a flat array. Each record element represents the state of the memory for one type in the C program. The exact type of M is shown in Fig. 9. The number of record elements is bounded by the number of structure types defined in the C program. Note that for each record element, if its type is a pointer, the element type of the corresponding array is $Addr$; otherwise, it is $Scalar$.

$$
\begin{array}{rlrl}
Ptr & : \text{uninterpreted type} & Scalar & : BitVec(m) \\
Offset & : BitVec(n) & Addr & : Ptr \times Offset
\end{array}
$$

$$
M : Record \left\{ \begin{array}{l} type_0 : (Addr \rightarrow Addr \mid Scalar), \\ type_1 : (Addr \rightarrow Addr \mid Scalar), \\ \ldots \\ type_k : (Addr \rightarrow Addr \mid Scalar) \end{array} \right\}
$$

$$
Size : Record \left\{ \begin{array}{l} type_0 : (Addr \rightarrow Scalar), \\ type_1 : (Addr \rightarrow Scalar), \\ \ldots \\ type_k : (Addr \rightarrow Scalar) \end{array} \right\}
$$

Fig. 9. Types of auxiliary variables for the encoding of the Burstall memory model

3.3 Partition Memory Model

The partition memory model is a novel experimental model implemented in Cascade. We here provide only an abridged summary of this model since a detailed description is beyond the scope of this paper.

In the partition model, the memory is divided according to distinct program pointers. A valid pointer has ownership of the associated memory region. This model allows arbitrary pointer arithmetic inside a region, as well as dereferencing pointers to any location inside a region. The model further supports all untyped operations except pointer aliasing. For example, consider a program that non-deterministically assigns either `&x` or `&y` to a pointer variable `s`. A subsequent update of `x`, respectively, `y` via pointer `s` would not be detected if `x` and `y` are assigned to regions that are disjoint from the region of `s`. To obtain a memory partition that takes into account pointer aliasing, Cascade incorporates Steensgaard's unification-based pointer analysis [26] as a preprocessing step. Each set of potentially aliasing pointers is assigned to one region. For the above example, the preprocessor will assign `x` and `y` to the same region. In most cases, the number of pointer classes is much larger than the number of types in the C code. Hence, the partition model often provides a more fine-grained partition of the memory into disjoint regions compared to the Burstall model. This can significantly speed up the analysis in some cases, which we confirm in our experimental evaluation.

Similar to Burstall's model, the state of the memory is encoded as a record. The detailed types of the auxiliary variables are shown in Figure 10. Every region has its own array, and the element type of the array can be determined by the type of the pointers associated with that region.

$$M : Record \left\{ \begin{array}{l} ptr_0 : BitVec(n) \rightarrow BitVec(m), \\ ptr_1 : BitVec(n) \rightarrow BitVec(m), \\ \ldots \\ ptr_k : BitVec(n) \rightarrow BitVec(m) \end{array} \right\}$$

$$Size : Record \left\{ \begin{array}{l} ptr_0 : BitVec(n) \rightarrow BitVec(m), \\ ptr_1 : BitVec(n) \rightarrow BitVec(m), \\ \ldots \\ ptr_k : BitVec(n) \rightarrow BitVec(m) \end{array} \right\}$$

Fig. 10. Types of auxiliary variables for the encoding of the partition memory model

Initially, the record is empty. During symbolic execution, new record elements are added for new variable definitions. If the execution context changes its scope we can safely delete those elements associated with pointers not in the current scope. In this way, the memory state tracks only the active pointers in the current scope. This significantly simplifies the query formula given to the SMT solver. Note that both the unordered mode and the ordered mode used in the flat model can also be applied to each region in the partition memory model.

3.4 Evaluation

We report on a set of experiments using the multiple memory models in Cascade to check properties of the NECLA suite of static analysis benchmarks [23]. These benchmarks contain C programs demonstrating common programming situations that arise in practice such as interprocedural data-flow, aliasing, array allocation, array size propagation and so on. We excluded benchmarks relying on string library functions and floating point number calculations. The results of our experiments are summarized in Table 1 and Table 2.[6] Note that these benchmarks have also been used in other recent tool papers such as the one introducing LLBMC [30] (which also included evaluations of CBMC 3.8, CBMC 3.9 [9] and ESBMC 1.16 [12]) and another introducing LAV [29] (which also evaluated CBMC, ESBMC, and KLEE [7]). For comparison purposes, we report our results in a similar format to that shown in [29] and also show the best result reported there (in the LAV column).

The benchmark suite includes both faulty and correct programs. There are two notable discrepancies with the results reported by LAV. For benchmark ex10.c, LAV reports an error while Cascade does not. The reason is that we made an additional assumption, namely that a pointer passed into the main function was properly allocated. Without this assumption, Cascade would find the same invalid address access as did LAV. The other discrepancy was in benchmark ex40.c. In this program, a loop iterates over an array of size 100 until the value 0 is found. If the array does not have a 0 entry, an out-of-bounds violation will occur in the 101st iteration. Cascade finds this bug if enough loop iterations are examined.

While collecting the statistics, we have compared the performance of the different memory models in Cascade[7]. For the flat memory model, we did not observe a significant performance improvement of the ordered mode over the unordered mode in most of the benchmarks. This is because the size of the benchmarks is limited and so is the size of the queries given to the SMT solver. However, the results for some benchmarks with a large number of loop unrollings (ex17-100, ex26-200), or with invariant reasoning (ex1-inv, ex18-inv) are encouraging. We also found that the ordered mode is slightly slower than the unordered mode for benchmark ex23-36, but we have not yet investigated why this is so.

Furthermore, from the results, we can see that both the Burstall model and partition model scale much better than the flat model – they solved the benchmarks (ex7-200, ex18-100, ex21-100, and ex22-50) that timed out in either LAV or Cascade with the flat model (or both). And the overall performance of partition model is much better than Burstall. In particular, benchmark ex27-200 was solved with the partition model, but timed out with the Burstall model. Partition model is the default memory model in Cascade.

[6] More information on the experiments including the benchmarks and control files is available at `http://cims.nyu.edu/~wwang1109/cascade/vmcai.html`.

[7] Note that the programs in this benchmark suite are all type-safe – the Burstall model is accurate enough to detect all bugs.

bnc.	F/V	#iter	LAV	Flat Model Unordered	Flat Model Ordered	Burstall Model	Partition Model
				Time(s)			
ex1	V	inv	-	68.561	50.079	0.645	**0.559**
		513	*	*	*	*	*
		3	0.35	0.895	0.663	0.512	0.771
ex2	V	inv	-	1.916	1.817	0.434	0.368
		1024	*	*	*	*	*
		3	0.47	0.534	0.535	0.332	0.42
ex3	F	inv	-	0.365	0.362	0.471	0.496
		10	-	0.576	0.552	0.558	0.774
		?	0.06	-	-	-	-
ex4	F	inv	-	0.39	0.425	0.375	0.452
		10	-	1.756	2.489	1.101	1.451
		?	0.24	-	-	-	-
ex5	V	-	0.02	0.136	0.132	0.114	0.109
ex6	V	-	0.11	0.187	0.134	0.159	0.142
ex7	V	inv	-	1.62	1.088	0.393	0.375
		200	*	*	*	9.266	**6.552**
		3	0.15	0.709	0.606	0.269	0.247
ex8	F	inv	-	0.173	0.128	0.193	0.193
		3	0.14	0.156	0.125	0.129	0.154
ex9	V	inv	-	0.666	0.768	0.452	0.441
		1024	*	*	*	*	*
		3	0.62	0.757	0.738	0.365	0.473
ex10	V	inv	-	2.795	2.605	0.94	1.115
		17	10.47	0.119	0.125	0.13	0.153
		3	1.14	0.763	1.407	0.471	0.689
ex11	V	3	0.08	0.215	0.222	0.211	0.215
ex12	F	10	-	1.099	0.985	0.724	0.948
		inv	-	0.363	0.376	0.428	0.381
		?	0.16	-	-	-	-
ex13	F	-	0.44	0.117	0.123	0.099	0.118
ex14	V	inv	-	0.344	0.332	0.338	0.304
		10	-	5.13	4.494	2.1	1.703
		?	0.13	-	-	-	-
ex15	V	-	0.34	1.731	1.522	0.235	0.233
ex16	F	inv	-	0.693	0.737	1.004	0.88
		4	-	0.927	0.929	0.955	0.931
		2	0.09F	0.22	0.232	0.266	1.82
ex17	V	inv	-	0.292	0.282	0.369	0.35
		100	-	19.274	17.068	29.193	**3.34**
		?	0.68	-	-	-	-

Table 1. Evaluation on NECLA Benchmarks. The experiments were conducted on a 1.7GHz, 4GB machine running Mac OS. A timeout (indicated by *) of 600 seconds was set for each experiment. V indicates the program verification succeeded, and F indicates the program contains a bug which was detected by Cascade. In the third column, "inv" indicates that deductive reasoning with a loop invariant was used; a number indicates the number of loop unrollings used; "-" indicates either that the program is loop-free or that a failure occurs before any loops are entered; and "?" indicates the unknown default iteration times used by LAV. In the fourth and later columns, "-" indicates that there is no corresponding result for that loop configuration.

bnc.	F/V	#iter	Time(s) LAV	Flat Model Unordered	Flat Model Ordered	Burstall Model	Partition Model
ex18	V	inv	-	424.414	65.531	**0.446**	0.804
		100	*	*	*	359.59	**7.066**
		10	3.0	8.088	9.412	2.527	1.298
ex19	F	inv	-	0.163	0.17	0.183	0.172
		3	0.08	0.371	0.424	0.466	0.387
ex20	F	inv	-	0.385	0.399	0.451	0.417
		1024	*	*	*	*	*
		1	0.32	0.498	0.389	0.315	0.29
ex21	V	inv	-	0.757	0.735	1.26	0.782
		100	-	*	*	**12.673**	16.2
		?	0.36	-	-	-	-
ex22	V	50	-	*	*	14.919	**12.133**
	V	?	4.1	-	-	-	-
ex23	V	inv	-	1.015	1.069	0.571	0.418
		36	6.46	27.44	34.0	**2.191**	2.847
ex25	F	inv	-	0.965	0.989	1.163	1.208
		3	0.20	1.654	1.046	0.876	1.506
ex26	F	inv	-	0.546	0.613	0.748	0.687
		200	-	31.122	28.062	20.314	**7.877**
		?	0.62	-	-	-	-
ex27	F	inv	-	1.584	1.913	1.219	4.68
		200	-	*	*	*	**55.585**
		?	5.28	-	-	-	-
ex30	F	-	-	0.134	0.147	0.596	0.395
		?	0.24	-	-	-	-
ex31	V	inv	-	0.308	0.312	0.348	0.317
		7	5.62	0.62	0.976	0.624	0.432
ex32	V	inv	-	0.892	0.826	1.067	0.608
		1000	-	*	*	*	*
		?	0.5	-	-	-	-
ex34	V	-	0.24	0.416	3.141	0.441	0.508
ex37	F	-	0.20	0.107	0.143	0.131	0.132
ex39	F	inv	-	0.228	0.238	0.292	0.255
		3	0.07	0.306	0.273	0.641	0.307
ex40	F	inv	-	0.323	0.288	0.336	0.314
	V	3	0.10	0.345	0.307	0.271	0.25
ex41	F	inv	-	0.24	0.23	0.267	0.25
		3	0.44	0.515	0.332	0.214	0.25
ex43	F	-	-	1.793	1.318	1.041	1.271
		inv	-	0.907	0.857	0.731	0.794
		?	17.91	-	-	-	-
ex46	F	3	*	0.127	0.124	0.174	0.159
ex47	F	inv	-	12.925	9.365	4.563	1.779
		2	1.38	0.315	0.449	0.192	0.235
ex49	F	inv	-	0.33	0.339	0.414	0.378
	V	3	0.08	0.24	0.246	0.235	0.204
inf1	F	-	0.22	0.261	0.29	0.261	0.282
inf2	F	-	1.25	0.223	0.158	0.185	0.195
inf4	F	-	0.38	0.403	0.502	0.345	0.662
inf5	F	-	0.15	0.221	0.235	0.15	0.209
inf6	V	-	0.12	0.247	0.243	0.207	0.216
inf8	V	-	0.19	0.333	0.417	0.323	0.439

Table 2. Evaluation on NECLA Benchmarks (continued).

4 Reasoning about Linked Data Structures

In this section, we discuss how to extend Cascade to reason about properties of linked data structures. Analysis of such data structures typically requires a *reachability predicate* to capture the unbounded number of dynamically allocated cells present in a linked list. For a given address u, the reachability predicate characterizes the set of cells $\{u, u.f, u.f.f, \ldots\}$ reachable from u via continuously visiting field f.

4.1 Theory of Reachability in Linked Lists

LISBQ. Rakamarić et al. [21] presented a ground logic and an NP decision procedure for reasoning about reachability in liked list data structures. The logic provides a ternary predicate $x \xrightarrow{f} z \xrightarrow{f} y$, which we refer to as the *between predicate*. The predicate states that cell y is reachable from cell x via field f, yet, not without going through cell z first. In other words, z is between x and y. Binary reachability $reach(f, x, y)$ via field f can then be expressed as $x \xrightarrow{f} y \xrightarrow{f} y$. The between predicate enables precise tracking of reachability information during symbolic execution of heap updates that modify field f (potentially creating cycles in the heap). In [19], Lahiri and Qadeer presented the logic of interpreted sets and bounded quantification (LISBQ), which includes the between predicate but also admits reasoning about the content of unbounded list data structures. They showed that LISBQ is still decidable in NP using a decision procedure that builds on an SMT solver.

LISBQ as a local theory extension. More recently, we explored the connection between Lahiri and Qadeer's result to local theory extensions [25]. A theory extension is a first-order theory that is defined by extending a base theory with additional symbols and axioms. For example, the theory of arrays over integer indices can be formalized as a theory extension where the base theory is the theory of linear integer arithmetic, the extension symbols are the array store and select functions, and the extension axioms are McCarthy's select over store axioms.

A theory $\mathcal{T}$ is called *local* if satisfiability modulo $\mathcal{T}$ can be decided by reduction to the base theory via local instantiation of the extension axioms. Here, local instantiation means that only those axiom instances are considered that do not introduce new terms to the input formula. Local theory extensions are interesting because they provide completeness guarantees for the quantifier instantiation heuristics implemented in modern SMT solvers, and at the same time give a simple syntactic restriction on the kinds of axiom instances that need to be considered.

In [28], we showed that LISBQ can be formalized as a local theory extension. This yields an interesting generalization of previous results in [19, 21]. For example, the base theory can now provide an interpretation of memory cells, e.g., as bitvectors, which results in a theory of reachability that admits address

arithmetic. Another generalization obtained this way is to interpret fields as arrays. The formulas generated during symbolic execution in [19, 21] can grow exponentially in the number of store operations $x.f := y$ along the executed path. The encoding of fields as arrays avoids this exponential blowup by deferring case splits on store operations to the SMT solver. Finally, the connection to local theory extensions also provides new possibilities to further improve the efficiency of SMT-based decision procedures for LISBQ.

4.2 Linked Lists in Cascade

We have explored some of these possibilities in the context of Cascade. We added a reachability predicate to the C expression language usable in assumptions and assertions. The axioms of LISBQ are encoded via a new theory axiom encoding module. The base theory interprets memory cells as bitvectors of fixed width in order to model pointer arithmetic; and fields are interpreted as arrays mapping bitvector indices to bitvector values. Cascade then instantiates the LISBQ theory axioms for the ground terms appearing in each SMT query and hands the axiom instances together with the query to the SMT solver.

In order to keep the input formula to the SMT solver relatively small, we exploit our results on locality: we only partially instantiate the theory axioms. That is, we only instantiate quantified variables that appear below function symbols in the axioms, while keeping the remaining variables quantified. The resulting quantified formulas fall into fragments for which the quantifier instantiation heuristics that are implemented in SMT solvers are guaranteed to be decision procedures. Partial instantiation provides a good compromise between an approach where we only rely on the solver's heuristics and do not instantiate axioms upfront, and an approach where we fully instantiate the axioms and do not use the heuristics in the solver at all. The former is typically fast but incomplete on satisfiable input formulas, the latter is complete but typically slow. This is confirmed by our experimental evaluation. In fact, often partial instantiation yields the best running time.

Example. Figure 11 shows the code and control file of a small list-manipulating procedure in C that appends two lists together. In this test case, we are interested in verifying that after the procedure returns, the head of the first list `l1` can reach the head of the second list `l2`.

Evaluation. In order to evaluate Cascade's new ability to reason about linked data structures, we chose two suites of benchmark programs manipulating singly-linked lists (SL) and doubly-linked lists (DL), respectively. In these benchmarks, various nontrivial reachability-related assertions are checked. The evaluation was performed on a 1.7GHz, 4GB machine running Mac OS. For each benchmark, the time limit was set to 60 seconds. The results appear in Table 3. We used three different instantiation heuristics for the quantified axioms, and partial instantiation is much more efficient than the other two options in most cases.

```
#define NULL (int *) 0

typedef struct NodeStruct {
  struct NodeStruct *next;
  int data;
} Node;

void append(Node *l1, Node *l2) {
  ASSUME(create_acyclic_list(l1, 5)
      && create_acyclic_list(l2, 5));

  Node *l = l1;
  Node *e = l1;
  Node *last = NULL;

  while (e) {
    last = e;
    e = e->next;
  }

  if (!last)
    l = l2;
  else
    last->next = l2;

  ASSERT(reach(next, l1, l2));
}
```

```
<controlFile>
  <sourceFile name="list_append.c"
              id="1" />
  <run>
    <startPosition fileId="1"
                   line="8" />
    <wayPoint fileId="1" line="16" >
      <loop iterTimes="5" />
    </wayPoint>
    <wayPoint fileId="1" line="21" />
    <endPosition fileId="1"
                 line="27" />
  </run>
</controlFile>
```

Fig. 11. list_append.c and list_append.ctrl. The predicate $create_acyclic_list(l, 5)$ indicates that l is a singly-linked list of size 5. The predicate $reach(next, l1, l2)$ indicates that $l1$ can reach $l2$ by following the link field $next$.

This is noteworthy considering that SMT solvers use sophisticated instantiation heuristics internally.

Note that for the DL benchmarks with invalid assertions, both no instantiation and full instantiation time out most of the time, while partial instantiation reports "unknown" immediately. For benchmarks with quantifiers that don't fall into a know complete fragment, an "unknown" indicates that the SMT solver was unable to find a proof using its instantiation heuristics. Thus, an unknown result should be considered the same as an invalid (satisfiable) result with the understanding that it could be a false positive. In other words, an immediate "unknown" result is the best we could hope for in this situation.

5 Related Work

In the last decade, a variety of SAT/SMT-based automatic verifiers for C programs have been developed, such as bounded model checkers (CBMC [9], ESBMC [12], LLBMC [30], LAV [29], Corral [20] and Cascade), symbolic execution tools (KLEE [7]), and modular verifiers (VCC [10], HAVOC [8], and Frama-C [13]). In most cases, these tools use either flat memory models (e.g., CBMC, LLBMC, ESBMC, LLBMC, LAV, KLEE and early versions of VCC), or Burstall-style memory models (e.g., Corral and Caduceus [15]).

Benchmark	F/V	NI	PI	FI	Benchmark	F/V	NI	PI	FI
sl_append	V	0.02	**0.02**	0.08	dl_append	V	**0.08**	0.34	0.14
sl_contains	V	**0.01**	0.01	0.05	dl_contains	V	**0.01**	0.03	0.02
sl_create	V	0.13	**0.05**	3.12	dl_create	V	12.98	**0.18**	1.81
sl_filter	F	0.11	**0.08**	0.13	dl_filter	F	*	0.31^{UN}	*
sl_findPrev	V	**0.02**	0.05	0.09	dl_findPrev	V	**0.05**	0.07	0.06
sl_getLast	V	0.01	**0.01**	0.06	dl_getLast	V	3.34	**0.02**	1.27
sl_insertBefore	V	**0.20**	0.26	3.03	dl_insertBefore	V	*	**2.74**	*
sl_partition	F	0.08	**0.08**	0.09	dl_partition	F	*	0.23^{UN}	*
sl_remove	F	1.34	**0.07**	1.76	dl_remove	F	*	0.68^{UN}	*
sl_removeLast	F	0.03	**0.02**	0.06	dl_removeLast	F	*	0.11^{UN}	*
sl_reverse	V	0.09	**0.06**	1.57	dl_reverse	V	*	**18.65**	*
sl_split	F	0.05	**0.03**	0.07	dl_split	F	*	0.11^{UN}	*
sl_traverse	V	0.03	**0.01**	0.52	dl_traverse	V	8.68	**0.02**	*

Table 3. Results on singly- and doubly-linked list benchmarks. A timeout (indicated by *) of 600 seconds was set for each experiment. NI is for no instantiation, PI is for partial instantiation, and FI is for full instantiation. The superscript UN indicates that the result from the SMT solver is "unknown".

As we have discussed, these models force the user to choose between scalability and being able to capture the effects of type-unsafe behaviors. The VCC developers proposed a typed memory model that attempts to strike a balance between scalability and precision [11]. This model maintains a set of valid pointers with disjoint memory locations, and restricts memory accesses only to them. Special code annotation commands called split and join are introduced to switch between a typed mode and a flat mode. However, the additional axioms introduced for the mode switching slow down reasoning [4]. Böhme et al. use a variant of the VCC model [4] but few details are given. A variant of Burstall's model is proposed in [22]. It employs a type unification strategy that simply removes the uniqueness of relative type constants when detecting type casts. However, this optimization is too coarse to handle code with even mild use of low-level address manipulations and type casts, as the memory model will quickly degrade into the flat model.

Frama-C also develops several memory models at various abstraction levels: Hoare, typed, and flat models. As an optimization strategy, Frama-C mixes the Hoare model and flat model by categorizing variables into two classes: logical variables and pointer variables. The Hoare model is used to handle the logical variables and the flat model manages the pointer variables. This strategy is similar to our partition model. However, our partition model provides a more fine-grained partition for the pointer variables.

Our partition model is similar to the memory model of VCC that divides memory based on various pointers. The main difference is that we map the pointers to separately updatable memory regions, and thus ease the burden of SMT axiomatization for distinguishing pointers. Steensgaard's pointer analysis

is incorporated to control the issue of pointer aliasing. Compared to the VCC model, our modeling seems more natural – we can detect untyped operations before memory splitting, and thus avoid switching between typed and flat modes. The direct performance comparison is difficult because of VCC's contract based approach to verification. However, results from [4] seem to confirm the folk wisdom that splitting the heap into disjoint regions performs best.

6 Conclusion

In this paper, we presented the latest version of Cascade, an automatic verifier for C programs. It supports multiple memory models in order to balance efficiency and precision in various ways. Our empirical evaluation shows that Cascade is competitive with other tools. Furthermore, we have shown that with a modest effort, it can be extended to reason about simple properties of linked data structures.

In the future, we will integrate an invariant inference engine to relieve the annotation burden on users. Moreover, we are planning to support procedure contracts that enable local reasoning via frame rules.

Acknowledgments

We would like to thank Christopher Conway, Morgan Deters, Dejan Jovanović, and Tim King for their contributions to the design and implementation of the Cascade tool. This work was supported by NSF grants CCF-0644299 and CCS-1320583.

References

1. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. *Construction and Analysis of Safe, Secure and Interoperable Smart Devices*, pages 49–69, 2005.
2. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. 6806:171–177, 2011.
3. A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. *Proceedings of Design Automation Conference (DAC'99)*, 317:226–320, 1999.
4. S. Böhme and M. Moskal. Heaps and data structures: A challenge for automated provers. *N. Bjørner and V. Sofronie-Stokkermans, editors, Automated Deduction*, 6803:177–191, 2011.
5. D. Brand and W. H. Joyner. Verification of protocols using symbolic execution. *Comput. Networks*, 2:351, 1978.
6. R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.
7. C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. *Proc. OSDI 2008*, pages 209–224, 2008.

8. S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamarić. A reachability predicate for analyzing low-level software. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 19–33, 2007.
9. E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. *Proc. TACAS 2004*, 2988:168–176, 2004.
10. E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs '09)*, 2009.
11. E. Cohen, M. Moskal, S. Tobies, and W. Schulte. A precise yet efficient memory model for c. *ENTCS*, 254:85–103, 2009.
12. L. Cordeiro, B. Fischer, and J. Marques-Silva. SMT-based bounded model checking for embedded ansi-c software. *ASE*, 0:137–148, 2009.
13. P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c a software analysis perspective, 2012.
14. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. *TACAS*, pages 337–340, 2008.
15. J. Filliâtre and C. Marché. Multi-prover verification of C programs. *International Conference on Formal Engineering Methods (ICFEM '04)*, pages 15–29, 2004.
16. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. *Programming Language Design and Implementation (PLDI)*, pages 234–245, 2002.
17. R. Grimm. Rats!, a parser generator supporting extensible syntax. 2009.
18. J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 385:226–394, 1976.
19. S. K. Lahiri and S. Qadeer. Back to the future. revisting precise program verification using SMT solvers. *POPL*, pages 171–182, 2008.
20. A. Lal, S. Qadeer, and S. K. Lahiri. Corral: A solver for reachability modulo theories. *CAV*, 2012.
21. Z. Rakamarić, J. Bingham, and A. J. Hu. An inference-rule-based decision procedure for verification of heap-manipulating programs with mutable data and cyclic data structures. *VMCAI*, pages 106–121, 2007.
22. Z. Rakamarić and A. J. Hu. A scalable memory model for low-level code. *VMCAI*, pages 290–304, 2009.
23. S. Sankaranarayanan. Necla static analysis benchmarks. 2009.
24. N. Sethi and C. Barrett. CASCADE: C assertion checker and deductive engine. *CAV 2006*, 4144:166–169, 2006.
25. V. Sofronie-Stokkermans. Interpolation in local theory extensions. *Logical Methods in Computer Science*, 4:4, 2008.
26. B. Steensgaard. Points-to analysis in almost linear time. *ACM Symposium on Principles of Programming Languages*, pages 32–41, 1996.
27. A. Stump, C. W. Barrett, D. L. Dill, and J. Levitt. A decision procedure for an extensional theory of arrays. *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, page 29, 2001.
28. N. Totla and T. Wies. Complete instantiation-based interpolation. *POPL*, 2013.
29. M. Vujosevic-Janicic and V. Kuncak. Development and evaluation of LAV: an SMT-based error finding platform. *Proc. VSTTE*, 2012.
30. M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. *SIGSOFT Softw. Eng*, page 29, 2004.