An approach to model checking C based on an explicit semantics

Stella Lau
Trinity College

UNIVERSITY OF CAMBRIDGE

A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for the Computer Science Tripos, Part III

University of Cambridge
Department of Computer Science and Technology
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: sl715@cam.ac.uk

June 2018

Declaration

I, Stella Lau of Trinity College, being a candidate for Computer Science Tripos, Part III, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed:

Date: 1 June 2018

Abstract

Formal verification augments traditional bug-finding techniques by proving or disproving correctness properties. The correctness of C programs, widely used throughout computing infrastructure, is critical. However, C lacks a clear and unambiguous semantics—the semantics of C is informally specified by the ANSI/ISO standards in prose form. Consequently, the semantics of C assumed by compilers, programmers, and existing C code have diverged both from the ISO standard and from each other. Formal verification tools for C must implicitly assume a semantics of C.

This project presents an approach to formal verification of C code using an explicit C semantics, Cerberus. Cerberus is a formal model for C that aims to both formalize a large fragment of the ISO standards and reconcile the ISO standards with C as used in practice. The Cerberus model is expressed through translation to a Core language that makes explicit many subtle aspects of C such as undefined behaviour, evaluation order, and unspecified values. We reduce correctness of C programs to correctness of corresponding Core programs, and reduce correctness of Core programs to satisfiability of SMT problems.

From a formal methods point of view, we demonstrate a principled approach to formal verification of ISO C11 based on a clear interpretation of C semantics. From a practical point of view, we implement a bounded model checker that exhaustively explores the behaviour of programs in fragments of both sequential and concurrent C, based on the C11 concurrency model.

Total word count: 11993[1]

[1] The word count was computed by TEXcount, `http://app.uio.no/ifi/texcount/`. This word count includes footnotes, but omits code listings.

Acknowledgements

I would like to express my utmost gratitude to my supervisor, Peter Sewell, for all his time, support, and guidance throughout this year. This work would also not have been possible without the support of Victor Gomes, who provided valuable insights in our discussions, and Kayvan Memarian, who patiently introduced me to Cerberus and promptly and thoroughly responded to all my questions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The correctness of hardware and software systems is critical. Although simulation and testing are well-established error-finding methods, exhaustive analysis of all possible behaviours is generally infeasible and thus so is guaranteeing the absence of bugs with these methods. Understanding the behaviour of systems code is especially challenging due to the low-level, and often architecture-specific, constructs used to generate heavily-optimized and efficient code. For instance, concurrent programming is pervasive to exploit multi-core hardware, yet its non-determinism is difficult to reason about and makes concurrency bugs difficult to diagnose [41]. Formal verification augments traditional bug-finding techniques by mathematically proving or disproving correctness properties.

The C programming language is widely used in systems programming and throughout our computing infrastructure. Writers of C code, compilers, and analysis tools need to handle the semantics of C and correctly reason about pointers, memory objects, and concurrency. However, the semantics of C is informally specified by the ANSI/ISO standards in prose form, and is ambiguous and lacking mathematical precision.

This dissertation presents an approach to automatic formal verification of C code using an explicit C semantics, Cerberus. Cerberus, introduced by Memarian et al. [44], is a semantic model for C that aims to both formalise a large fragment of the ISO standards and reconcile the ISO standards with C as used in practice, the *de facto* standards. From a formal methods point of view, we demonstrate a principled approach to verification of C that is based on a clear interpretation of C semantics. From a practical point of view, we present a prototype *bounded model checker* that exhaustively explores the behaviour of programs in fragments of both sequential and concurrent C, based on the C11 concurrency model [13, 20].

1.1 Motivation

This project is motivated by the following:

Principled approach to correctness Formal verification can be defined as mathematically proving correctness with respect to a *formal specification*. Automatic formal verification tools for C programs generate a formal specification from C source code, expressing safety conditions under which erroneous behaviours do not occur. C programs are then checked against the generated specification. This process implicitly makes assumptions about C's language semantics, with the interpretation of the language tightly coupled with the implementation of the verification tool.

C lacks an unambiguous and mathematically precise semantics: the language is specified in prose form in the ISO standard, with ambiguities and subtleties leading to different interpretations of C programs. Consequently, it is often unclear what exactly is being verified. A more thorough approach is to explicitly base analysis on a formalisation of the ISO standard. We take a principled approach to building a model checker, by using a formal model for C (Cerberus) to translate C code into an intermediate "Core" language with clear and consistent semantics, and expressing the correctness of the resulting Core program as an SMT (satisfiability modulo theories) problem.

Explicit, de facto C semantics While a number of projects [13, 33, 34, 38] have worked on formalising aspects of the C semantics, there are discrepancies between the ISO de facto standards; compilers, legacy C code, and analysis tools assume behaviours either inconsistent with, or not clearly specified in, the ISO standard [44].

The ISO standard defines the notion of *undefined behaviour* [37, §3.4.3]:

> *behavior, upon which use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements*

From this ISO point of view, compilers are free to optimize programs under the assumption that undefined behaviour never occurs. Consequently, code that works when compiled without optimizations may fail when compiled with a higher optimization level, potentially due to the compiler eliminating code considered dead [56]. Large projects such as the Linux kernel are compiled with flags such as `-fno-strict-overflow` and `-fno-strict-aliasing` to instruct the compiler to not assume certain behaviours classified as undefined in the ISO standard (such as signed integer overflow and dereferencing a pointer aliasing another of an "incompatible type") are undefined. These flags explicitly ask for a different semantics than that described in the ISO standard.

As described by Bessey et al. [14], based on experiences with commercializing a static analysis tool, reporting "false" positives that are contrary to users' expectations (despite these being real errors as defined by ISO) is not always desirable. These inconsistencies arise from discrepancies between the ISO and de facto standards. Analysis tools thus *implicitly* encode these de facto standards by, for instance, disabling certain analyses. We aim to build our tool based upon an *explicit* formal model of C that unifies the ISO and de facto standards and precisely defines a range of allowed behaviours.

Relaxed memory model Programming languages define an interface to memory. In multithreaded programs, the memory model defines the semantics of memory accesses. The simplest (and "strongest") memory model is *sequential consistency* (SC), in which atomic memory accesses between threads are interleaved in a total order, with each read reading from the most recent write to the same address. This is often implemented by restricting compiler optimisations and inserting hardware fence instructions appropriately into compiled code.

C/C++11, as well as hardware architectures such as ARM and x86, has a weak memory model, with atomic memory actions without the aforementioned interleaving semantics, for efficient implementation. For example, *release-acquire* accesses allow "message passing" between threads without the full implementation cost of SC, and the weaker and less expensive *relaxed* accesses are intended to be compiled to simple loads and stores with minimal synchronization guarantees.

The possible executions arising from the C11 concurrency model may be challenging to reason about and exhaustively test [41]. We aim to verify C programs based on the C11 relaxed memory semantics and provide tool support to allow programmers to explore the range of allowed behaviours.

1.2 Contributions

We present a *principled* approach to building a bounded model checker that formally verifies correctness of a fragment of C by proving the existence or absence of errors such as failed user-defined assertions and undefined behaviours. We verify correctness with respect to a formal model of ISO C11, such that the interpretation of C programs is precise and unambiguous. Finally, we produce a tool enabling exploration and visualization of possible executions based on C11's relaxed memory model along the lines of CPPMEM [12], but with the ability to handle a larger fragment of C.

We demonstrate our approach on a fragment of C without arbitrary pointer arithmetic and pointer type casting. Our model checker has two modes: a simplified *sequential* mode for single-threaded programs which assumes a total memory order, and a *concurrent* mode that implements the C11 relaxed memory model as formalised by Batty et al. [11, 13], with the exception of locks, fences, and read-modify-writes.

Figure 1.1 presents the model checker architecture. The Cerberus semantics are expressed via translation from C to a typed Core language, which is parametrised on a specific implementation and decoupled from the memory layout and concurrency

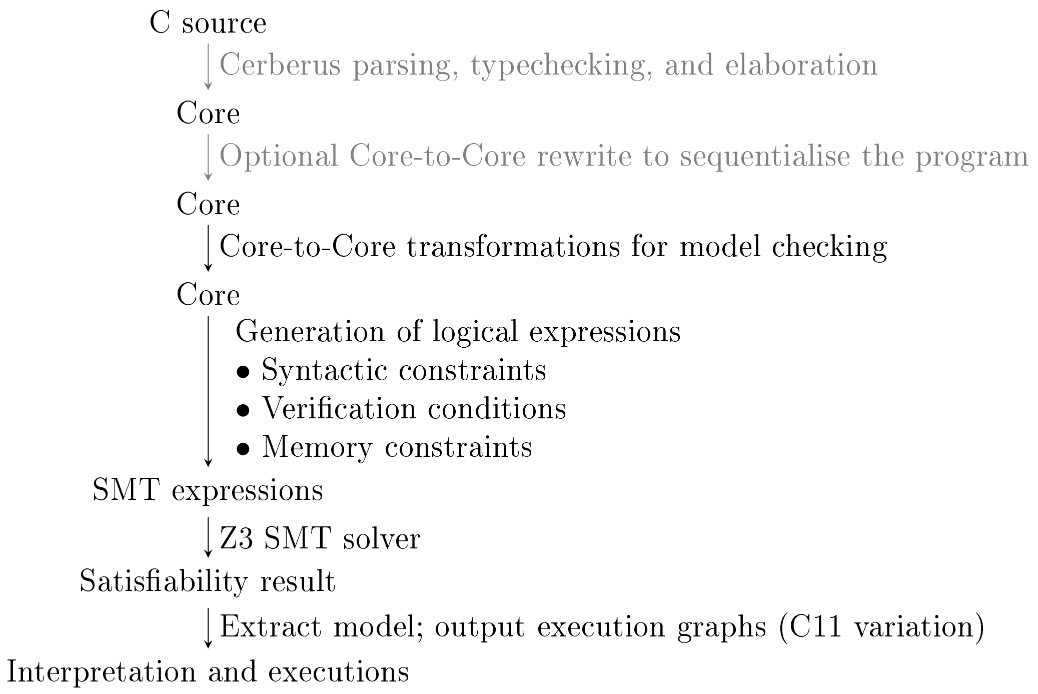

C source
Cerberus parsing, typechecking, and elaboration
Core
Optional Core-to-Core rewrite to sequentialise the program
Core
Core-to-Core transformations for model checking
Core
Generation of logical expressions
• Syntactic constraints
• Verification conditions
• Memory constraints
SMT expressions
Z3 SMT solver
Satisfiability result
Extract model; output execution graphs (C11 variation)
Interpretation and executions

Figure 1.1: Model checker architecture. The gray components are part of the Cerberus architecture and systems [44]; the other parts were implemented in this project.

models, and an optional Core-to-Core rewrite sequentialising the program (for the sequential model). We implement Core-to-Core rewrites to facilitate model checking, replacing implementation-defined expressions with a given implementation and inlining pure functions. The resulting Core program is translated into a set of first-order logic expressions representing syntactic constraints, verification conditions, and memory constraints, with the memory constraints differing between the sequential and concurrent models. The constraints are passed to Z3 [32], an SMT solver, which either produces a counterexample trace for a violation of a safety property or a proof that none exists. For the concurrent mode, all possible executions are extracted and presented to the user.

Chapter 2

Preliminaries

This chapter provides the necessary background to understand the rest of the report. We assume basic knowledge of C. § 2.1 recalls the notion of undefined behaviours and correctness of C programs. § 2.2 introduces the Cerberus model and Core language. Finally, § 2.3 introduces SMT-based bounded model checking.

2.1 Correctness of C programs

There is a design trade-off in the extent to which a programming language under-specifies program behaviour to allow compilers to generate efficient code for a target architecture. Programmers would like to predict both the performance and behaviour of code, preferring that programs behave identically across platforms. On the other hand, programmers want high performance obtainable by allowing compilers to exploit properties of target architectures.

For example, x86 raises an exception for division-by-zero, whereas PowerPC ignores it [55]. Rather than enforce uniform behaviour across instruction sets (i.e. by forcing compilers for PowerPC to generate instructions detecting division-by-zero), the C language specification defines division-by-zero as undefined behaviour [37, §6.5.5]. Compilers are free to assume undefined behaviour does not occur, and can optimize programs based on this assumption.

The ISO standard defines the following types of compiler freedom, which can result in programs exhibiting different behaviours when compiled for different architectures or with different parameters:

unspecified behaviour

> *use of an unspecified value, or other behavior where this International Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance*

The compiler chooses one possibility from a set of possibilities for each instance. For example, the evaluation order of arguments to functions and certain operators

Example	Condition	Undefined behaviour
`x/y`, `x%y`	$\mathtt{y} = 0$	division-by-zero
`*p`	$\mathtt{p} = \mathtt{NULL}$	null pointer deference
`a[i]`	$\mathtt{i} \geq \mathtt{ARRAY\_SIZE}(\mathtt{a})$	buffer overflow
$\mathtt{x} + \mathtt{y}$	$\mathtt{x}_\infty + \mathtt{y}_\infty \notin [-2^{\mathtt{n}-1}, 2^{\mathtt{n}-1} - 1]$	signed integer overflow
$\mathtt{x} << \mathtt{y}, \mathtt{x} >> \mathtt{y}$	$\mathtt{y} < 0 \vee \mathtt{y} \geq \mathtt{n}$	negative or oversized shift
use `q` after `free(p)`	`alias(p, q)`	use after free
`int f() {}`	value of `f()` used	end of function w/o return

Table 2.1: Examples of undefined behaviour, largely taken from [56]. Here, `p` and `q` are pointers, `x` and `y` are `n`-bit integers, $\mathtt{x}_\infty$ means to consider `x` as an infinitely-ranged mathematical integer, and `alias(p, q)` is true iff `p` and `q` point to the same object.

such as `(+)` is unspecified.

```
int foo(void) { printf("foo"); return 0;}
int bar(void) { printf("bar"); return 0;}
int main(void) {
    int x = foo() + bar();
    printf("\n");
}
```

The above program can output either `foobar` or `barfoo`.

implementation-defined behaviour

unspecified behavior where each implementation documents how the choice is made

The compiler chooses and documents a behaviour and obeys it consistently. For example, `sizeof(int)` is implementation defined.

undefined behaviour

behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements

Compilers can assume undefined behaviour never occurs and if it does, the behaviour of the program is entirely unconstrained. Undefined behaviour occurring is clearly undesirable.

In this report, we say a program is erroneous or incorrect if it can exhibit undefined behaviour or failed assertions. We prove correctness with respect to a specific implementation, accounting for all possibilities with regard to unspecified behaviours.

2.1.1 Undefined behaviour

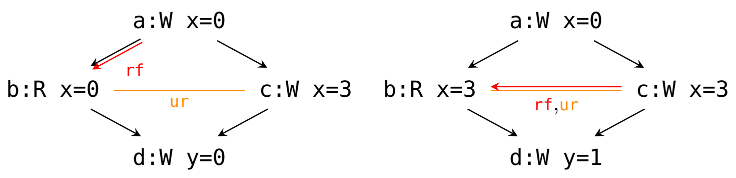
Table 2.1 provides some examples of undefined behaviour. The ISO standard defines ~200 types of undefined behaviour. These are all defined in prose form and require non-trivial understanding of the semantics. For example, the following is undefined behaviour [37, §J.2]:

> Pointers that do not point to the same aggregate or union (nor just beyond the same array object) are compared using relational operators.

This requires understanding of pointer aliasing, which depends on the memory object layout model.

Another source of undefined behaviour is *unsequenced races* (*ur*). In C, reads to the same location can be unordered. However, a write that is unordered with respect to another read or write from the same thread and on the same location is considered undefined behaviour.

```c
int main() {
  int x = 0;
  int y = (x == (x=3));
}
```



The program order, represented by black arrows, permits either of the above executions with `(b)` reading from (*rf*) `(a)` or `(c)`, resulting in `y=0` or `y=1`. This program has undefined behaviour.

2.2 Cerberus and the Core language

Cerberus [44] is a formal model for a large fragment of C designed to be in close correspondence with the ISO text. The Cerberus semantics are defined by elaboration: after parsing and typechecking, it compositionally translates C into a typed "Core" language that makes explicit many subtle aspects of C such as undefined behaviours, evaluation order, and integer conversions. The syntax of Core is summarized in Figure 2.1.

Core is essentially a typed, first-order lambda calculus extended with features such as `run` and `save` to represent constructs such as loops and gotos. The type system distinguishes between *pure* expressions (expressions without memory actions, sequencing, or multiple threads) and effectful expressions, with the elaboration aiming to map C expressions into pure expressions where possible.

Figure 2.2 provides an example of the elaboration to Core of the following C program.

oTy ::= **types for C objects**

\| `integer`
\| `floating`
\| `pointer`
\| `cfunction` (oTy, $\overline{oTy_i}^{\,i}$)
\| `array` (oTy)
\| `struct` tag
\| `union` tag

bTy ::= **Core base types**

| `unit`	unit
| `boolean`	boolean
| `ctype`	Core type of C type exprs
| ($\overline{bTy_i}^{\,i}$)	tuple
| oTy	C object value
| `loaded` oTy	oTy or unspecified

pat ::= **Core patterns**

| _	wildcard pattern
| $ident$	identifier pattern
| $ctor(pat_1, .. ,pat_n)$	constructor pattern

$ctor$ ::= **Core constructors**

| `Tuple`	tuple
| `Ivmax`	max integer value
| `Ivmin`	min integer value
| `Ivsizeof`	sizeof value
| `Ivalignof`	alignof value
| `IvOR`	bitwise OR
| `Iv..`	bitwise ..
| `Specified`	non-unspecified loaded value
| `Unspecified`	unspecified value

$value$ ::= **Core values**

| $object\_value$	C object value
| $loaded\_value$	loaded C object value
| `Unit`	unit
| `True`	true
| `False`	false
| $ctype$	C type expr as value
| $(value_1, .. ,value_n)$	tuple

a ::= **memory actions**

\| `create` (pe_1,pe_2)
\| `alloc` (pe_1,pe_2)
\| `kill` (pe)
\| `store` (pe_1,pe_2,pe,$mem\text{-}order$)
\| `load` (pe_1,pe_2,$mem\text{-}order$)
\| `rmw` (pe_1,pe_2,pe_3,pe_4,$mem\text{-}order_1$,$mem\text{-}order_2$)
\| `fence` (pe)

pa ::= **memory actions with polarity**

| a	positive, sequenced by `let weak` and `let strong`
| `neg` (a)	negative, only sequenced by `let strong`

pe ::= **Core pure expressions**

| $ident$	Core identifier
| <$impl\text{-}const$>	implementation-defined constant
| $value$	value
| `undef` ($ub\text{-}name$)	undefined behaviour
| `error` ($string$,pe)	impl-defined static error
| $ctor(pe_1, .. ,pe_n)$	constructor application
| `case` pe `with` $\overline{\vert pat_i \texttt{=>} pe_i}^{\,i}$ `end`	pattern matching
| `array_shift`(pe_1,$ctype$,pe_2)	pointer array shift
| `member_shift`(pe,tag.$member$)	pointer struct/union member shift
| `not` (pe)	boolean not
| pe_1 $binop$ pe_2	binary operators
| (`struct` tag){ $\overline{.member_i = pe_i}^{\,i}$ }	C struct expression
| (`union` tag){.$member$ = pe}	C union expression
| $name(pe_1, .. ,pe_n)$	pure Core function call
| `let` pat = pe_1 `in` pe_2	pure Core let
| `if` pe `then` pe_1 `else` pe_2	pure Core if
| `is_scalar`(pe)	
| `is_integer`(pe)	
| `is_signed`(pe)	
| `is_unsigned`(pe)	

e ::= **Core expressions**

| `pure` (pe)	pure expression
| `ptrop` ($ptrop$,pe_1, .. ,pe_n)	pointer op involving memory
| pa	memory action
| `case` pe `with` $\overline{\vert pat_i \texttt{=>} e_i}^{\,i}$ `end`	pattern matching
| `if` pe `then` e_1 `else` e_2	Core if
| `skip`	skip
| `ccall` (pe,pe_1, .. ,pe_n)	Core cfunction call
| `unseq` (e_1, .. ,e_n)	unsequenced expressions
| `let weak` pat = e_1 `in` e_2	weak sequencing
| `let strong` pat = e_1 `in` e_2	strong sequencing
| `let atomic` $(sym : oTy)$ = a_1 `in` pa_2	atomic sequencing
| `indet` [n](e)	indeterminately sequenced expr
| `bound` [n](e)	...and boundary
| `nd` (e_1, .. ,e_n)	nondeterministic sequencing
| `save` $label$($\overline{ident_i : ctype_i}^{\,i}$) `in` e	save label
| `run` $label$($\overline{ident_i := pe_i}^{\,i}$)	run from label
| `par` (e_1, .. ,e_n)	cppmem thread creation
| `wait` ($thread\text{-}id$)	wait for thread termination

$definition$ ::= **Core definitions**

| `fun` $name$($\overline{ident_i : bTy_i}^{\,i}$) : bTy := pe	Core function definition
| `proc` $name$($\overline{ident_i : bTy_i}^{\,i}$) : `eff` bTy := e	Core procedure definition

The result of elaborating a C program includes a set of Core declarations together with the name of the startup (main) function; a set of names, core types, and allocation/initialisation expression for C objects with static storage duration; definitions of implementation-defined constants (some of which are Core functions); and a library of Core utility functions and procedures used by the elaboration.

Figure 2.1: Summary of relevant aspects of the Core syntax, heavily based upon [44, Figure 2]. The grey text indicates features not addressed in this report. Here, *mem-order* is as defined by the ISO standard, *ctype* ranges over representations of C type expressions, *label* ranges over C labels. The semantics of `Ivmin`, `Ivmax`, `is_signed`, etc. are implementation defined. This report only address the *ptrop* `PtrValidForDeref`.

```
 1  proc main (): eff loaded integer :=
 2    let strong x: pointer =
 3      create(Ivalignof("signed␣int"), "signed␣int") in
 4    let strong a_74: loaded integer =
 5      {-#6.5.6#-}
 6      (let weak (a_75: loaded integer, a_76: loaded integer) =
 7        unseq(pure(Specified(42)), pure(Specified(1))) in
 8      pure(case (a_75, a_76) of
 9        | (Specified(a_77: integer), Specified(a_78: integer)) =>
10            Specified(catch_exceptional_condition("signed␣int",
11            conv_int("signed␣int", a_77) + conv_int("signed␣int", a_78)))
12        | _: (loaded integer,loaded integer) =>
13            undef(<<UB036_exceptional_condition>>)
14      end)) in
15    store("signed␣int", x, conv_loaded_int("signed␣int", a_74)) ;
16    let strong a_80: loaded integer =
17      bound[0] (let weak a_79: pointer = pure(x) in
18                load("signed␣int", a_79)) in
19    kill(x) ;
20    run ret_71(conv_loaded_int("signed␣int", a_80)) ;
21    kill(x) ;
22    (save ret_71: loaded integer (a_72: loaded integer:= Specified(0)) in
23      pure(a_72))
```

Figure 2.2: Elaboration of `int main(void){int x = 42 + 1; return 0;}` into Core

```
int main(void) {
  int x = 42 + 1;
  return x;
}
```

Memory actions The Core language makes many aspects of C programs explicit, with interaction with the memory object and concurrency models factored via primitive language constructs (*memory actions*). The lifetime of `x` is made explicit with the `create` and `kill` memory actions, and the semantics of memory accesses such as `store` (line 15) and `load` (line 18) are determined by a linked memory model. The alignment constraint of `x` is expressed as the implementation-defined `Ivalignof`.

Evaluation order The unsequenced evaluation order of arguments to `(+)` is expressed using `unseq` (line 7), with `let strong/weak` expressing other aspects of C's loose evaluation order.

Arithmetic and integer conversions Arithmetic in Core is defined over the mathematical integers. The ISO standard [37, §6.3.1] describes rules for *integer conversions*, for converting between different bit widths and signedness. For example, converting between unsigned integer types of different sizes is well-defined (by essentially taking the remainder), but a signed integer is not always *representable* in a smaller signed type. Integer conversions are performed using the Core function `conv_int` (see Appendix A), which returns a Core (mathematical) integer. C arithmetic operations implicitly perform integer conversions on the arguments to make them the same type (line 11). The `catch_exceptional_condition` function (line 10) checks that the sum is representable as a signed integer, returning an `undef` expression otherwise (indicating, for instance, signed integer overflow).

```
fun catch_exceptional_condition(ty: ctype, n: integer) : integer :=
  if is_representable(n, ty) then
    n
  else
    undef(...)
```

Undefined behaviour Undefined behaviour can arise from undefined arithmetic operations (e.g. signed integer overflow or division-by-zero) or memory actions (e.g. unsafe memory accesses). For the former, the elaboration to Core introduces an explicit test, `undef` (line 13). The latter depends on the memory object model, and are factored using Core expressions such as `ptrop(PtrValidForDeref,ptr)`.

Loops and gotos C control-flow constructs such as loops, `break`, and `return` are elaborated using Core *labels*, `run` and `save`, a goto-like construct. Essentially, `save` $label(\overline{ident_i : ctype_i := pe_i}^i)$ `in` e evaluates to $e[\overline{pe_i/ident_i}^i]$ (substitution of pe_i for each occurrence of $ident_i$ in e) and `run` $label(\overline{pe_i}^i)$ "jumps" to the corresponding `save` with pe_i substituted for each bound identifier. For example, C functions are elaborated to Core procedures ending with `save ret`, and a C "return" to a `run ret` jumping to the aforementioned save. The substitution handles the lifetime of block-scoped variables in C. For example, when jumping into the middle of a block such as {`int x; label:  ...`}, the lifetime of local objects start. Annotating `runs/saves` with these objects allows the dynamics of a `run` to do a `create` for objects in scope at the target `save` but not at the `run`, and a `kill` for the converse [44].

Specified and unspecified values In the ISO standard, uninitialised values are *indeterminate values*, which are either *unspecified values* (valid values of the relevant type with no requirement on which value is chosen [37, §3.19.3]) or *trap representations* (object representations that need not represent a value of the object type [37, §3.19.4]). It is undefined behaviour to load trap representations, and sometimes undefined to load unspecified values.

As examined in a survey by Memarian et al. [45] and discussed in a WG14 defect report [58], the semantics of reading from an uninitialized variable is unclear. Possible interpretations could be:

1. undefined behaviour
2. making the result of any expression involving the read value unpredictable
3. an arbitrary and unstable value (with a different value on another read)
4. an arbitrary but stable value (with the same value if read again)

The Cerberus semantics mostly[1] adopts (2) by extending values with a token "unspecified value" introduced by reads of uninitialised values. Thus for every C object type oTy, there is a Core `loaded` oTy type (e.g. `loaded_integer`):

```
loaded oTy :=
  | Specified of oTy
  | Unspecified
```

In most cases, if one or more arguments of an expression are unspecified, the result is unspecified. However if there exists a value giving undefined behaviour (e.g. dividing by an unspecified value), the result is undefined. These semantics are made explicit in the elaboration to Core.

2.3 SMT-based bounded model checking

Model checking is an automatic formal verification technique for finite-state systems, with both the system and specification expressed as precise logical formulae, based on exhaustively "traversing" states to determine whether the specification holds. Verification of a safety property amounts to checking whether a given set of states is reachable [25]. Bounded model checking (BMC) with the aid of satisfiability (SAT) solving [16, 17] is a falsification-based approach successfully used for verification of hardware designs and embedded systems [31]. BMC expresses the verification problem as a propositional formula satisfiable iff there exists a counterexample path of a given length. As loops and unbounded data structures may give rise to infinite-state systems, BMC reduces programs to finite-state systems by, for instance, restricting the depth of recursive function calls and loops.

BMC has essentially two steps. First, the sequential behaviour of the program over a finite number of steps is encoded as a logical formula. Second, the formula is given to a decision procedure (e.g. a SAT solver) to obtain a satisfying assignment or a proof that none exists. The logical formula represents a counterexample trace, such that satisfiability corresponds to a concrete counterexample proving a correctness property is violated.

Consider the following Core program:

[1]Reading an uninitialised object is undefined for types (e.g. `_Bool`) implementation-defined to have trap representations.

```
1  fun foo() : integer :=
2    let x : integer = 5 in
3    if x > 0 then
4      undef(...)
5    else
6      x
```

To verify `undef` can not be reached in any execution, we want to assert $\neg(x > 0)$. From line 2, we have the constraint $x = 5$. As BMC is a falsification approach, we construct a formula that is satisfiable if any safety condition (in this case, just $\neg(x > 0)$) fails to hold. Thus, we consider the negation of the conjunction of the verification conditions. The resulting formula passed to an SAT/SMT solver would represent

$$(x = 5) \wedge \neg(\neg(x > 0))$$

This is satisfiable; the program has undefined behaviour.

2.3.1 SAT and SMT solvers

SAT solvers reason about propositional logic[2], determining if there exists an *assignment* of the variables V in a propositional formula F, that is a map $\sigma : V \to \{0, 1\}$, such that F evaluates to 1 under σ. If such an assignment exists, the formula is *satisfiable*.

The expression $(x = 5) \wedge (\neg(\neg(x > 0))$ is not directly expressible in propositional logic, which does not have the notion of integers nor predicate symbols such as $(=)$ or $(>)$. This formula can naturally be expressed in first-order logic, which extends propositional logic to allow reasoning about members of a non-empty universe, containing quantifiers ($\forall$ and $\exists$), predicates, and functions in addition to the usual variables. We are interested not in general first-order satisfiability of the above expression, which allows non-standard interpretations of $(>)$ or $(=)$, but rather in satisfiability with respect to the *background theory* of integer arithmetic where $(>)$ and $(=)$ are respectively the usual ordering and equivalence relation over integers. Background theories fix an interpretation of certain predicate and function symbols.

Satisfiability modulo theories, or SMT, is the problem of deciding whether a first-order logic formula is satisfiable with respect to some background theory. SMT solvers often work with a many-sorted first-order logic, in which variables are associated with a sort (such as `int` or `bool`), and enable satisfiability problems to be expressed more naturally at a higher level of abstraction such as that of integers and bit-vectors (finite-sized Boolean arrays).

While SMT solvers can be considered front-ends to SAT solvers (e.g. an n-dimensional bit-vector can be translated into n propositional variables), the more

[2]Recall in propositional logic, logical expressions are defined over Boolean variables and logical connectives ($\wedge$, $\vee$, $\neg$).

compact representation enables specialized inference methods and optimizations in the decision procedure [22]. For example, while the satisfiability of first-order formulae built with $\{0, 1, +, *, <, =\}$ is decidable in the theory of *real* numbers (non-linear integer arithmetic is not decidable), faster solvers can be implemented by restricting the language to quantifier-free formulas [8, 21]. SMT solvers can identify occurrences of these “sub-logics” and apply more specialized and efficient techniques. Armando et al. [3] demonstrated that bounded model checking with SMT solvers instead of SAT solvers resulted in more compact formulae and better scaling. We refer to [9] for a more formal introduction to SMT.

Notation In this report, we represent SMT expressions either in mathematical notation or in the *SMT-LIB* [8] language as S-expressions. The SMT-LIB format is designed to be a common language for SMT solvers. In SMT-LIB syntax, the expression $(x = 5) \wedge \neg(\neg(x > 0))$ would be represented as

```
(declare-fun x () Int)
(assert (= (x 5)))
(assert (not (not (> x 0))))
```

Functions are declared using `declare-fun`, with constants such as `x` declared as 0-place function symbols. The integer sort `Int` is built in, and the interpretation of `(>)` and `(=)` are defined by SMT-LIB as is standard for the theory of integers. This is clearly satisfiable with x interpreted as the integer 5, or

```
(define-fun x () Int 5)
```

We introduce SMT-LIB notation as necessary throughout the report.

Chapter 3

Model checking sequential Core programs

This chapter discusses the bounded model checking of a sequential fragment of Core (highlighted in Figure 2.1 on Page 8) corresponding to C constructs expressible with an "abstract" memory model, in which variables are modifiable only using direct assignments or via pointer access if the address was obtained using the address-of (`&`) operator. We do not address arbitrary pointer arithmetic, pointer type-casting, unions, or floating points. Non-sequential constructs such as concurrency and unsequenced evaluation order are addressed in Chapter 4.

We first provide an overview of the translation from C to SMT problems, followed by more detailed discussion of the formulation. We aim for a *compositional* translation from Core expressions to SMT expressions based on recursing over the Core AST.

3.1 Overview

We reduce the model checking problem to an SMT problem. An overview of the reduction (recall Figure 1.1) is as follows:

1. **Translation of C to Core**: The Cerberus semantics is expressed through elaboration to a Core language [44], making explicit aspects of C such as undefined behaviour and evaluation order.
2. **Core-to-Core rewrite**: Implementation-dependent Core expressions are rewritten based on a provided C implementation. We uniquely rename Core identifiers (analogous to single static assignment, or SSA) and inline pure function calls to a specified bound. We impose a sequential order on `unseq`(...) expressions as a Core-to-Core rewrite, fixing an evaluation order (this restriction is removed in Chapter 4).

3. **Generate Z3 assertions**: We recurse over the Core AST to *compositionally* generate the following types of logical assertions:

   – **Syntactic constraints**: Core "let" expressions of the form

   $$\texttt{let}\ pat = e\ \texttt{in}\ \ldots$$

   generate equality constraints between the SMT representation of *pat* and *e*. We create fresh SMT variables for Core identifiers in *pat* and bind them appropriately to *e*.

   – **Verification conditions**: Verification conditions (VCs) represent safety properties such that an erroneous state (i.e. `undef` and `error`) is not reached. These correspond to a set of guards for control flow resulting in errors or undefined behaviour (e.g. for `if cond then undef else...`, a VC would be ¬`cond`).

   – **Memory constraints**: The memory constraints constrain the values corresponding to loads and stores, the semantics of which are dependent on the memory model. This chapter implements an abstract memory model with sequential memory accesses. Chapter 4 discusses the C11 concurrency model.

4. **Solving the SMT problem**: Let C be the set of memory and syntactic constraints, and V the set of VCs. We pass the following SMT problem to an SMT solver:

   $$\bigwedge_{c \in C} c \wedge \neg \bigwedge_{v \in V} v$$

   The memory and syntactic constraints represent *assumptions* and are designed such that $\bigwedge_{c \in C}$ is satisfiable. The expression is satisfiable iff any of the VCs fail to hold under the assumptions C (the program is erroneous) and unsatisfiable otherwise.

The Cerberus semantics are formally specified in Lem [48], which generates executable OCaml. We implement our model checker in OCaml and use the Z3 SMT solver [32], which has an OCaml API.

3.2 Example: Core program to SMT problem

Figure 3.1 presents a Core program asserting (in a normal form) $1 + 1 \geq 2$ along with a corresponding SMT problem.

Core-to-Core rewrite Every Core variable (and expression) has a type. We represent every Core variable (e.g. `x`) as a well-sorted SMT constant, such that in a given execution every SMT constant has one value (analogous to SSA form). Thus, we rename Core identifier `z` to `z1` in the first branch (line 5) and `z2` in the second (lines 7-9).

```
fun main() : integer :=
  let x : integer = 1 in
  let y : integer = x + x in
  if y >= 2 then
    let z : integer = 0 in z
  else
    let z : integer =
      error("assert␣failure", ...)
    in z
```

```
(declare-fun |x| () Int)
(declare-fun |y| () Int)
(declare-fun |z1| () Int)
(declare-fun |z2| () Int)
(declare-fun error!1 () Int)
(assert (= |x| 1))
(assert (= |y| (+ |x| |x|)))
(assert (= |z1| 0))
(assert (= |z2| error!1))
; verification condition
(assert (not (>= |y| 2)))
```

Figure 3.1: An example SMT problem (right) in SMT-LIB notation for a simple Core pure function (left). This SMT problem is not satisfiable. No erroneous behaviour occurs.

Syntactic constraints For each Core `let` *pat* = *e* `in` ... expression, we declare fresh SMT constants for all Core identifiers in *pat* and generate equality constraints for the SMT representation of expression *e*. For instance, Core variables `x`, `y`, `z1`, and `z2` of type `integer` are declared as 0-place function symbols (constants) `|x|`, `|y|`, `|z1|`, and `|z2|` of sort `Int`[1].

For each Core expression (such as `1`, `x + x`, and `let z1 : integer = 0 in z1`), we recursively compute an SMT representation (correspondingly `1`, `|x| + |x|`, and `|z1|` of sort `Int`). This allows us to express syntactic equality constraints such as `|x| = 1` and `|y| = |x| + |x|`. We represent `undef` and `error` expressions as fresh uninterpreted SMT constants of the appropriate sort (e.g. `error!1`). This allows expressions such as `undef(...) + 1` to be *compositionally* translated to SMT expressions.

Verification conditions We generate VCs expressing that `undef` and `error` expressions are unreachable. In this example, an `error` occurs iff `not (y >= 2)`, which is not satisfiable under the assumptions of the syntactic constraints.

Memory constraints Pure expressions do not have memory constraints. The distinction between pure and effectful expressions simplifies the translation to SMT problems.

3.3 Core-to-Core rewrites

We implement Core-to-Core rewrites primarily to incorporate implementation-defined behaviour, but also to simplify model checking by ensuring variables are in SSA form and inlining pure functions. This makes the program being checked

[1]Or `Bit-Vector`

more explicit. In this chapter, Core-to-Core rewrites are additionally used to sequentialise the Core program by ordering `unseq` expressions. This demonstrates how the semantics of C programs is decoupled from the model checking procedure, with altering the evaluation order being a Core-to-Core rewrite independent of the translation from Core to SMT.

The following program checks for signed integer overflow when incrementing $2^{15}-1$ by 1. There is undefined behaviour if 2^{15} is not representable as, or within the allowed range of, a "signed int" in the given C implementation.

```
fun is_representable (n: integer, ty: ctype): boolean :=
  Ivmin(ty) <= n /\ n <= Ivmax(ty)

fun catch_exceptional_condition (ty: ctype, n: integer) : integer :=
  if is_representable(n, ty) then
    n
  else
    undef(<<UB036_exceptional_condition>>)

fun main() : integer :=
  let ty : ctype = "signed␣int" in
  let large_int : integer = 32767 in -- (2^15 - 1)
  catch_exceptional_condition(ty, large_int + 1)
```

Inlining pure functions We inline pure functions to a specified depth, with function calls exceeding the depth replaced with an *unwind assertion* (a Core `error` expression that, if reachable, indicates the inlining depth was insufficient). Identifiers in inlined functions are freshly renamed such that symbols are uniquely bound for each function call. The result of inlining functions in the above is below:

```
fun main() : integer :=
  let ty1 : ctype = "signed␣int" in
  let large_int : integer = 32767 in
  let (ty2 : ctype, n1 : integer) = (ty1, large_int + 1) in
  if Ivmin(ty2) <= n1 /\ n1 <= Ivmax(ty2) then
    n1
  else
    undef(<<UB036_exceptional_condition>>)
```

Implementation-defined constants Implementation-defined constructs such as `Ivmax`(*ctype*), the maximum value of a C integer type, are rewritten as Core expressions based on a given implementation.

For example, assuming an implementation with two's complement representation and 16-bit ints, `Ivmax`(*ty*) would be rewritten as a sequence of if-then-else statements (terminating with an error for invalid arguments):

```
    if ty == "signed_int"   then 32767 else --(2^15-1)
    if ty == "unsigned_int" then 65535 else --(2^16-1)
    ...
    error(...)
```

The correctness of a Core program depends on the C implementation. In the above program, there is undefined behaviour (signed integer overflow) assuming 16-bit integers, but not with 32-bit integers.

Instead of replacing implementation-defined constructs such as `Ivmax` with Core expressions based on an implementation, we could declare uninterpreted SMT functions accompanied with implementation-defined constraints. For example, assuming the `Ctype` sort were appropriately defined with `signedInt` and `unsignedInt` of `Ctype`, we could define `Ivmax` as follows:

```
    (declare-fun Ivmax (Ctype) Int)
    (assert (= (Ivmax signedInt) 32767))
    (assert (= (Ivmax unsignedInt) 65535))
    (assert (= (Ivmax ...) ...))
```

However, some implementation-defined constructs are represented as Core pure functions and constraining the SMT problem such that the solver computes a desired interpretation of a function for a large domain is potentially expensive or complex to express efficiently. For consistent treatment of implementation-defined behaviour and pure functions, and also to make the C semantics explicit in the Core program, we elect to “inline” implementation-defined constructs and pure functions.

3.4 Representation of Core expressions in SMT

Our model checker is designed to be compositional, with the procedure based on recursing over the AST. For example, for a Core expression such as pe_1 *binop* pe_2, we compute for each subexpression pe_1 and pe_2 a well-sorted SMT representation (along with VCs and syntactic/memory constraints). The representation of pe_1 *binop* pe_2 is then the application of an SMT function representing the binary operator to the SMT representations of pe_1 and pe_2.

3.4.1 Core types to SMT sorts

SMT solvers reason with a many-sorted first-order logic—every SMT variable and expression has a *sort*. To express Core expressions in SMT, we first represent Core types as SMT sorts.

We define a function

```
mk_sort :  bTy -> Z3Sort
```

taking a Core base type and returning a Z3 sort, summarized in Table 3.1.

bTy	mk_sort(bTy)
unit	| Unit
boolean	Boolean (built-in)
ctype	Ctype: variant type of C type expressions
integer	Int or fixed-width BitVector (built-in)
pointer	| Addr of alloc_id * index
loaded integer	| Specified of mk_sort(integer)
	| Unspecified of Ctype
loaded pointer	| Specified of mk_sort(pointer)
	| Unspecified of Ctype

Table 3.1: SMT sorts corresponding to Core types. The "`| Constructor of ty`" notation is used to construct algebraic datatypes (more specifically, variants). For example, a `loaded integer` has two constructors: `Specified` which takes a value of sort `Int` or `BitVector`, and `Unspecified` which take a value of sort `Ctype`. `Ctype` is the sort containing values of C type expressions. For example, `void`, `signed int`, and `signed_int[5]` (for an array of size 5 storing signed integers) are Core values with Core type `ctype`, translated to SMT values with sort `Ctype`.

Z3 supports algebraic datatypes. For example, `loaded integer` is represented as a datatype with two constructors, `Specified` and `Unspecified`, and we define `Ctype` (with values representing C types such as "signed int" and "void") as an algebraic datatype. A Core `integer` corresponds to mathematical integers and would most accurately be represented with sort `Int` in Z3. However, non-linear integer arithmetic (e.g. multiplication and division) is undecidable and there is no/limited support for exponentiation and bitshifting. In contrast to the theory of integers, satisfiability of the theory of bit-vectors is decidable even with multiplication [10]. Furthermore, bit-wise operations are efficiently supported. On the other hand, avoiding overflow or underflow in Core may involve additional checks or use of unnecessarily large bit-vectors. Our implementation supports both representations, toggled by a flag.

Finally, we represent Core pointers with their associated "address" (discussed in § 3.7).

3.4.2 Compositional representation of Core expressions

We recursively compute the Z3 representation of Core expressions with the function

```
INTERP : expr -> Z3Expr
```

as presented in Table 3.2.

Each Core identifier is mapped to a Z3 variable maintained in a *symbol table*, such that `INTERP`(*ident*) either creates a fresh Z3 variable or looks up the bound Z3 representation. The `undef` and `error` expressions are represented as fresh Z3 constants of the appropriate sort, enabling compositional translation of expressions such as `undef` $+ 1$.

Core constructors such as `Specified` are represented as Z3 functions from constructor arguments to Z3 expressions of the constructed sort. For instance, the Z3 function `LoadedInt_Specified` takes an `Int` (or `BitVector`) and returns a value of sort `LoadedInteger`. We represent `case` expressions as a sequence of if-then-else expressions. Let expressions of the form `let pat = e1 in e2` are represented as `INTERP(e2)` (with symbols in `pat` or `e1` added to the symbol table beforehand by computing `INTERP(e1)` and binding symbols from `pat`). `nd`$(e_1, \ldots, e_n)$ represents non-deterministic choice of execution of exactly one of the e_i expressions. We generate n choice variables `nd_1`$, \ldots,$ `nd_n` such that `XOR(nd_1,...,nd_n)` and represent the non-deterministic expression as a sequence of if-then-else statements branching on the choice variables. We discuss representation of memory-related expressions, function calls, and runs/saves in later sections.

3.5 Syntactic constraints

"Let" expressions give rise to syntactic constraints, or assumptions, on variable bindings. The set of syntactic constraints is always satisfiable. The following program expressing the division of loaded integers `x` and `y` is undefined if `x` or `y` is unspecified or `y` is `0`.

```
 1  fun main() : integer :=
 2    let (x : loaded integer, y : loaded integer) =
 3      (Specified(42), Specified(2)) in
 4    case (x, y) of
 5      | (Specified (a1 : integer), Specified(a2 : integer)) =>
 6         if a2 = 0 then
 7           undef(...)
 8         else
 9           a1 / a2
10      | _ : (loaded integer, loaded integer) =>
11         undef(...)
12    end
```

Listing 3.1: Pattern matching example

We generate syntactic constraints (individually for tuple elements) for `let` $pat = pe_1$ `in` $\ldots$ by computing the representation of pe_1 and generating equality expressions using

```
mk_eq :  Z3Expr -> pattern -> Z3Expr
```

such that (for lines 2-3)

`mk_eq(Tuple(|x|,|y|), (Specified(42), Specified(2)))`
$\rightarrow (|\mathtt{x}| = \mathtt{LoadedInt\_Specified}(42)) \wedge (|\mathtt{y}| = \mathtt{LoadedInt\_Specified}(2))$

In the process, we declare fresh SMT constants for the identifiers in *pat*:

`pexpr`	`INTERP(pexpr)`
$ident$	`if` $ident \notin$ `sym_table then sym_table`$[ident] =$ `mk_fresh(ty);`
	`sym_table`$[ident]$
$value$	`mk_val`$(value)$
<$impl\text{-}const$>	N/A (inlined)
`undef(...)`	`mk_fresh(ty)`
`error(...)`	`mk_fresh(ty)`
`ctor`$(pe_1, \ldots, pe_n)$	`mk_ctor(ctor)`(`INTERP`$(pe_1), \ldots,$ `INTERP`(pe_n))
`case` pe `with` $\overline{\mid pat_i \Rightarrow pe_i}^{i}$ `end`	`mk_ite`(`MATCH`(pe, pat_1), `INTERP`(pe_1),
	`mk_ite`(`MATCH`(pe, pat_2), `INTERP`$(pe_2), \ldots)$
`array_shift`$(pe_1, ctype, pe_2)$	`mk_shift`(`INTERP`(pe_1), `INTERP`(pe_2))
`not`(pe)	`mk_not`(`INTERP`(pe))
$pe_1\ binop\ pe_2$	`mk_binop`$(binop,$ `INTERP`(pe_1), `INTERP`(pe_2))
$name(pe_1, \ldots, pe_n)$	N/A(inlined)
`let` $pat = pe_1$ `in` pe_2	`INTERP`(pe_2)
`if` pe `then` pe_1 `else` pe_2	`mk_ite`(`INTERP`(pe), `INTERP`(pe_1), `INTERP`(pe_2))
`is_signed`(pe)	N/A(inlined)
`...`	

`expr`	`INTERP(expr)`
`pure`(pe)	`INTERP`(pe)
`create`(pe_1, pe_2)	`mk_addr`(`INTERP`(pe_2))
`store`$(pe_1, pe_2, pe, mem - order)$	`mk_unit()`
`ptrop`(`PtrValidForDeref`, pe_1)	$0 \leq$ `index(addr_of(INTERP`(pe_1)`)))` $<$ `alloc_size(alloc_of(INTERP`(pe_1)`)))`
`load`$(pe_1, pe_2, mem - order)$	`mk_fresh(bty_of_ctype`(pe_1)`)`
`case` pe `with` $\overline{\mid pat_i \Rightarrow e_i}^{i}$	`mk_ite`(`MATCH`(pe, pat_1), `INTERP`(e_1),
	`mk_ite`(`MATCH`(pe, pat_2), `INTERP`$(e_2), \ldots)$
`if` pe `then` e_1 `else` e_2	`mk_ite`(`INTERP`(pe), `INTERP`(e_1), `INTERP`(e_2))
`skip`	`mk_unit()`
`ccall`$(pe, pe_1, \ldots, pe_n)$	`ret_value`(`body`$(pe, pe_1, \ldots, pe_n)$)
`unseq`$(e_1, \ldots, e_n)$	(`INTERP`$(e_1), \ldots,$ `INTERP`(e_n))
`let weak` $pat = e_1$ `in` e_2	`INTERP`(e_2)
`let strong` $pat = e_1$ `in` e_2	`INTERP`(e_2)
`nd`$(e_1, \ldots, e_n)$	`mk_ite`(`nd_1`, `INTERP`(e_1),
	`mk_ite`(`nd_2`, `INTERP`$(e_2), \ldots$
`save` $label(\overline{ident_i := pe_i}^{i})$ `in` e	`INTERP`$(e[\overline{pe_i/ident_i}^{i}])$
`run` $label(\overline{ident_i := pe_i}^{i})$	`INTERP`(`find_save`$(label)[\overline{pe_i/ident_i}^{i}]$)
`par`$(e_1, \ldots, e_n)$	`mk_unit()`

Table 3.2: A simplified representation of the mapping of Core expressions into SMT. Here, `ty` is the Z3 sort of the expression. We omit side effects such as adding new patterns to the symbol table with "let" expressions. Type parameters to the functions are elided. For reasons of scope, in this report we omit object lifetimes and simplify `PtrValidForDeref` to a bounds check (effectively asserting that the address was created).

```
    (declare-fun |x| () LoadedInt)
    (declare-fun |y| () LoadedInt)
```

3.5.1 Pattern matching

Defining algebraic datatypes, such as `LoadedInteger`, in Z3 results in the definition of *recognizer* function, such as `isLoadedInt_Specified` which takes a `LoadedInteger` and returns `true` iff the argument was constructed using `LoadedInt_Specified`. We use recognizer function to pattern match in `case` expressions.

We define the function

```
MATCH : Z3Expr * pat -> Z3Expr
```

taking a Z3 expression and Core pattern, and returning an SMT Boolean predicating whether the expression matches the pattern. For example:

```
MATCH(|x|, Specified (a1 :  integer)) = isLoadedInt_Specified(|x|)
```

Matching the wildcard pattern or a Core identifier (guaranteed fresh by the SSA transformation) is always true.

In `case` pe `with` $\overline{|pat_i \Rightarrow pe_i}^{i}$ `end`, the k^{th} case is selected if the k^{th} pattern is matched ($\texttt{MATCH}(\texttt{INTERP}(pe), pat_k)$) and all previous patterns pat_i for $i < k$ are not matched. That is

$$\texttt{select}_k = \texttt{MATCH}(\texttt{INTERP}(pe), pat_k) \land \neg \bigvee_{i<k} \texttt{MATCH}(\texttt{INTERP}(pe), pat_i)$$

For each case, we generate a guarded equality constraint binding the pattern to pe:

$$\texttt{select}_k \Rightarrow \texttt{mk\_eq}(pe, pat_k)$$

In Listing 3.1, the pattern matching on line 5 gives

$$\texttt{isLoadedInt\_Specified(|x|)} \land \texttt{isLoadedInt\_Specified}(|y|)$$
$$\Rightarrow \texttt{|a1| = getSpecified(|x|)} \land \texttt{|a2| = getSpecified(|y|)}$$

Here, `getSpecified` is an *accessor* function defined by Z3 upon construction of the `LoadedInt` datatype, retrieving the argument passed to the `LoadedInt_Specified` constructor function.

3.5.2 Summary table

We define a function

```
ASSUME : expr -> Z3Expr
```

generating a Z3 Boolean representing the syntactic constraints derived from a Core expression (summarized in Table 3.3).

These constraints are recursively generated from subexpressions. We carefully ensure they do not need to be guarded by control flow (apart from `case` expressions) by appropriately guarding VCs and memory constraints; syntactic constraints represent *definitions* or bindings of SMT variables, and variables are guaranteed to be defined at most once by the SSA Core-to-Core rewrite. Thus although $\texttt{ASSUME}(\texttt{if}\ pe\ \texttt{then}\ pe_1\ \texttt{else}\ pe_2)$ would naturally be expressed as

$$\texttt{ASSUME}(pe) \wedge (\texttt{INTERP}(pe) \Rightarrow \texttt{ASSUME}(pe_1)) \wedge (\neg(\texttt{INTERP}(pe) \Rightarrow \texttt{INTERP}(pe_2)) \\ \Rightarrow \texttt{ASSUME}(pe_2)$$

the branch condition guards are not required. For non-deterministic sequencing `nd`, we assert exactly one of the (fresh) choice variables `nd_i` is true. Again, the generated constraints are not guarded by the choice variables. This simplifies the implementation of `ASSUME` as well as the SMT problem.

3.6 Verification conditions

Verification conditions represent safety conditions that must hold for program correctness. Encoding reachability of erroneous states (`undef` and `error`) corresponds to encoding control flow. In Listing 3.1 on page 20, an `undef` is reached if the first case is selected and `a2 = 0`, or if the second case is selected. Thus, the set of VCs corresponds to

$$\{\neg\left(\texttt{select}_1 \wedge (\texttt{|a2|} = 0)\right),\ \neg\texttt{select}_2\}$$

Figure 3.2 presents the SMT problem for Listing 3.1, consisting of the constraints along with the negation of the conjunction of the VCs.

The computation of verification conditions (using `VC : expr -> Z3Expr`) is summarized in Table 3.4. Observe that `VC(undef(...))` is false, such that

$$\texttt{VC(if cond then undef else 1)} = (\texttt{|cond|} \Rightarrow \perp) = \neg\texttt{|cond|}$$

which corresponds to asserting the first branch is not taken. The VCs for subexpressions of `case` expressions are guarded by whether or not the case was selected, and the VCs for subexpressions e_i of the non-deterministic expression $\texttt{nd}(e_1, \ldots, e_n)$ are guarded by choice variables. For effectful sequencing expressions $\texttt{let weak/strong}\ pat = e_1\ \texttt{in}\ e_2$, e_2 might not execute if e_1 contains a return or goto statement (a Core `run`). Thus the VCs for e_2 are guarded by whether the continuation of e_1 (e_2) is dropped (expressing e_2 is not reachable and hence $\texttt{VC}(e_2)$ does not need to hold). This is discussed in § 3.8.

Notice that we reduced the problem of generating safety conditions to guarding control flow without discussion of implementation-defined behaviour, C semantics,

pexpr	ASSUME(pexpr)
ident	$\top$
value	$\top$
<*impl-const*>	N/A (inlined)
`undef(...)`	$\top$
`error(...)`	$\top$
`ctor`$(pe_1, \ldots, pe_n)$	$\bigwedge_i$ `ASSUME`(pe_i)
`case` pe `with` $\overline{\vert pat_i \Rightarrow pe_i}^{i}$ `end`	$\bigwedge_i \big($`MATCH(INTERP`$(pe), pat_i) \wedge \neg \bigvee_{k<i}$ `MATCH(INTERP`$(pe), pat_k)\big)$ $\Rightarrow$ `mk_eq(INTERP`$(pe), pat_i)$
`array_shift`$(pe_1, ctype, pe_2)$	`ASSUME`$(pe_1) \wedge$ `ASSUME`(pe_2)
`not`(pe)	`ASSUME`(pe)
pe_1 *binop* pe_2	`ASSUME`$(pe_1) \wedge$ `ASSUME`(pe_2)
$name(pe_1, \ldots, pe_n)$	N/A (inlined)
`let` $pat = pe_1$ `in` pe_2	`mk_eq(INTERP`$(pe_1), pat) \wedge$ `ASSUME`$(pe_1) \wedge$ `ASSUME`(pe_2)
`if` pe `then` pe_1 `else` pe_2	`ASSUME`$(pe) \wedge$ `ASSUME`$(pe_1) \wedge$ `ASSUME`(pe_2)
`is_signed`(pe)	N/A (inlined)
...	
expr	ASSUME(expr)
`pure`(pe)	`ASSUME`(pe)
`create`(pe_1, pe_2)	*see* 3.7
`store`$(pe_1, pe_2, pe, mem - order)$	*see* 3.7
`load`$(pe_1, pe_2, mem - order)$	*see* 3.7
`ptrop(PtrValidForDeref`$, pe)$	`ASSUME`(pe)
`case` pe `with` $\overline{\vert pat_i \Rightarrow e_i}^{i}$	$\bigwedge_i \big($`MATCH(INTERP`$(pe), pat_i) \wedge \neg \bigvee_{k<i}$ `MATCH(INTERP`$(pe), pat_k)\big)$ $\Rightarrow$ `mk_eq(INTERP`$(pe), pat_i)$
`if` pe `then` e_1 `else` e_2	`ASSUME`$(pe) \wedge$ `ASSUME`$(pe_1) \wedge$ `ASSUME`(pe_2)
`skip`	$\top$
`ccall`$(pe, pe_1, \ldots, pe_n)$	`ASSUME(body`$(pe, pe_1, \ldots, pe_n)) \wedge \bigwedge_i$ `ASSUME`(pe_i)
`unseq`$(e_1, \ldots, e_n)$	$\bigwedge_i$ `ASSUME`(e_i)
`let weak` $pat = e_1$ `in` e_2	`mk_eq(INTERP`$(e_1), pat) \wedge$ `ASSUME`$(e_1) \wedge$ `ASSUME`(e_2)
`let strong` $pat = e_1$ `in` e_2	`mk_eq(INTERP`$(e_1), pat) \wedge$ `ASSUME`$(e_1) \wedge$ `ASSUME`(e_2)
`nd`$(e_1, \ldots, e_n)$	`XOR(nd_1,...,nd_n)` $\wedge \bigwedge_i$ `ASSUME`(e_i)
`save` $label(\overrightarrow{ident_i := pe_i}^{i})$ `in` e	`ASSUME`$(e[\overline{pe_i/ident_i}^{i}])$
`run` $label(\overrightarrow{ident_i := pe_i}^{i})$	`ASSUME(find_save`$(label)[\overline{pe_i/ident_i}^{i}])$
`par`$(e_1, \ldots, e_n)$	$\bigwedge_i$ `ASSUME`(e_i)

Table 3.3: Syntactic constraints (simplified) generated from Core expressions. Constraints from `case` expressions are guarded by pattern matching guards (constraints of the case branches are omitted). Although syntactic constraints for `if-then-else` expressions can be guarded by control flow, we carefully ensure this is unnecessary by appropriately guarding the verification conditions and memory constraints by control flow. Syntactic constraints represent definitions (e.g. of variables) and every variable is guaranteed to be defined at most once due to the SSA transformation; the syntactic constraints, on their own, are always satisfiable.

`pexpr`	`VC(pexpr)`
$ident$	$\top$
$value$	$\top$
<$impl\text{-}const$>	N/A (inlined)
$\texttt{undef}(\ldots)$	$\bot$
$\texttt{error}(\ldots)$	$\bot$
$\texttt{ctor}(pe_1, \ldots, pe_n)$	$\bigwedge_{i=1}^{n} \texttt{VC}(pe_i)$
$\texttt{case}\ pe\ \texttt{with}\ \overline{\vert pat_i \Rightarrow pe_i}^{i}\ \texttt{end}$	$\bigwedge_i \big(\big(\texttt{MATCH}(pe, pat_i) \land \neg \bigvee_{k<i} \texttt{MATCH}(pe, pat_k)\big) \Rightarrow \texttt{VC}(pe_i)\big)$
$\texttt{array\_shift}(pe_1, ctype, pe_2)$	$\texttt{VC}(pe_1) \land \texttt{VC}(pe_2)$
$\texttt{not}(pe)$	$\texttt{VC}(pe)$
pe_1 binop pe_2	$\texttt{VC}(pe_1) \land \texttt{VC}(pe_2)$
$name(pe_1, \ldots, pe_n)$	N/A(inlined)
$\texttt{let}\ pat = pe_1\ \texttt{in}\ pe_2$	$\texttt{VC}(pe_1) \land \texttt{VC}(pe_2)$
$\texttt{if}\ pe\ \texttt{then}\ pe_1\ \texttt{else}\ pe_2$	$\texttt{VC}(pe) \land (\texttt{INTERP}(pe) \Rightarrow \texttt{VC}(pe_1)) \land (\neg(\texttt{INTERP}(pe)) \Rightarrow \texttt{VC}(pe_2))$
$\texttt{is\_signed}(pe)$	N/A(inlined)
...	

`expr`	`VC(expr)`
$\texttt{pure}(pe)$	$\texttt{VC}(pe)$
$\texttt{create}(pe_1, pe_2)$	$\texttt{VC}(pe_1) \land \texttt{VC}(pe_2)$
$\texttt{store}(pe_1, pe_2, pe, mem - order)$	$\texttt{VC}(pe_1) \land \texttt{VC}(pe_2) \land \texttt{VC}(pe)$
$\texttt{load}(pe_1, pe_2, mem - order)$	$\texttt{VC}(pe_1) \land \texttt{VC}(pe_2)$
$\texttt{ptrop}(\texttt{PtrValidForDeref}, pe_1)$	$\texttt{VC}(pe_1)$
$\texttt{case}\ pe\ \texttt{with}\ \overline{\vert pat_i \Rightarrow e_i}^{i}$	$\bigwedge_i \big(\big(\texttt{MATCH}(pe, pat_i) \land \neg \bigvee_{k<i} \texttt{MATCH}(pe, pat_k)\big) \Rightarrow \texttt{VC}(pe_i)\big)$
$\texttt{if}\ pe\ \texttt{then}\ e_1\ \texttt{else}\ e_2$	$\texttt{VC}(pe) \land (\texttt{INTERP}(pe) \Rightarrow \texttt{VC}(pe_1)) \land (\neg(\texttt{INTERP}(pe)) \Rightarrow \texttt{VC}(pe_2))$
$\texttt{skip}$	$\top$
$\texttt{ccall}(pe, pe_1, \ldots, pe_n)$	$\texttt{VC}(\texttt{body}(pe, pe_1, \ldots, pe_n)) \land \bigwedge_i \texttt{VC}(pe_i)$
$\texttt{unseq}(e_1, \ldots, e_n)$	$\bigwedge_i \texttt{VC}(e_i)$
$\texttt{let weak}\ pat = e_1\ \texttt{in}\ e_2$	$\texttt{VC}(e_1) \land (\neg\texttt{drop\_cont}(e_1) \Rightarrow \texttt{VC}(e_2))$
$\texttt{let strong}\ pat = e_1\ \texttt{in}\ e_2$	$\texttt{VC}(e_1) \land (\neg\texttt{drop\_cont}(e_1) \Rightarrow \texttt{VC}(e_2))$
$\texttt{nd}(e_1, \ldots, e_n)$	$\bigwedge_i \texttt{nd\_i} \Rightarrow \texttt{VC}(e_i)$
$\texttt{save}\ label(\overrightarrow{ident_i := pe_i}^{i})\ \texttt{in}\ e$	$\texttt{VC}(e[\overline{pe_i/ident_i}])$
$\texttt{run}\ label(\overrightarrow{ident_i := pe_i}^{i})$	$\texttt{VC}(\texttt{find\_save}(label)[\overline{pe_i/ident_i}^{i}])$
$\texttt{par}(e_1, \ldots, e_n)$	$\bigwedge_i \texttt{VC}(e_i)$

Table 3.4: (Simplified conjunction of) verification conditions generated compositionally from Core expressions. Verification conditions express control flow paths in which erroneous behaviours occur.

```
(declare-fun |x| () LoadedInt)
(declare-fun |y| () LoadedInt)
(assert (= |x| (LoadedInt_Specified(42))))
(assert (= |y| (LoadedInt_Specified(2))))
; matching case 1
(assert (=> (and (isSpecified(|x|)) (isSpecified(|y|)))
            (and (= |a1| (getSpecified(|x|))
                 (= |a2| (getSpecified(|y|))))))
; matching case 2 (wildcard - no syntactic constraints)
(assert (=> (not (and (isSpecified(|x|)) (isSpecified(|y|))))
             true))
; verification conditions
(assert (not (and (not (and (isSpecified(|x|))
                             (isSpecified(|y|))
                             (= |a2| 0)))
                  (not (not (and (isSpecified(|x|))
                                  (isSpecified(|y|))))))))
```

Figure 3.2: SMT problem corresponding to Listing 3.1 on Page 20. Assuming the other assertions hold, the problem is satisfiable iff any of the verification conditions fail to hold ($\neg \bigwedge_{v \in VC} v$).

or the > 200 sources of undefined behaviour described in the ISO standard. The semantics of C are expressed in the elaboration to Core, such that the model checking procedure largely does not depend on the nuances of C semantics.

3.7 Memory constraints

Memory interactions such as `store` and `ptrop(PtrValidForDeref,...)` are primitive constructs in Core, the semantics of which are defined by a linked sequential memory layout or concurrency model. This section describes the interpretation of Core memory expressions by the model checker for a sequential model in which loads read the most recent stores to the same location.

3.7.1 Modelling program state

Programming languages define an interface to memory. However, the semantics of pointers and memory objects (the *memory object model*) as specified by the ISO standard is unclear. Analysis tools for imperative languages such as C must consider how to model program state.

In type-safe languages, state could be modelled with an *abstract* memory model as a map from (*object*, *field*) pairs to values, with aliasing arising only from two pointers of the same type pointing to the same location [30]. However, in C, values are not purely abstract (types only provide a means for *interpreting* memory): the

```
proc main() : eff loaded integer :=
  let strong x : pointer =
      create(Ivalignof("signed␣int"), "signed␣int") in
  store("signed␣int", x, pure(Specified(0));
  store("signed␣int", x, pure(Specified(1));
  let alias_x : pointer = pure(x) in
  load("signed␣int", alias_x)
```

Figure 3.3: Example Core program with memory accesses

standard allows manipulation of underlying representations with `char*` pointers and casts between pointer and integer types.

On the other extreme, state could be modelled as a finite map from addresses to bytes in a *concrete* model. While this is the case at runtime, the standard defines abstract notions of types, pointer provenance [44], and uninitialised values (for which the value can change without direct action of the program [58]).

Work by Memarian et al. [43, 44] contributes to developing a memory object model for de facto C. For reasons of scope, we focus instead on the C11 concurrency model and limit our support to a fragment of C allowing the assumption of an *abstract* memory model, in which variables are modified only using direct assignments or through pointers if the address was obtained using `(&)`. We do not support pointer typecasting or arbitrary pointer arithmetic.

3.7.2 Pointer sort

The Core program in Figure 3.3 creates a pointer `x`, writes to its address, and returns the value stored at `x`.

Each object declaration in C is translated to a `create` of a Core `pointer` that we associate with a unique *allocation identifier*. Each allocation is associated with a number of locations, one for types such as “signed int” and n for arrays or structs with n elements or members; creating a pointer to an array of five elements with allocation id $@_1$ results in 5 locations: $(@_1, 0), (@_1, 1), \ldots, (@_1, 4)$.

We represent locations in Z3 as tuples `(alloc_id, index)`. This representation allows array indexing and struct member accesses via pointer arithmetic on `index`. For instance, an `array_shift` of a pointer `ptr` by some (numerical) expression `pe` can be represented in Z3 as

```
(alloc_id(addr_of(INTERP(ptr))), index(addr_of(ptr)) + INTERP(pe))
```

Recalling the compositional approach to model checking, we compute the SMT representation of `ptr` and `pe` with `INTERP`. Here, `addr_of`, `alloc_id`, and `index` are SMT functions for accessing the address of a pointer, allocation id of an address, and index of an address respectively. As `pe` has type `integer`, its SMT representation has the corresponding sort, either `Int` or `BitVector`. To allow pointer arithmetic, `index` must have the same sort as `mk_sort(integer)`.

We do not support the creation of pointers from pointer arithmetic with pointers of different allocation identifiers, and do not allow a pointer derived from an array shift or pointer operation to have a different allocation id to that of its source(s). This does not conflict with the ISO standard and corresponds to the notion of "pointer provenance" [45, 57].

In the elaboration, pointer dereferences are guarded with a check for pointer validity:

```
if memop(PtrValidForDeref, ptr) then
   (* Pointer can be dereferenced *)
else
  pure(undef(...))
```

As we currently do not account for object lifetimes[2] due to scope, we define `memop(PtrValidForDeref, ptr)` to be true if `ptr` is not `NULL` and its index is inbounds, or:

```
0 <= index(addr_of(|ptr|)) < alloc_size(alloc_of(|ptr|))
```

This enables detection of buffer overflows from accessing out-of-bounds indices.

C's memory model is not abstract and thus a more fine-grained representation is needed to model arbitrary pointer arithmetic and typecasting. For simplicity, in the following, we assume a pointer or address is represented only with an allocation id.

3.7.3 Memory actions

Create(ty) Each dynamic instance of a `create` (in one-to-one correspondence with static instances after Core transformations) is associated with a fresh allocation id (in Z3, an integer or bit-vector). A Core `create` returns a Core pointer, thus the SMT representation of `create(...)` is a variable of pointer sort. When representing pointers only as allocation ids, from `let strong x : pointer = create(Ivalignof("signed int"), "signed int")` we might assert

```
(assert (= |x| (Ptr (Loc (0)))))
```

to represent $x \mapsto @_0$. We associate with each location a *sequence* of fresh assignment variables, such that $@_0^0$ represents the initial value at $@_0$, $@_0^1$ the next `store` to that location, etc. We "initialize" the initial value to an `Unspecified` value (of loaded object type) to represent *uninitialized values*:

[2]Object lifetimes are explicit with `create` and `kill` and can be represented as an SMT expression predicating whether a `kill` for the corresponding location was encountered in the control flow.

```
    (assert (= loc!0!0 (LoadedInt_Unspecified(signedInt))))
```

where `loc!0!0` represents $@_0^0$. We internally maintain a memory map, `mem`, mapping each location to an SMT expression representing its current value. Thus we have

$$\texttt{mem}[@_0] = @_0^0$$

To implement structs and arrays, a `create(...)` of struct or array type, such as

```
create(Ivalignof("signed_int[5]"), "signed_int[5]")
```

results in n creates with the same allocation id (where n is the array size or number of fields in the struct) and indices from 0 to $n-1$, along with an assertion that

$$\texttt{alloc\_size}(\texttt{new\_alloc\_id}) = n$$

Locations are initialized as above, and a pointer to the location at index 0 is returned.

Store(ptr,value) Without alias analysis, the location pointed to by a Core pointer symbol is unknown. The naïve solution to model a store is to generate equations, one for each created location, of the form

$$(\texttt{addr\_of(|ptr|)} = @_i) \Rightarrow \texttt{next}(@_i) = \texttt{value}$$
$$\wedge(\texttt{addr\_of(|ptr|)} \neq @_i) \Rightarrow \texttt{next}(@_i) = \texttt{mem}(@_i)$$

and update `mem` such that

$$@_i \xmapsto{\texttt{mem}} \texttt{next}(@_i)$$

where $\texttt{next}(@_i)$ is a fresh sequence variable. This defines $\texttt{next}(@_i)$ to be the new value if `ptr` points to location i, and retains the old value otherwise.

We reduce the number of generated equations by approximating, for each Core pointer, the set of locations it may alias with in § 3.7.5.

Load(ptr) In the sequential model, a load reads the previous store at the same location. We create a Z3 constant, say `load`, for the loaded value, and generate a set of equations

$$(\texttt{addr\_of(|ptr|)} = @_i) \Rightarrow \texttt{load} = \texttt{mem}(@_i)$$

which define the value of `load` based on the current values in memory. In the concurrent model, the notion of "previous store" does not necessarily exist and the "reads-from" relation is more complex.

3.7.4 Branching control flow

The contents of `mem` after effectful, branching expressions (e.g. `case`, `if-then-else`, and `nd`) must be guarded by control flow. Suppose $\texttt{mem}_k$ is the memory map resulting from model-checking the expression in the k^{th} branch. To compute the new memory map `mem`, we declare for every location $@_i$ potentially updated in the branches

$$\texttt{mem}[@_i] = \texttt{next}(@_i)$$

for fresh $\texttt{next}(@_i)$ and assert

$$\texttt{guard}_k \Rightarrow \texttt{next}(@_i) = \texttt{mem}_k[@_i]$$

where $\texttt{guard}_k$ is a Boolean predicating whether the k^{th} branch was taken (as when computing VCs). Unchanged locations retain their old values. We could represent $\texttt{mem}[@_i]$ as a sequence of if-then-else expressions instead of creating fresh variables, however our chosen representation simplifies SMT expressions with memory values (which are now just SMT constants).

3.7.5 Optimisation: alias analysis

Instead of generating equations for every location, we perform alias analysis to approximate the set of "addresses" each Core pointer might represent, and which addresses locations storing pointers might point to.

We maintain a map from Core pointer symbol to set of addresses

$$\texttt{alias : symbol} \mapsto \texttt{addr set}$$

and compute for each expression (of Core pointer type) a set of addresses it may represent. Thus an expression of the form `let x : pointer = create(...)` maps `x` to the address returned by `create`, and an expression of the form `let x2 : pointer = x` ... gives

```
alias[x2] = alias[x]
```

We maintain a `points_to` map

$$\texttt{points\_to : addr} \mapsto \texttt{addr set}$$

from addresses containing pointer values to the set of addresses the stored pointer may "point to". This increases the precision of the analysis when accessing the location from a pointer loaded from memory.

In Figure 3.4, the two creates (lines 3 and 7) with respective allocations $@_0$ and $@_1$ give

$$\texttt{alias[x]} = \{@_0\}$$
$$\texttt{alias[px]} = \{@_1\}$$

```
 1  proc main() : eff loaded integer :=
 2    let strong x : pointer =
 3      create(Ivalignof("signed␣int", "signed␣int")) in
 4    store("signed␣int", x, Specified(0));
 5
 6    let strong px : pointer =
 7      create(Ivalignof("signed␣int*", "signed␣int*")) in
 8    store("signed␣int*", px, Specified(x));
 9
10    let strong y : loaded pointer =
11        load("signed␣int*", px) in
12    case y of
13      | Specified (z : pointer) => load("signed␣int", z)
14      | _ => error(...)
15    end
```

Figure 3.4: Example Core program demonstrating aliasing and C pointers. Here `y` and `z` alias with `x`.

and storing a pointer on line 8 gives

$$\forall @_i \in \texttt{alias[px]}.\,(\texttt{points\_to}[@_i] = \texttt{points\_to}[@_i] \cup \texttt{alias[x]})$$

This indicates that any location `px` represents may point to any location `x` represents, and simplifies to

$$\texttt{points\_to}(@_1) = \{@_0\}$$

The set of locations potentially associated with the `load` on line 11 corresponds to

$$\bigcup_{@_i \in \texttt{alias[px]}} \texttt{points\_to}[@_i]$$

which is $@_0$. Thus we have

$$\begin{aligned} \texttt{alias}(y) &= @_0 \\ \texttt{alias}(z) &= @_0 \end{aligned}$$

and know the load on line 13 reads from address $@_0$. Only one equation is generated for the load.

3.8 Runs and saves

C loops and returns are elaborated using Core *labels*, `runs`, and `saves`. A `run` "jumps to" the corresponding `save`, such that the continuation is not executed.

Figure 3.5 shows a C function call, with a simplified corresponding Core program. C functions are elaborated into procedures terminating with `save`. For the purposes of this report, the semantics of `save`$(\overline{ident_i := pe_i}^{i})$ `in` e is $e[pe_i/ident_i]$,

```
int foo(int x) {
  return x;
}

int main(void) {
  if (1) return 2;
  return foo(1);
}
```

```
proc foo (x : pointer) : eff loaded integer :=
  let strong y : loaded integer =
      load("signed␣int", pure(x)) in
  run ret(y);
  (save ret : loaded integer
      (a : loaded integer :=
          undef(<<end_of_function>>)) in
      pure(a))

proc main () : eff loaded integer :=
  if true then run ret(Specified(2))
          else skip;
  let strong tmp_ptr : pointer =
      create(Ivalignof("signed␣int", "signed␣int")) in
  store("signed␣int", tmp_ptr, Specified(1));
  let strong z : loaded integer =
      ccall(Cfunction(foo), tmp_ptr) in
  run ret(z);
  (save ret: loaded integer
      (a: loaded integer := Specified(0)) in
      pure(a))
```

Figure 3.5: A simplified, hand-written Core program (right) representing the C program on the left. The function call is never executed.

that is e with pe_i substituted for each $ident_i$. Recall these substitutions are used to handle object lifetimes in C cross-scope gotos. The `main` procedure terminates with a "default" `save` with value `Specified(0)` such that if no `run ret` is reached, the procedure returns 0. Other procedures returning values have `undef` as the default "return value", indicating undefined behaviour if the end of a function is reached (and the value is used by the caller).

The `run` $label(pe_1, \ldots, pe_n)$ acts as a "goto" to the corresponding `save` $label$, with pe_i substituted for each occurrence of $ident_i$ in the `save`. For example, executing `ret(Specified(2))` jumps to the end of the procedure with the new continuation being `pure(Specified(2))` and the old continuation (`let strong tmp_ptr`...) "dropped".

`let strong/weak` pat = e_1 `in` e_2 expressions involve sequencing of two effectful expressions, such that e_1 may contain a `run`. If e_1 contains a `run`, the "continuation" e_2 is dropped and not reachable. We define a function

```
drop_cont :  expr -> Z3Expr
```

computing an SMT Boolean predicating whether a `run` occurred in the expression (guarded by control flow). The VCs of e_2 are guarded by $\neg(\texttt{drop\_cont}(e_1))$ such that

$$\texttt{VC}(\texttt{let weak } pat = e_1 \texttt{ in } e_2) = \texttt{VC}(e_1) \wedge (\neg\texttt{drop\_cont}(e_1) \Rightarrow \texttt{VC}(e_2))$$

3.8.1 Return values

The return value of a procedure is constrained by SMT assertions guarded by control flow, accounting for whether a return was previously encountered. For `main` in Figure 3.5, this is expressed as

$$\begin{aligned}(\top \Rightarrow &\texttt{|ret| = LI\_Specified(2)})\\ \wedge(\neg\top \Rightarrow &\texttt{|ret| = LI\_Specified(z)}\\ \wedge(\neg(\top \vee \neg\top) \Rightarrow &\texttt{|ret| = LI\_Specified(0)})\end{aligned}$$

For procedure return values, there are no expressions after `save` $ret(\overline{ident_i := \dots}^{i})$ in e, thus we can translate `run` $ret(\overline{pe_i})$ simply to $e[pe_i/ident_i]$. This is not the case for arbitrary `runs` and `saves` (see § 3.8.3).

3.8.2 Function calls

C functions are elaborated into Core procedures with arguments of `pointer` type. Function calls are elaborated by creating temporary Core pointers for arguments.

The Core expression `ccall(Cfunction(foo), tmp_ptr)` is represented in SMT as the *return value* of the called function, with `tmp_ptr` substituted for each occurrence of the argument `x : pointer` in the function body of `foo`. For general `ccall`$(pe, pe_1, \dots, pe_n)$, the model checking procedure is called on the `body` of pe with $pe_1, \dots, pe_n$ substituted for function arguments (written `body`$(pe, pe_1, \dots, pe_n)$), with

$$\begin{aligned}&\texttt{VC(ccall}(pe, pe_1, \dots, pe_n))\\ =&\texttt{VC(body}(pe, pe_1, \dots, pe_n))\end{aligned}$$

To handle recursive functions, calls beyond a specified depth are placed with `error(...)`. This unwinds function calls to a given bound.

Although pure function calls are inlined prior to model checking, we elect not to inline C function calls to exploit that the `save` for returns occurs at the end of Core procedures and has no continuation.

3.8.3 Loops

Consider the following (simplified) Core program corresponding to

```
while (n > 0) { n = n - 1; }
```

```
int bar(int n) {
  while (n > 0) {
    n = n - 1;
  }
  return n;
}
```

```
proc bar(n : pointer) : eff loaded integer :=
  save loop: integer (n_ptr: pointer := n) in
    let strong Specified(n: integer) =
      load("signed␣int", n_ptr) in
    if n > 1 then
      store("signed␣int", n_ptr, Specified(n-1));
      run loop(n_ptr)
    else
      skip;
  load ("signed␣int", i_ptr)
```

`run loop` is handled similarly to `run ret` for returns, except the `save` is followed by a "continuation" (`load ("signed int", i_ptr)`). Instead of `run loop` being simply replaced by the `save` expression, the `run` is replaced with the `save` expression followed by the continuation:

```
let strong _ = save_expr in continuation
```

The `save_expr` contains a `run`. To keep model checking bounded, we track the loop unwinding depth similarly to function calls and replace `run` with `error` after a given depth.

3.9 Summary

This chapter discussed the compositional translation of Core expressions to SMT constraints and safety properties for a sequential fragment of Core. Next, we discuss C11's relaxed memory model and model checking of concurrent programs.

Chapter 4

Model checking C11 concurrency

This chapter discusses model checking an illustrative fragment of C11's concurrency model. We remove the sequential constraint imposed previously, restoring unsequenced evaluation order and parallel execution. We first introduce relaxed memory models and the C11 concurrency model, and then demonstrate how the axiomatic formalization of Batty et al. [11, 13] is naturally translated to SMT memory constraints.

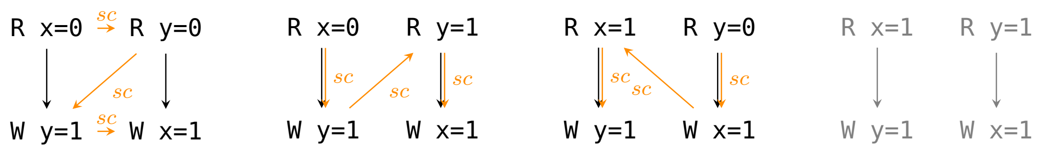
4.1 Relaxed memory models

The concurrency model describes the interaction between threads via shared memory. A *strong* memory model (called *sequentially consistent* or SC) guarantees atomic memory accesses between threads are interleaved to form a total order. A memory model exposing more behaviours than from an interleaving of threads is called *relaxed* or *weak*.

Subtle differences between memory models are often illustrated with short parallel programs called *litmus tests*. Consider the load buffering litmus test below.

Initialisation: `x = 0`, `y = 0`	
Thread 1	Thread 2
`r1 = x.load()`	`r2 = y.load()`
`y.store(1)`	`x.store(1)`

Assuming loads and stores are interleaved in a total SC order, the values of `r1` and `r2` cannot both be `1`. The first three executions below illustrate a sequentially consistent ordering of the memory actions, such that `(r1,r2)=(0,0)`,`(0,1)`, *or* `(1,0)`.



In architectures with weak memory models such as ARM and x86, memory actions can be rearranged. For instance, the loads and stores may be rearranged and both `r1` and `r2` can equal `1`. SC ordering can be implemented by inserting fence instructions appropriately into compiled code.

4.2 C11 memory model

To enable efficient implementations on modern parallel architectures, the standardization committees extended C with support for low-level atomic operations and an accompanying relaxed memory model. C11 programs manipulate a set of shared memory locations, each of *atomic* or *non-atomic* type. Atomic locations are accessed with atomic reads, writes, and read-modify-writes. Atomic and non-atomic locations are accessed with non-atomic reads and writes. There are additionally fences and locks (omitted for simplicity). Each action is associated with a thread id *tid* and a unique action id *aid*.

Definition (Memory action (fences and locks omitted)).

$$\begin{array}{lll} \text{action} = & & \\ & aid, tid : \mathrm{R}_{na}\, l = v & \text{non-atomic read} \\ \mid & aid, tid : \mathrm{W}_{na}\, l = v & \text{non-atomic write} \\ \mid & aid, tid : \mathrm{R}_{mo}\, l = v & \text{atomic read} \\ \mid & aid, tid : \mathrm{W}_{mo}\, l = v & \text{atomic write} \end{array}$$

Atomic memory actions have a *memory order* (*mo*) [37, J.2]:
Definition (Memory order[1]).

$$\begin{array}{ll} o ::= \texttt{RLX} & \text{relaxed} \\ \quad \mid \texttt{ACQ} & \text{acquire, reads/RMWs only} \\ \quad \mid \texttt{REL} & \text{release, writes/RMWs only} \\ \quad \mid \texttt{AR} & \text{acquire-release, RMWs only} \\ \quad \mid \texttt{SC} & \text{sequentially consistent, the default} \end{array}$$

The memory order, defaulting to `SC`, constrains how memory accesses are ordered around an atomic operation.

[1] We exclude `memory_order_consume` for simplicity as in [11] (their actual formalization includes `memory_order_consume`). Although `memory_order_consume` exists to allow more efficient implementations, it can (and is, in many compilers) safely be substituted with `memory_order_acquire`. Furthermore, its definition is subject to discussion and revision by the WG14 committee [42]

Relaxed: The *relaxed* order imposes few constraints, only guaranteeing atomicity and a total ordering between atomic writes to the same location .

Release/acquire: Release/acquire allows synchronization between threads. A write-release can not be reordered with any memory action *preceding* it in program order. A read-acquire can not be reordered with any memory action *following* it in program order. Release/acquire synchronization supports the following idiom where the producer writes to `x` and sets a flag `y`, while the consumer spins on `y`. The consumer must see the data written by the producer.

```
// producer         | // consumer
x = ...             | while(0 == y.load(ACQ));
y.store(1, REL)     | r = x
```

Sequentially consistent: The *sequentially consistent* order establishes a total order on all `SC` operations. `SC` operations also order memory in the same way as release/acquire ordering, such that everything that happens before an `SC` write in one thread is "visible" in the thread that "reads-from" the write.

In the C11 memory model, the semantics of a program are defined in terms of a set of allowed *executions*. An execution is essentially a partial order over memory actions. The ISO standard constrains the set of allowed executions with *consistency axioms* defined using a number of relations. For example, the *modification order* relation [37, §5.1.2.4] ensures stores in each atomic location are totally ordered, such that there is a memory-coherent ordering shared by the program. Modification order must be consistent with the *happens before* relation, which encompasses ordering between events from the same thread as well as synchronisation across threads. The set of allowed executions describes possible behaviours that can occur.

4.2.1 Relations

We summarize the relations defined in the ISO standard:

sequenced-before (sb): a strict partial order between events from the same thread established by the operational semantics. This captures program order.

sequential consistency (sc): a strict total order between `SC` events. The total order establishes an "interleaving" semantics.

reads-from (rf): a relation linking writes to reads, such that $(w, r) \in rf$ if event r reads the value stored by w.

modification order (mo): a strict total order between writes to the same atomic location, representing a memory-coherent ordering shared by the program.

synchronizes-with (sw): a transitive relation capturing synchronisation between events from different threads. Synchronisation occurs from thread create/join and release/acquire synchronisation.

happens-before (hb): the transitive closure of *sb* and *sw*. In relaxed memory models, there is no global linear time so the "most recent" write does not necessarily exist. Instead, *rf* is constrained by the *hb* order.

Consider the following store buffering (a.k.a. Dekker's) litmus test[2] with release-acquire pairs.

```
int main() {
  _Atomic(int) x=0, y=0;
  int z1, z2;
  {{{ { atomic_store_explicit(&y, 1, memory_order_release);
        z1 = atomic_load_explicit(&x, memory_order_acquire); }
  ||| { atomic_store_explicit(&x, 1, memory_order_release);
        z2 = atomic_load_explicit(&y, memory_order_acquire); }
  }}};

  return z1 + (2 * z2);
}
```

There are four possible return values: $(0, 1, 2, 3)$. These executions are depicted in Figure 4.1 with loads and stores to `z1` and `z2` elided. Each column contains memory actions of a different thread, with actions within a thread ordered by the `sb` relation. Action `(b)` is synchronized-with actions `(c)` and `(e)` from the thread create. The `mo` relation orders stores for non-atomic locations.

Each read (actions `(d)` and `(f)`) reads-from either of the two writes in the corresponding locations. Figure 4.1a demonstrates relaxed memory ordering. A read-acquire reading from a write-release leads to an `sw` edge, such that all memory actions that happen-before the write-release can not be reordered with the write-release and all memory action that happen after the read-acquire can not be reordered with the read-acquire.

4.3 Operational semantics: pre-execution

We first extract a set of executions (called *pre-executions*) based on the operational C11 semantics from the Core program. Each pre-execution is compatible with memory actions in individual threads, but does not account for behaviour of shared memory and hence over-approximates allowed behaviours.

Definition (Pre-execution). A *pre-execution* X consists of a tuple (E, thd, sb, asw) where

– E is a set of memory actions or events

– thd is an equivalence relation between events from the same thread

– $sb \subseteq thd$ is the *sequenced-before* relation

[2]The notation `{{{...|||...}}}` expresses parallel execution.

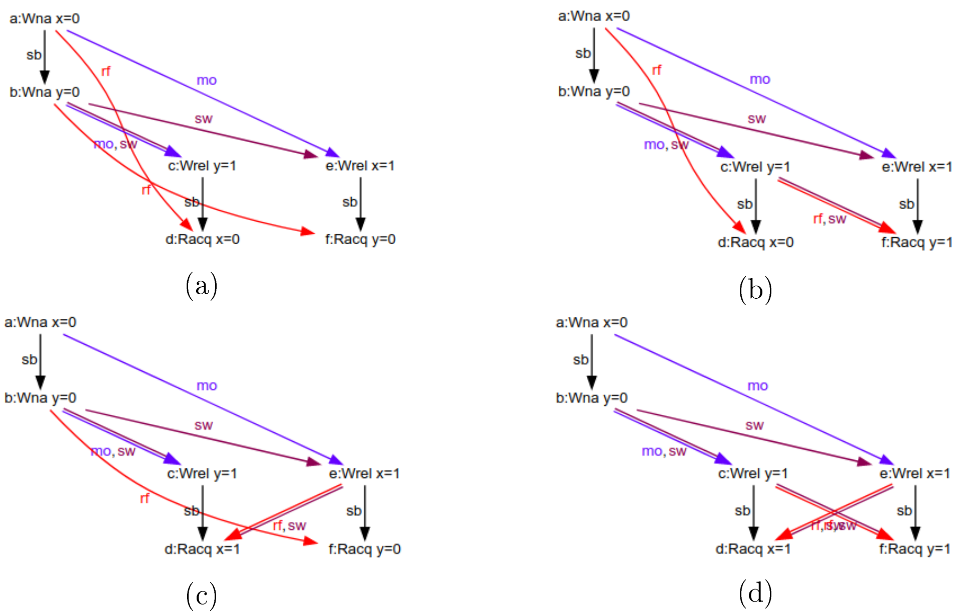


Figure 4.1: Four possible executions for the store buffering litmus test. Actions in the same column are in the same thread and related by *sb*. A read-acquire reading from (*rf*) a write-release introduces a synchronization *sw* edge. The top left demonstrates relaxed ordering.

– *asw* is the *additional-synchronized-with* relation, capturing thread creates/joins

Event sort Let `events(expr)` be the set of events in Core expression `expr`. We accumulate a set of events by recursing over the the AST. For instance, $\texttt{events}(\texttt{unseq}(e_1, \ldots, e_n)) = \bigcup_{i=1}^{n} \texttt{events}(e_i)$. Each event has an associated memory action (with location, type, etc.) and control-flow *guard*, an SMT expression predicating whether the action actually occurs (as when computing VCs).

sb and asw The syntactic structure of C/Core programs determines the *sb* and *asw* relation, independent of memory model. In Core, sequencing of memory actions is made explicit via `unseq`, `nd`, and `let strong/weak`. Memory actions have a polarity $\in \{+, -\}$ in Core, with positive actions sequenced by both `let strong` and `let weak` and negative actions sequenced only by `let strong`. The compositional computation of *sb* is summarized in Table 4.1. The `sb` relation on a Core expression is essentially the union of the `sb` relations on the subexpressions (with the exception of `let strong/weak pat = e1 in e2`, which sequences actions in `e1` before actions in `e2`).

The *asw* relation is computed similarly (as the union of *asw* of the subexpressions) apart from with $\texttt{par}(e_1, \ldots, e_n)$. We track thread creates and joins, such that if thread t_1 creates thread t_2, all maximal actions in t_1 before the creation of thread t_2 (that is, actions a from t_1 such that $\neg\exists b.(a, b) \in sb_1$ for sb_1 the *sb* relation before thread creation) are *asw*-related before minimal actions in t_2, and similar

`expr`	`sb(expr)`
`pure`(pe)	$\{\}$
`create`(pe_1, pe_2)	$\{\}$
`store`$(pe_1, pe_2, pe, \mathit{memory\text{-}order})$	$\{\}$
`load`$(pe_1, pe_2, \mathit{memory\text{-}order})$	$\{\}$
`case` pe `with` $\overline{\vert pat_i \Rightarrow e_i}^{i}$ `end`	$\bigcup_i$ `sb`(e_i)
`if` pe `then` e_1 `else` e_2	`sb`$(e_1) \cup$ `sb`(e_2)
`skip`	$\{\}$
`pcall`$(pe, pe_1, \ldots, pe_n)$	`sb(body`$(pe, pe_1, \ldots, pe_n)$`)`
`unseq`$(e_1, \ldots, e_n)$	$\bigcup_i$ `sb`(e_i)
`let weak` $pat = e_1$ `in` e_2	$\{(a_1, a_2) \in$ `E`$(e_1) \times$ `E`$(e_2) \vert$ `is_pos`$(a_1) \wedge$ `is_pos`$(a_2)\}$ $\cup$`sb`$(e_1) \cup$ `sb`(e_2)
`let strong` $pat = e_1$ `in` e_2	$\{(a_1, a_2) \in$ `E`$(e_1) \times$ `E`$(e_2)\}$ $\cup$`sb`$(e_1) \cup$ `sb`(e_2)
`nd`$(e_1, \ldots, e_n)$	$\bigcup_i$ `sb`(e_i)
`save` $label(\overline{ident_i := pe_i}^{i})$ `in` e	`sb`$(e[\overline{pe_i/ident_i}^{i}])$
`run` $label(\overline{ident_i := pe_i}^{i})$	`sb(find_save`$(label)[\overline{pe_i/ident_i}]$`)`
`par`$(e_1, \ldots, e_n)$	$\bigcup_i$ `sb`(e_i)

Table 4.1: Computation of the sequenced-before relation. `sb(expr)` is essentially the union of the sb relation of the subexpressions, apart from `let weak` and `let strong` which explicitly sequence memory actions. Here, `E(expr)` is the set of events in `expr`. Omitted here, initial events from a `create` are sequenced before all non-initial events.

for thread joins.

4.4 Candidate executions

We augment pre-executions with existentially quantified *witnesses* to form candidate executions, constrained using *well-formed* and *consistency* axioms. A candidate execution introduces (memory) constraints on the load and store values.

Definition (Candidate execution)**.** A *candidate execution* consists of a pair (X, w), where X is a pre-execution and $w = (\mathit{rf}, \mathit{mo}, \mathit{sc})$ a *witness* consisting of the reads-from, modification order, and sequential consistency relations.

Definition (well-formed)**.** A candidate execution is *well-formed* if the following axioms [11] hold:

Axiom 1 *well-formed reads-from*: every read event is linked to a unique write event of the same location ($=_{loc}$) such that the values match.

$$\forall e \in R. \exists! e' \in W. (e', e) \in \mathit{rf}$$
$$\text{and} \qquad \mathit{rf} \subseteq (=_{loc} \cap =_{val})$$

Axiom 2 *well-formed modification order*: the modification order relates in a strict total order (mo is acyclic) all and only those events that write to the same atomic location

$$\begin{aligned}&(mo \cup mo^{-1}) = (=_{loc} \cap W^2 \setminus nal^2 \setminus id)\\ &\text{and} \qquad \text{acy}(mo)\end{aligned}$$

Axiom 3 *well-formed sequential consistency*: SC relates in a strict total order all and only the `SC` events:

$$\begin{aligned}&(SC \cup SC^{-1}) = (\texttt{SC}^2 \setminus id)\\ &\text{and} \qquad \text{acy}(SC)\end{aligned}$$

We express the well-formed axioms using SMT expressions quantified over the set of events. First, we define SMT functions to represent relations:

`rf`	reads-from	$E \rightarrow E$
`mo`	modification-order	$E \rightarrow \mathbb{Z}$
`sc`	sequential-consistency	$E \rightarrow \mathbb{Z}$
`hb`	happens-before	$E \times E \rightarrow \mathbb{B}$
`sw`	synchronizes-with	$E \times E \rightarrow \mathbb{B}$
`fr`	from-reads	$E \times E \rightarrow \mathbb{B}$

We define reads-from as a *function* `rf : E -> E` such that `rf(r) = w` if `r` $\in R$ reads from `w` $\in W$. Expressing `rf` as a function ensures every read is associated with exactly one write, and reduces the number of quantifiers needed when reasoning about *rf*. For example, Axiom 1 can be expressed in first-order logic as follows:

$$\begin{aligned}&\forall \texttt{e.isRead(e)} \wedge \texttt{guard(e)} \Rightarrow\\ &\quad \texttt{isWrite(rf(e))} \wedge \texttt{guard(rf(e))}\\ &\quad \wedge \texttt{loc(e)} = \texttt{loc(rf(e))}\\ &\quad \wedge \texttt{val(e)} = \texttt{val(rf(e))}\end{aligned}$$

Instead of using two quantifiers to express $\forall e \in R.\exists! e' \in W$, we use a single quantifier. As `rf` is defined over the entire domain of events, we constrain the value of `rf(e)` only if `e` is a read and the event actually occurs (`guard(e)` holds).

As mo and sc are relations expressing strict total order, they can be represented as functions from events to integers such that

$$\text{events } \texttt{a} \text{ and } \texttt{b}\ mo\text{-related} \Rightarrow \texttt{mo(a) < mo(b)}.$$

Expressing mo and sc using these “clock” constraints removes the need to directly encode transitive closure when defining mo and sc in Z3. The implication does not go the other way; for example two events at different locations could be ordered by the `mo`-clock. Rather than assert the "all and only the" part of the well-formed axioms, for simplicity we only constrain the Z3 relations or functions on the relevant events and guard their usage.

For instance, Axiom 2 can be expressed as

$$\begin{aligned}&\forall \texttt{e1, e2.isWrite(e1)} \wedge \texttt{isWrite(e2)} \wedge \texttt{guard(e1)} \wedge \texttt{guard(e2)}\\&\qquad \wedge (\texttt{loc(e1) = loc(e2)}) \wedge \texttt{isAtomic(loc(e1))} \wedge (\texttt{e1} \neq \texttt{e2})\\&\qquad \Rightarrow (\texttt{mo(e1)} < \texttt{mo(e2)}) \vee (\texttt{mo(e1)} > \texttt{mo(e2)})\end{aligned}$$

We only constrain the *mo*-ordering of distinct write events at the same non-atomic location that occur in the program, allowing arbitrary ordering between other event pairs.

4.5 Consistent executions

After asserting executions are well-formed, we assert their consistency with the memory model.
Definition (consistent)**.** As formalized by Batty et al. [11], a candidate execution (X, w) is *consistent* if it is well-formed and satisfies the following:

Axiom 1 irr(hb): happens-before contains no cycles[3]

Axiom 2 Coherence axioms: relationship between hb and mo (see Figure 4.2)

CoRR two reads ordered by hb may not read from two writes mo-ordered in the opposite direction ($\texttt{irr}(rf^{-1}; mo; rf; hb)$[4])

CoWR a read must not observe a write that is happens-before hidden by a later write in the modification order ($\texttt{irr}(rf^{-1}; mo; hb)$)

CoWW happens-before and modification order must not disagree ($\texttt{irr}(mo; hb)$)

CoRW a read must not observe a write that is mo-after a write it happens before ($\texttt{irr}(mo; rf; hb)$)

Axiom 3 $\texttt{irr}(rf; hb)$: a read must not observe a write that happens after it

Axiom 4 $\texttt{empty}((rf; [nal] \setminus vis)$: a read of a non-atomic location must observe a visible write, where a write w is visible to a read r (written $vis(w, r)$) if it is the most recent write in happens-before.

[3]As happens-before is transitively closed, this is equivalent to happens-before being *irreflexive*.
[4]rf^{-1} denotes the inverse relation of rf. $R; S$ denotes the composition of relations R and S, such that $aRb \wedge bSc \Rightarrow a(R; S)c$

$$\begin{tikzcd}
\texttt{a:W x=1} \arrow[r, "\texttt{rf}"] & \texttt{c:R x=1} \arrow[d, "\texttt{hb}"] \\
\texttt{b:W x=2} \arrow[u, "\texttt{mo}"] \arrow[r, "\texttt{rf}"] & \texttt{d:R x=2}
\end{tikzcd}$$

(a) CoRR

$$\begin{tikzcd}
\texttt{a:W x=1} \arrow[r, "\texttt{mo}"] \arrow[dr, "\texttt{rf}"] & \texttt{b:W x=2} \arrow[d, "\texttt{hb}"] \\
 & \texttt{c:R x=1}
\end{tikzcd}$$

(b) CoWR

$$\begin{tikzcd}
\texttt{a:W x=1} \arrow[d, bend left, "\texttt{hb}"] \\
\texttt{b:W x=2} \arrow[u, bend left, "\texttt{mo}"]
\end{tikzcd}$$

(c) CoWW

$$\begin{tikzcd}
\texttt{a:W x=1} \arrow[r, "\texttt{rf}"] & \texttt{b:R x=1} \arrow[d, "\texttt{hb}"] \\
 & \texttt{c:W x=2} \arrow[ul, "\texttt{mo}"]
\end{tikzcd}$$

(d) CoRW

Figure 4.2: The four irreflexive coherence axioms. An execution is not consistent if any of these cycles occur.

Axiom 5 $\texttt{acy}(SC^2 \setminus id \cap (hb \cup fr \cup mo))$[5]: SC must be consistent with happens-before, "from-read", and modification-order. The from-read relation links each read to all the writes that are mo after the write it read from ($fr = rf^{-1}; mo$).

These axioms are expressed in Z3 using the *mo* and *sc* clock and *rf* functions to reduce the complexity of the quantified expression.

For example, the CoRR constraint can be expressed by quantifying over two read events:

$$\begin{aligned}\forall \texttt{e1, e2}.\texttt{isRead(e1)} \wedge \texttt{isRead(e2)} \wedge \texttt{guard(e1)} \wedge \texttt{guard(e2)} \wedge \\ \texttt{hb(e1,e2)} \wedge (\texttt{loc(e1)} = \texttt{loc(e2)}) \wedge \texttt{isAtomic(loc(e1))} \\ \Rightarrow \neg(\texttt{mo(rf(e2))} < \texttt{mo(rf(e1))})\end{aligned}$$

In Figure 4.2a, `e1` and `e2` correspond to actions `(c)` and `(d)`. We assert that assuming `e1` and `e2` are happens-before ordered, the writes they read from (`(a)` and `(b)`) must not be *mo*-ordered in the opposite direction. Our definition of *rf* as a function from reads to writes and *mo* as a "clock" (function from events to integers) allows us to quantify over two events to avoid expressing relational composition (which naturally involves an existential, with $a(R;S)c \iff \exists b.\ aRb \wedge bRc$). The trade-off is the need to introduce additional (simple) guards, such as asserting the events are at the same atomic location (else the `mo` ordering is meaningless) and `e1` and `e2` are reads (else `rf(e1)` and `rf(e2)` is meaningless).

The first three (irreflexive) axioms are expressed similarly. Axiom 4 is represented by asserting `rf(e)` is visible to `e` (which is true if `rf(e)` happens-before `e` and there does not exist another write `w` at the same location such that $\texttt{rf(e)} \xrightarrow{\texttt{hb}} \texttt{w} \xrightarrow{\texttt{hb}} \texttt{e}$). Axiom 5 is expressed by asserting that if distinct SC events `e1` and `e2` are related by `hb`, `fr`, or <-ordered by `mo`, they must be <-ordered by`sc`.

[5]This axiom from Batty et al.'s "Overhauling SC Atomics" [11] strengthens the C11 memory model (disallowing certain executions). They argue that this simplifies the formalization (reducing the number of SC axioms from seven to one) and is a natural strengthening.

4.5.1 Auxiliary relations

We directly encode the `hb`, `sw`, and `fr` relations as functions $E \times E \rightarrow \mathbb{B}$ in Z3. A clock (such as used for `mo`) can not be used for `hb` as there is no global linear time.

happens-before The transitive closure of sequenced before and synchronizes-with is expressed in Z3 as follows:

$$\begin{aligned}\forall \texttt{e1,e2}.\ \texttt{hb(e1,e2)} = &\texttt{sb(e1,e2)} \vee \texttt{sw(e1,e2)}\\ &\vee \exists \texttt{e3}.\ ((\texttt{sb(e1,e3)} \vee \texttt{sw(e1,e3)}) \wedge \texttt{hb(e3,e2)})\end{aligned}$$

synchronizes-with Excluding RMWs, locks, and fences, the *sw* relation is defined by Batty et al. in [11] as

$$sw = ([rel]; rf; [acq]) \setminus thd$$

where

$$\begin{aligned}acq &= \texttt{ACQ} \cup (\texttt{SC} \cap R)\\ rel &= \texttt{REL} \cup (\texttt{SC} \cap W)\end{aligned}$$

This captures release-acquire synchronization, where a read-acquire reading from a write-release introduces a synchronization edge. We additionally assert the existence of synchronization edges for thread creates and joins (the "additional-synchronizes-with" relation). In Z3:

$$\begin{aligned}\forall \texttt{e1,e2}.\ \texttt{sw(e1,e2)} = &\texttt{asw(e1,e2)} \vee\\ &(\texttt{isRel(e1)} \wedge \texttt{isAcq(e2)} \wedge\\ &\ \texttt{guard}(e_1) \wedge \texttt{guard}(e_2) \wedge (\texttt{e1} = \texttt{rf(e2)}) \wedge\\ &\ \texttt{isAtomic}(\texttt{loc}(e_1)) \wedge \neg\texttt{thd(e1,e2)})\end{aligned}$$

from-reads fr relates each read to all writes that are mo after the write it reads from ($fr = rf^{-1}; mo$). We express this as follows, guarding the `rf` and `mo` events appropriately:

$$\begin{aligned}\forall \texttt{e1,e2}.\texttt{fr(e1,e2)} = &\texttt{isRead(e1)} \wedge \texttt{isWrite(e2)} \wedge\\ &\texttt{guard(e1)} \wedge \texttt{guard(e2)} \wedge \texttt{isAtomic(loc(e2))} \wedge\\ &\texttt{mo(rf(e1))} < \texttt{mo(e2)}\end{aligned}$$

Ultimately, we observe that the axioms and relations are encoded naturally according to the axiomatic formulation of Batty et al., with simplifications by using clock-constraints for `sc` and `mo` and expressing `rf` as a function.

```
int main() {
  int x = 1;
  int y;
  {{{ x = 2;
  ||| y = (x == 2);
  }}}
}
```

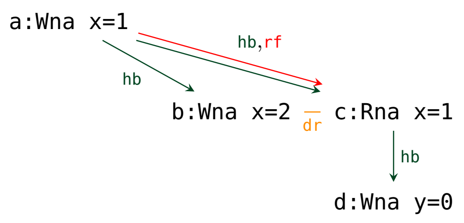


Figure 4.3: Data race example. There is a data race between actions `(b)` and `(c)`.

4.6 Race-free executions

Consistent executions (executions consistent with memory ordering) have undefined behaviour if they have *data races* or *unsequenced races*.

A data race occurs when there are

> two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other [37, §5.1.2.4]

In Figure 4.3, the conflicting, non-atomic actions `(b)` and `(c)` are not *hb*-related. Action `(c)` can consistently read-from `(a)` or `(b)`. A data race exists.

We either assert no executions contain races, or extract consistent executions from Z3 and identify (outside of Z3) exactly which executions contain data races. We implement both options.

A no-data-race assertion resembles the following, asserting conflicting actions from different threads are *hb*-related:

$$\begin{aligned}
\forall \texttt{e1,e2}.\ & \texttt{e1} \neq \texttt{e2} \wedge \texttt{isWrite(e1)} \wedge \texttt{isWrite(e2)} \wedge \\
& \big(\texttt{loc(e1)} = \texttt{loc(e2)}\big) \wedge \neg\texttt{thd(e1,e2)} \wedge \\
& \big(\texttt{is\_na(e1)} \vee \texttt{is\_na(e2)}\big) \wedge \texttt{guard(e1)} \wedge \texttt{guard(e2)} \\
& \Rightarrow \texttt{hb(e1,e2)} \vee \texttt{hb(e2,e1)}
\end{aligned}$$

A pre-execution has an *unsequenced race* (recall § 2.1.1) if there is a write and another read/write to the same location from the same thread not sequenced by sequenced-before:

$$\begin{aligned}
\forall \texttt{e1,e2}.\ & \big(\texttt{e1} \neq \texttt{e2}\big) \wedge \big(\texttt{loc(e1)} = \texttt{loc(e2)}\big) \wedge \\
& \big(\texttt{isWrite(e1)} \vee \texttt{isWrite(e2)}\big) \wedge \\
& \big(\texttt{thd(e1,e2)} \wedge \texttt{guard(e1)} \wedge \texttt{guard(e2)} \\
& \Rightarrow \texttt{sb(e1,e2)} \vee \texttt{sb(e2,e1)}
\end{aligned}$$

4.7 Summary: model checking procedure

This chapter described the extraction of memory actions and programs order from Core programs, and how memory constraints were expressed using C11 well-formed and consistency axioms in Z3. We summarize this below:

1. As in Chapter 3, we compute syntactic constraints and verification conditions. Only memory constraints for loads and stores differ in the concurrent mode.
2. We extract the memory events along with the sequenced-before relation (representing program order) and additional-synchronized-with relation (representing thread create/join):

   create: address creation is identical. We create an initial action representing an unspecified value (as before) that is sequenced-before non-initial actions.

   store: create a write memory event associated with the value being stored.

   load: create a read memory event associated with the value being written. Distinct from the sequential mode, we do not explicitly track the value currently at the location being read from.
3. We assert well-formed and consistency constraints to constrain candidate executions and the values of loads/stores. The conjunction of syntactic constraints S act as assumptions. The conjunction of memory constraints M given the syntactic assumptions, expressed as below

   $$\bigwedge_{s \in S} s \wedge \bigwedge_{m \in M} m$$

   is satisfiable iff there exists a consistent execution.
4. Given a satisfiable interpretation, we optionally extract all consistent executions from Z3 by successively adding "distinctness" assertions to require different executions. We then directly (in Ocaml) compute the existence of races and output visualizations of all executions.
5. Alternatively (or additionally, by removing the distinctness assertions from Z3's "stack" of assertions), we treat the race check as a verification condition.

   $$\bigwedge_{s \in S} s \wedge \bigwedge_{m \in M} m \wedge \neg(\bigwedge_{v \in V} v)$$

   This equation is satisfiable iff any of the verification conditions (including race assertions) fail to hold.

Chapter 5

Evaluation

We demonstrate the model checker on simple C programs, followed by experimental evaluation of implementation correctness and performance. § 5.4 discusses related work.

5.1 Example programs

Figure 5.1 contains two C programs analysed with the model checker.

5.1.1 Single-threaded program

The first program, in Figure 5.1a, computes the factorial of 5 in two ways: the first by initializing an array such that `a[i] = i+1` and then multiplying the elements in the array, and the second with a `while` loop. The program features arrays, control flow, loops, function calls, integer arithmetic, and pointers.

Translation to Core The C program is translated into a 500-line Core program based on the Cerberus semantics. This makes explicit aspects of C such as pointer safety checks, implementation-defined behaviour, and, crucially, undefined behaviour.

Core-to-Core rewrites We rewrite implementation-defined expressions based on a given implementation. We optionally sequentialise the program by selecting, as a Core-to-Core rewrite, an ordering for unspecified evaluation orders (arising from, for instance, evaluating arguments to operators such as `(+)`). This enables usage of the sequential model checker presented in Chapter 3 with the trade-off being losing the ability to detect unsequenced races and all behaviours when arguments have side effects.

```
void init(int* p, int n) {
  for (int i = 0; i < n; i++) {
    p[i] = i + 1;
  }
}

int multiply(int* p, int n) {
  int ret = 1;
  for (int i = 0; i < n; i++) {
    ret *= p[i];
  }
  return ret;
}

int factorial(int n) {
  int x = 1;
  while (n > 0) {
    x *= n;
    n--;
  }
  return x;
}
```

```
int main(void) {
  int a[5];
  // ERROR: assert(a[1] == 0);
  // ERROR: a[-1] = 0;
  if (42) {
    a[1] = 0;
  }
  a[1] = 0;
  assert(a[1] == 0);
  init(a, 5);
  // ERROR: init(a, 6);
  // ERROR: assert(a[1] == 0)
  assert(a[1] == 2);
  int fact = multiply(a, 5);
  // ERROR: multiply(a, 6);
  assert (fact == 120);
  // ERROR: assert(fact != 120);
  int fact2 = factorial(5);
  assert(fact == fact2);
  // ERROR: a[0] = 1 / (a[1] - 2);
  // ERROR: a[1] = INT_MAX + 1;
}
```

(a) A C program with control flow, loops, function calls, integer arithmetic, arrays, and pointers. The model checker returns `NOT SATISFIABLE`, indicating no faulty states could be reached. Uncommenting any of the `ERROR` lines results in the model checker returning `SATISFIABLE` due to buffer overflow/underflow, division by zero, or failed assertions.

```
#include <stdatomic.h>

int main(void) {
  _Atomic(int) x = 0;
  _Atomic(int) y = 0;
  int z1; int z2; int z3; int z4;
  {{{ atomic_store_explicit(&x, 1, memory_order_release);
  ||| atomic_store_explicit(&y, 1, memory_order_release);
  ||| { z1=atomic_load_explicit(&x, memory_order_acquire);
        z2=atomic_load_explicit(&y, memory_order_acquire); }
  ||| { z3=atomic_load_explicit(&y, memory_order_acquire);
        z4=atomic_load_explicit(&x, memory_order_acquire); }
  }}};
  return (z1 + 2 * (z2 + 2 * (z3 + 2 * z4))); }
```

(b) The IRIW (Independent-Reads-Independent-Writes) litmus test. The reading threads do not have to see the stores in the same order, so there are 16 possible return returns. If the memory order were sequentially consistent, the `(z1,z2,z3,z4)=(1,0,0,1)` execution would not be allowed. Thus there would be 15 possible return values.

Figure 5.1: Examples of C programs handled by the model checker

Generating an SMT problem Function calls and loops are unwound to a given depth. We encode the syntactic constraints of the Core program as well as the verification and imposed memory constraints into an SMT problem, which is passed to the Z3 SMT solver.

Result For the program in Figure 5.1a, Z3 returns `NOT SATISFIABLE`, indicating no error occurs. Uncommenting any of the `ERROR` lines results in Z3 returning `SATISFIABLE` due to reachability of errors such as buffer overflow, signed integer overflow, division by zero, and failed user-defined assertions. This entire process, with loop unrolling depth of 6, took ~7s with the sequential mode. The SMT problem had ~7750 fresh variables and ~7250 assertions (mostly from syntactic assumptions on variables).

5.1.2 Multi-threaded litmus test

Figure 5.1b contains a litmus test, a short parallel program designed to identify subtle differences between memory consistency models. The test featured is IRIW (independent-reads-independent-writes), which tests whether the reader threads can see the stores in a different order. With relaxed or release/acquire memory orders, the threads can see the stores in a different order and there are 16 possible executions (each load can read 0 or 1). With the sequentially consistent memory order, the execution `(z1,z2,z3,z4)=(1,0,0,1)` is not allowed and there are only 15 possible return values.

Initialisation: $x = 0$, $y = 0$

Thread 1	Thread 2	Thread 3	Thread 4
`x.store(1)`	`y.store(1)`	`z1 = x.load()`	`z3 = y.load()`
		`z2 = y.load()`	`z4 = x.load()`

Based on Cppmem [12], our tool outputs the 16 (or 15) execution graphs to enable visualization and reasoning about concurrent behaviours. Figure 5.2 presents one such output graph with the reads-from, sequenced-before, synchronizes-with, and modification-order relations. The reader threads see the stores in a different order.

5.2 Correctness of implementation

Using Cerberus as a model for C clarifies the interpretation of the ISO standard and thus the interpretation of the C source analysed by the model checker. Using a principled approach increases confidence in implementation correctness in terms of consistency with the ISO standard. Unspecified and implementation-defined behaviours are made explicit in the elaboration to Core, forcing the model checker to explicitly handle these behaviours.

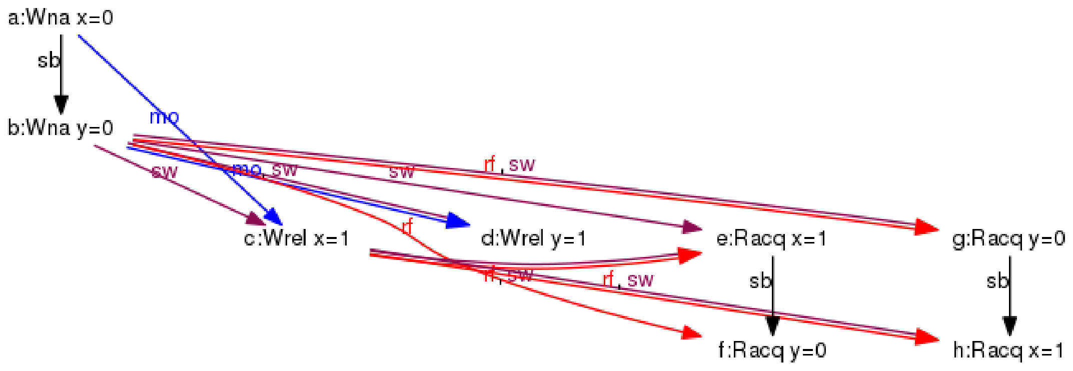


Figure 5.2: Example execution graph for the IRIW test when the reader threads see the stores in a different order. The writes in `(c)` and `(d)` are not related by happens-before, the transitive closure of `sb` and `sw`, and there is no globally consistent linear ordering. The graph was generated by our tool from an execution extracted from Z3, based on pretty-printing code from CPPMEM.

```
void bit_shift_aux(int shift) {
  int a = 1;
  int ret;
  ret = a << shift;
}

void bit_shift() {
  bit_shift_aux(32);
}
```

```
void overrun_st() {
  int buf[5];
  int *p;
  int i;
  p = buf;
  for (i = 0; i <= 5; i ++) {
    *p = 1;
    p++;
  }
}
```

Figure 5.3: Examples of programs in the Toyota test suite. Left: undefined behaviour from bit shift (assuming 32-bit integers). Right: buffer overflow error.

We produced a suite of unit tests used to guide development. Furthermore, we test our implementation on the Toyota software analysis benchmark[1] [54] and litmus tests.

5.2.1 Toyota software analysis benchmark

The Toyota benchmark is a test suite designed to evaluate static analysis tools, with "defects" ranging from buffer overflows to dead code. Not all defects are undefined behaviour. We test our model checker on five categories of defects: bit shifts, data over/underflow, static buffer overrun, and division-by-zero. Tests are of varying complexity, containing pointer aliasing, loops, function calls, and arrays. Two example tests are in Figure 5.3

Our test coverage is summarized in Table 5.1. For each variation, there are tests with and without defects. Some were not analysed due to unimplemented features

[1] `https://github.com/Toyota-ITC-SSD/Software-Analysis-Benchmark/`

Defect type	# Analysed/# of Variations		Defects detected	Defects not UB	False positives
	w/ Defects	w/o Defects			
Bit shifts	15/17	15/17	13	2	0
Data overflow	19/25	19/25	12	7	0
Data underflow	9/12	9/12	8	1	0
Static buffer overrun	35/54	35/54	35	0	0
Zero division	11/16	11/16	11	0	0

Table 5.1: Summary of our coverage of the Toyota test suite for selected defect types. Some tests were not analysed due to unimplemented features in Cerberus or our model checker (e.g. tests with floats). Some defects were not actually undefined behaviour (e.g. unsigned integer overflow) and hence would not be detected by the model checker, or only contained undefined behaviour dependent on the implementation. Accounting for this, the model checker behaved correctly for all analysed tests.

in Cerberus (bitfields), the model checker (i.e. floats, structs) or lack of a `rand()` function. Some "defects" (such as unsigned integer overflow) were not instances of undefined behaviour and thus were not detected (as desired). The model checker behaved as expected, detecting all undefined behaviours in the covered tests with no false positives.

Tests with bit shifts were executed using bit-vectors of size 128 to represent integers, as the Z3 API does not support integer exponentiation and the theory of bit-vectors efficiently implements bit shifts. Tests without loops ran in less than 2s. Tests with loops ran in less than 5s (unwound to depth 6).

5.2.2 Litmus tests

We test our implementation of the C11 concurrency model by analysing litmus tests. These short parallel programs exercise particular behaviours of the memory model, with different behaviours allowed with different memory orders. We ran our tool on all the litmus tests presented in Cppmem examples [12], along with additional variations on the memory order (see Appendix B). The executions found by our tool were the same as those of Cppmem, with the outputted graphs used for additional verification.

5.3 Performance

Although the focus of this project was not efficiency, we evaluate the performance by first discussing the scaling of the translation from C to Core, followed by quantitative evaluation of runtime over a sequence of C programs of increasing size.

5.3.1 From C to Core

Aspects of C programs such as integer conversions, sequencing, and undefined behaviour are implicit in C code. The Cerberus semantics are expressed via a *compositional* translation from an AST to Core, which introduces additional let-bound variables and case-splits. For example, the C program below is elaborated into the Core program presented on page 53. Assuming 32-bit integers, our model checker proves the existence of undefined behaviour resulting from signed integer overflow (from an `undef` in `catch_exceptional_condition` on lines 38-39 being reachable).

```
#include <limits.h>

int main(void) {
  int x = INT_MAX;
  if (x != 32767) {
    x = x + 1;
  }
}
```

The resulting 72-line program contains function calls to `conv_int` and `catch_exceptional_condition` to handle integer conversions and range checks of integers ("representability"). Each of the 13 calls to `conv_int` or `conv_loaded_int` are expanded into ∼20 lines after inlining pure function calls and implementation-defined expressions, resulting in a multi-hundred-line Core program.

There are opportunities for optimization to reduce Core program size (i.e. with constant propagation and branch elimination) or more efficiently represent common functions such as `conv_int` in Z3 (instead of simply inlining the function). Nonetheless, without attempts at optimization, our tool detects undefined behaviour in less than 500ms (in both sequential and concurrent modes).

Non-deterministic evaluation of if-statements In the elaboration, branching on an unspecified value is translated into non-deterministic evaluation of the branches:

```
if Unspecified(_) then e1 else e2 => nd(e1,e2)
```

More generally:

```
if (cond) { e1 } else { e2 } => let bool = ... in
                                case bool of
                                | Specified(a: integer) ->
                                    if not(a==1) then e1 else e2
                                | Unspecified(_) -> nd(e1,e2)
                                end
```

where `bool` is defined to be either `Specified(1)`, `Specified(0)`, or `Unspecified(_)` (lines 5-25) based on interpretation of `cond`.

```
proc main (): eff loaded integer :=
  let strong x: pointer = create(Ivalignof("signed␣int"), "signed␣int") in
  let strong a_74: loaded integer = pure(Specified(2147483647)) in
  store("signed␣int", x, conv_loaded_int("signed␣int", a_74)) ;
  (let weak a_76: loaded integer =
    let weak (a_83: loaded integer, a_84: loaded integer) =
      unseq(let weak (a_78: loaded integer, a_79: loaded integer) =
              unseq(let weak a_77: pointer = pure(x) in
                    load("signed␣int", a_77), pure(Specified(32767))) in
            pure(case (a_78, a_79) of
              | (Specified(a_80: integer), Specified(a_81: integer)) =>
                  if not(conv_int("signed␣int", a_80) == conv_int("signed␣int",
                  a_81)) then
                    Specified(1)
                  else
                    Specified(0)
              | _: (loaded integer,loaded integer) => Unspecified("signed␣int")
            end), pure(Specified(0))) in
    pure(case (a_83, a_84) of
      | (Specified(a_85: integer), Specified(a_86: integer)) =>
          if conv_int("signed␣int", a_85) == conv_int("signed␣int", a_86) then
            Specified(1)
          else
            Specified(0)
      | _: (loaded integer,loaded integer) => Unspecified("signed␣int")
    end) in
  case a_76 of
    | Specified(a_75: integer) =>
        if not(a_75 == 1) then
          let strong _: loaded integer =
            bound[0] (let weak (a_88: pointer, a_89: loaded integer) =
              unseq(pure(x),
              let weak (a_91: loaded integer, a_92: loaded integer) =
                unseq(let weak a_90: pointer = pure(x) in
                      load("signed␣int", a_90), pure(Specified(1))) in
              pure(case (a_91, a_92) of
                | (Specified(a_93: integer), Specified(a_94: integer)) =>
                    Specified(catch_exceptional_condition("signed␣int",
                    conv_int("signed␣int", a_93) + conv_int("signed␣int", a_94)))
                | _: (loaded integer,loaded integer) =>
                    undef(<<UB036_exceptional_condition>>)
              end)) in
            let weak _: unit =
              neg(store("signed␣int", a_88, conv_loaded_int("signed␣int", a_89))) in
            pure(conv_loaded_int("signed␣int", a_89))) in
          pure(Unit) ;
          skip)
        else
          skip
    | Unspecified(_: ctype) =>
        nd(let strong _: loaded integer =
             bound[0] (let weak (a_88: pointer, a_89: loaded integer) =
               unseq(pure(x),
               let weak (a_91: loaded integer, a_92: loaded integer) =
                 unseq(let weak a_90: pointer = pure(x) in
                       load("signed␣int", a_90), pure(Specified(1))) in
               pure(case (a_91, a_92) of
                 | (Specified(a_93: integer), Specified(a_94: integer)) =>
                     Specified(catch_exceptional_condition("signed␣int",
                     conv_int("signed␣int", a_93) + conv_int("signed␣int", a_94)))
                 | _: (loaded integer,loaded integer) =>
                     undef(<<UB036_exceptional_condition>>)
               end)) in
             let weak _: unit =
               neg(store("signed␣int", a_88, conv_loaded_int("signed␣int", a_89))) in
             pure(conv_loaded_int("signed␣int", a_89))) in
           pure(Unit) ;
           skip, skip)
  end) ;
  kill(x) ;
  (save ret_71: loaded integer (a_72: loaded integer:= Specified(0)) in
    pure(a_72))
```

As the branches `e1` and `e2` are duplicated in the Core expression, a sequence of C programs of the form

```
                  if(1) { x = x + 1; }
          if(1) { if(1) { x = x + 1; } }
  if(1) { if(1) { if(1) { x = x + 1; } } }
```

would result in an exponential growth in size of the generated Core program due to the nested `case` expressions, with `x=x+1` appearing twice in the first program (once for each branch of `case`), four times in the second, and eight in the third.

Branching based on unspecified or uninitialized values is a rare condition that is often unintentional; the WG14 committee discussed whether control-flow choices based on unspecified values should be undefined behaviour [43] (ISO C11 is unclear). We adopt this latter semantics and consider branching on unspecified values to be undefined:

```
if Unspecified(_) then e1 else e2 => undef(...)
```

This conveniently removes the exponential growth in code size with nested ifs (and loops, which branch on the loop condition) and was implemented as a four-line change in the definition of the Cerberus model. This demonstrates the decoupling of the model checker with the semantic interpretation of C, with the model checker's implementation being unchanged.

5.3.2 Experiments

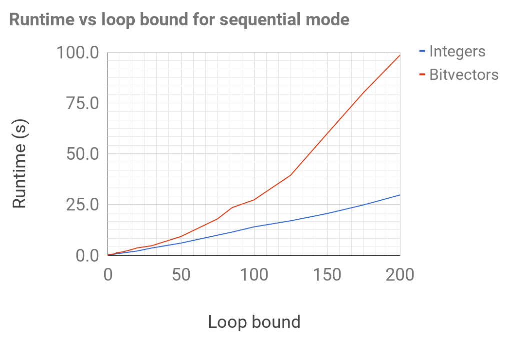
Sequential model: loops

We evaluate the performance of our sequential model on loops of increasing depth `n`, with programs of the following from:

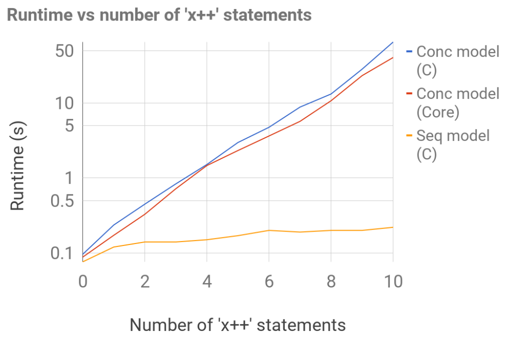
```
int main(void) {
  int x = 0;
  while (x < n) {
    x++;
  }
  assert(x == n);
  return 0;
}
```

Figure 5.4a plots the loop unwind depth against the runtime in seconds averaged over five runs. The runtime using Z3 integers to represent numerical values scales linearly with loop unwind depth, with loops of depth 50 analysed in ∼5s. This corresponds to Core program size scaling linearly (Core program size in characters is approximately $15000n$).

The runtime of the bit-vector representation (size 64) appeared to scale quadratically with the loop unwind and program size. For this program, integer representation appeared more efficient. Integer representation is also consistent with the



(a) Runtime vs loop bound for the sequential mode on a program incrementing a counter in a while loop:
`{int x=0; while(x<n){x++;} assert...}`



(b) Runtime (log scale) vs number of `x++` statements for a simple program of the form
`{int x=0; x++; ...; x++; assert...}`

Figure 5.4: Experiment runtimes

semantics of Core integers as mathematical integers. However, non-linear integer arithmetic is undecidable and operations such as bit-shifts are implemented more efficiently with bit-vectors. We elect to enable the choice of representation with a flag.

Concurrent mode: scaling with memory actions

While the sequential model scales linearly with program size, the concurrent model depends on the number of memory actions. We illustrate this by considering a sequence of C programs of the following form, where the nth program contains n `x++` statements. The generated Core program, and the number of memory actions, scales linearly with the number of `x++` statements (the nth program contains $2n+3$ memory actions).

```
int main(void) {
  int x = 0;         // create, store
  x++;               // load, store; repeat n times
  x++;
  assert (x == 2);   // load: assert (x == n)
}
```

The results are plotted in Figure 5.4b. We plot the runtime for the concurrent and sequential models on the C program, as well as for the concurrent model on analogous hand-written Core programs to reduce the size of the Core program being analysed. We observe exponential scaling with the number of memory actions for the concurrent implementation, with minor differences between analysing C programs and hand-written Core programs. This suggests the number of memory actions dominates runtime for the concurrent model. The sequential model handles these programs in <500ms.

We expressed memory constraints in the concurrent mode using SMT quantifiers.

	# actions	# executions	C runtime (s)	Hand-written Core runtime (s)
LB+acq_rel+acq_rel	12 (8)	3	11.1	1.7
LB+rlx_rlx+rlx_rlx	12 (8)	4	8.4	1.1
LB+Rsc_Wsc+Rsc_Wsc	12 (8)	3	11.7	1.7
MP+na_rel+acq_na	12 (8)	3	12.0	1.3
SB+rel_acq+rel_acq	12 (8)	4	9.7	1.1
SB+Wsc_Rsc+Wsc_Rsc	12 (8)	3 (6)	13.9 (22.2)	0.9
WRC+rel+acq_rel+acq_acq	15 (9)	7	25.0	2.1
IRIW+rel+rel+acq_acq+acq_acq	18 (10)	16	83.5	4.6
IRIW+Wsc+Wsc+Rsc_Rsc+Rsc_Rsc	18 (10)	15 (180)	79.7 (186.3)	5.3

Table 5.2: Runtime to find all consistent executions for various litmus tests. The number of actions refer to the number of memory actions in a Core program generated from the C program, and the number in a hand-written Core litmus test (parenthesized). Here, the number of consistent executions treat different `sc` relations with the same reads-from relation as the same execution, with the number of executions including different `sc` relations (and corresponding runtime) parenthesized.

Solving constraints with quantifiers is generally less efficient [7] than solving the simple Boolean constraints used to express memory constraints in the sequential mode. Further work could involve reducing or eliminating the use of quantifiers.

Concurrent mode: litmus test

The concurrency mode was designed as a tool to explore C11's relaxed memory model, along the lines of CPPMEM [12] but with the ability to handle a larger fragment of C. We summarize the runtimes to find *all* consistent executions in Table 5.2 for litmus tests written in C as well as hand-written litmus tests in Core (see Appendix B).

5.4 Related work

5.4.1 Bounded model checking

SAT-based bounded model checking is an efficient technique for verification of C programs [15]. CBMC [39], a bounded model checker targeted at embedded software, models a range of properties, including array bound violations and pointer safety, by unwinding loops and passing a bit-vector equation to a decision procedure. Our approach was inspired by CBMC, with a focus on using principled C semantics and exploring the C11 concurrency model.

CBMC uses *GOTO* programs as an *intermediate representation*, performing static analysis to instrument GOTO programs with assertions guarding for behaviours such as buffer overflow. Our intermediate representation, Core, is based on elaboration of a formal C semantics, which explicitly represents undefined behaviour as a primitive construct. By using Cerberus as a semantic interpretation for C, our model checker is designed for consistency with the ISO standard. For example, in CBMC, no error is detected in the following—reads of uninitialised values appear to be given a stable value, though these semantics are unclear in the ISO standard.

```
int main(void) {
  int x, y, z;
  if (x) {
    y = x;
    z = x;
    assert (y == z);
  }
}
```

CBMC supports concurrency and weak memory [1], however the support is targeted at embedded systems and does not address the C11 concurrency model. We contribute towards this area.

CBMC inspired various other BMCs, such as ESBMC [47], targeted for multi-threaded software, and `LLBMC` [46], which uses LLVM's intermediate representation as an intermediate language. Using Core as an intermediate language shares the advantage of having a more explicit syntax and semantics than C to aid logical encoding and the program being analysed being closer to the program actually executed. Using LLVM-IR as the intermediate language has the disadvantage that bugs "optimized away" by the compiler will not be detected, and does not focus on checking programs according to the ISO standard.

5.4.2 C11 concurrency

Mathematical formalisations of the C11 concurrency model have been developed by Batty et al. [11, 13] and others [49]. The Cppmem tool [12] enables the visualization and analysis of possible executions of litmus test programs. Cppmem exhaustively enumerates all program executions, and checks their validity against semantics specified in Isabelle/HOL [50]. Blanchette et al. [19] used the Nitpick tool [18] to efficiently explore the C/C++11 executions. These tools were intended to explore litmus test examples — we aim for an approach that handles a larger fragment of sequential C, while being easily adaptable for concurrent C.

Norris and Demsky presented CDSChecker [51], a stateless model checker that exhaustively explores all possible executions consistent with the C/C++11 memory model using a hand-written constraints solver and partial order reduction. We also use a constraints-based approach to model the C11 memory model, using an SMT solver as the backend as opposed to a hand-written constraint solver. This constraints-based approach is largely decoupled from the remainder of the model checker: one could potentially automatically generate constraints based on a memory model specification [2] to analyse different memory models. Finally, CDSChecker assumes a total order of operations in a thread (no unsequenced evaluation). We relax this assumption.

Chapter 6

Conclusion

This dissertation presented a principled approach to bounded model checking C based on an explicit, formal semantics. By first translating C source into an intermediate language based on a formalization of ISO C11 that makes explicit many subtle aspects of C such as undefined behaviours and uninitialized values, this approach decouples the often-ambiguous interpretation of C semantics with the implementation of the model checker and increases the clarity in what is being verified.

In addition to model checking single-threaded C programs, we implement an illustrative fragment of the C11 concurrency model based on an axiomatic formalization by Batty et al. [11, 13]. Apart from verifying the correctness of multi-threaded programs, our tool can be used to analyse and visualize the possible executions and weak memory behaviour of concurrent programs.

6.1 Future work

Along the lines of consistency with ISO C11, future work could involve increasing feature completeness and thoroughly handling the semantics of pointers and memory objects by implementing a more explicit C memory object model. The C memory object model is unclear in the ISO standard and in some aspects conflicts with de facto usage [43].

There is scope for exploring optimizations to generate satisfiability problems that are can be solved more efficiently by satisfiability solvers. For instance, constant propagation and common subexpression elimination could be used to reduce the size of the SMT problem, and incremental techniques used for lazy unrolling of loops. The efficient representation of consistent executions in the C11 concurrency model as an SMT problem is an area for future work.

Bibliography

[1] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *International Conference on Computer Aided Verification*, pages 141–157. Springer, 2013.

[2] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(2):7, 2014.

[3] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *International Journal on Software Tools for Technology Transfer*, 11(1):69–83, 2009.

[4] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A Decade of Software Model Checking with SLAM. *Commun. ACM*, 54(7):68–76, July 2011.

[5] Thomas Ball, Andreas Podelski, and Sriram Rajamani. Boolean and Cartesian abstraction for model checking C programs. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 268–283, 2001.

[6] Balyo, Tomáš and Heule, Marijn JH and Järvisalo, Matti. Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions. 2017.

[7] Clark Barrett, Morgan Deters, Leonardo De Moura, Albert Oliveras, and Aaron Stump. 6 years of SMT-COMP. *Journal of Automated Reasoning*, 50(3):243–277, 2013.

[8] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, volume 13, page 14, 2010.

[9] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.

[10] Clark W Barrett, David L Dill, and Jeremy R Levitt. A decision procedure for bit-vector arithmetic. In *Proceedings of the 35th annual Design Automation Conference*, pages 522–527. ACM, 1998.

[11] Mark Batty, Alastair F Donaldson, and John Wickerson. Overhauling SC atomics in C11 and OpenCL. *ACM SIGPLAN Notices*, 51(1):634–648, 2016.

[12] Mark Batty, Scott Owens, Jean Pichon, Susmit Sarkar, and Peter Sewell. Cppmem. Available at `http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/`.

[13] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. *ACM SIGPLAN Notices*, 46(1):55–66, 2011.

[14] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[15] Dirk Beyer. Software verification with validation of results. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 331–349. Springer, 2017.

[16] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, 1999.

[17] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Masahiro Fujita, and Yunshan Zhu. Symbolic model checking using sat procedures instead of bdds. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 317–320. ACM, 1999.

[18] Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *International conference on interactive theorem proving*, pages 131–146. Springer, 2010.

[19] Jasmin Christian Blanchette, Tjark Weber, Mark Batty, Scott Owens, and Susmit Sarkar. Nitpicking C++ Concurrency. In *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming*, PPDP '11, pages 113–124, New York, NY, USA, 2011. ACM.

[20] Hans-J Boehm and Sarita V Adve. Foundations of the C++ concurrency memory model. In *ACM SIGPLAN Notices*, volume 43, pages 68–78. ACM, 2008.

[21] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi Junttila, Peter Van Rossum, Stephan Schulz, and Roberto Sebastiani. An incremental and layered procedure for the satisfiability of linear arithmetic logic. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 317–333. Springer, 2005.

[22] Robert Brummayer and Armin Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177. Springer, 2009.

[23] Sebastian Burckhardt, Rajeev Alur, and Milo MK Martin. CheckFence:

checking consistency of concurrent data types on relaxed memory models. In *ACM SIGPLAN Notices*, volume 42, pages 12–21. ACM, 2007.

[24] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 73–88. ACM, 2001.

[25] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal methods in system design*, 19(1):7–34, 2001.

[26] Edmund Clarke and Daniel Kroening. Hardware Verification using ANSI-C Programs as a Reference . In *Proceedings of ASP-DAC 2003*, pages 308–311. IEEE Computer Society Press, January 2003.

[27] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

[28] Edmund Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of the 40th annual Design Automation Conference*, pages 368–371. ACM, 2003.

[29] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[30] Ernie Cohen, Michał Moskal, Stephan Tobies, and Wolfram Schulte. A precise yet efficient memory model for C. *Electronic Notes in Theoretical Computer Science*, 254:85–103, 2009.

[31] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. *IEEE Transactions on Software Engineering*, 38(4):957–974, 2012.

[32] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[33] Chucky Ellison and Grigore Rosu. An executable formal semantics of C with applications. *ACM SIGPLAN Notices*, 47(1):533–544, 2012.

[34] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. Defining the undefinedness of C. In *ACM SIGPLAN Notices*, volume 50, pages 336–345. ACM, 2015.

[35] ISO. ISO C Standard 1999. Technical report, 1999. ISO/IEC 9899:1999 draft.

[36] Franjo Ivancic, Ilya Shlyakhter, Aarti Gupta, Malay K Ganai, Vineet Kahlon, Chao Wang, and Zijiang Yang. Model checking C programs using F-Soft. In *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on*, pages 297–308. IEEE, 2005.

[37] ISO Jtc. SC22/WG14. ISO/IEC 9899: 2011. *Information technology—Programming languages—C. http://www. iso. org/iso/iso_ catalogue/catalogue_ tc/catalogue_ detail. htm*, 2011.

[38] Robbert Krebbers and Freek Wiedijk. A Formalization of the C99 Standard in HOL, Isabelle and Coq. In *International Conference on Intelligent Computer Mathematics*, pages 301–303. Springer, 2011.

[39] Daniel Kroening and Michael Tautschnig. CBMC–C bounded model checker. In *TACAS*, pages 389–391, 2014.

[40] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++ 11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 618–632. ACM, 2017.

[41] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *ACM SIGOPS Operating Systems Review*, 42(2):329–339, 2008.

[42] Paul E McKenney, Torvald Riegel, Jeff Preshing, Hans Boehm, Clark Nelson, and Olivier Giroux. Towards implementation and use of memory order consume. *ISO SC22 WG21 N*, 4321, 2015.

[43] Kayvan Memarian, Victor Gomes, and Peter Sewell. N2223: Clarifying the C Memory Object Model: Introduction to N2219 - N2222. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2223.htm`, 2018.

[44] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert NM Watson, and Peter Sewell. Into the depths of C: elaborating the de facto standards. In *ACM SIGPLAN Notices*, volume 51, pages 1–15. ACM, 2016.

[45] Kayvan Memarian and Peter Sewell. What is C in practice? (Cerberus survey), 2015. Website.

[46] Florian Merz, Stephan Falke, and Carsten Sinz. LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In *International Conference on Verified Software: Tools, Theories, Experiments*, pages 146–161. Springer, 2012.

[47] Jeremy Morse, Mikhail Ramalho, Lucas Cordeiro, Denis Nicole, and Bernd Fischer. ESBMC 1.22. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 405–407. Springer, 2014.

[48] Dominic P Mulligan, Scott Owens, Kathryn E Gray, Tom Ridge, and Peter Sewell. Lem: reusable engineering of real-world semantics. In *ACM SIGPLAN Notices*, volume 49, pages 175–188. ACM, 2014.

[49] Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. An operational

semantics for C/C++11 concurrency. Draft available at `www.cl.cam.ac.uk/~pes20/cerberus`, 2016.

[50] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[51] Brian Norris and Brian Demsky. CDSchecker: checking concurrent data structures written with C/C++ atomics. In *ACM SIGPLAN Notices*, volume 48, pages 131–150. ACM, 2013.

[52] Ishai Rabinovitz and Orna Grumberg. Bounded model checking of concurrent programs. In *Computer Aided Verification*, pages 319–325. Springer, 2005.

[53] Bastian Schlich and Stefan Kowalewski. Model checking C source code for embedded systems. *International Journal on Software Tools for Technology Transfer*, 11(3):187–202, Jul 2009.

[54] Shinichi Shiraishi, Veena Mohan, and Hemalatha Marimuthu. Test suites for benchmarks of static analysis tools. In *Software Reliability Engineering Workshops (ISSREW), 2015 IEEE International Symposium on*, pages 12–15. IEEE, 2015.

[55] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. Undefined behavior: what happened to my code? In *Proceedings of the Asia-Pacific Workshop on Systems*, page 9. ACM, 2012.

[56] Xi Wang, Nickolai Zeldovich, M Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 260–275. ACM, 2013.

[57] WG14. Defect report 260. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_260.htm`, 2004.

[58] WG14. Defect report 451. `http://www.open-std.org/Jtc1/sc22/WG14/www/docs/dr_451.htm`, 2013.

Appendix A

Core integer conversion functions

This appendix contains the Core pure functions used to perform integer conversion as used in Cerberus. The functions are annotated with the their corresponding definition in the ISO standard.

```
fun is_representable (n: integer, ty: ctype): boolean :=
  Ivmin(ty) <= n /\ n <= Ivmax(ty)

-- see 5.1.2 from ISO/IEC 10967-1:1994(E)
fun wrapI(ty: ctype, n: integer) : integer :=
    let dlt: integer = Ivmax(ty) - Ivmin(ty) + 1 in
    let r: integer = n rem_f dlt in
    if r <= Ivmax(ty) then r
                      else r - dlt

fun catch_exceptional_condition (ty: ctype, n: integer) : integer :=
  if is_representable(n, ty) then n
  else undef(<<UB036_exceptional_condition>>)

fun conv_int (ty: ctype, n: integer): integer :=
  -- (STD 6.3.1.2#1) When any scalar value is converted to _Bool,
  -- the result is 0 if the value compares equal to 0;
  -- otherwise, the result is 1.
  if ty = !(_Bool) then
    if n = 0 then 0 else 1
  -- (STD 6.3.1.3#1) When a value with integer type is converted to
  -- another integer type other than _Bool, if the value can be represented
  -- by the new type, it is unchanged.
  else
    if is_representable(n, ty) then
      n
  -- (STD 6.3.1.3#2) Otherwise, if the new type is unsigned, the value
  -- is converted by repeatedly adding or subtracting one more than the
  -- maximum value that can be represented in the new type until the value
  -- is in the range of the new type.
    else
      if is_unsigned(ty) then
        wrapI(ty, n)
  -- (STD 6.3.1.3#3) Otherwise, the new type is signed and the value
  -- cannot be represented in it; either the result is
  -- implementation-defined or an implementation-defined signal is raised.
      else
        <Integer.conv_nonrepresentable_signed_integer>(ty, n)
```

Appendix B

Litmus tests

This appendix contains the litmus tests (taken from the Cerberus test suite and Cppmem) used to test and evaluate the model checker.

LB: Load buffering

The load buffering litmus test with acquire/release pairs. The values of `z1` and `z2` cannot both be 1. This program can return 0, 1, or 2. If the memory order were relaxed, the values of `z1` and `z2` can both be 1 (and the program can also return 3).

```
#include <stdatomic.h>

int main() {
  _Atomic(int) x=0, y=0;
  int z1, z2;
  {-{ { z1 = atomic_load_explicit(&x, memory_order_acquire);
        atomic_store_explicit(&y, 1, memory_order_release); }
  ||| { z2 = atomic_load_explicit(&y, memory_order_acquire);
        atomic_store_explicit(&x, 1, memory_order_release); }  }-};
  return z1 + (2 * z2);
}
```

MP: Message passing

Message passing of data held in the non-atomic variable x with release/acquire synchronisation on y. If the value of z1 is 1, the read must see the write and thus the value of z2 should also be 1. This program could return 1 or 2, but not 0.

```
#include <stdatomic.h>

int main(void) {
  int x = 0;
  _Atomic(int) y = 0;
  int z1, z2;
  {-{ { x = 1;
        atomic_store_explicit(&y, 1, memory_order_release); }
  ||| { z1 = atomic_load_explicit(&y, memory_order_acquire);
        if (z1 == 1)
          z2 = x;
        else
          z2 = 2; }  }-};
  return z2;
}
```

SB: Store buffering

The store buffering litmus test with release-acquire pairs. The reads can both see 0 in the same execution. This program can return 0, 1, 2, or 3. If the memory orders were instead SC, the reads would not be able to both see 0.

```
#include <stdatomic.h>

int main() {
  _Atomic(int) x=0, y=0;
  int z1, z2;
  {-{ { atomic_store_explicit(&y, 1, memory_order_release);
        z1 = atomic_load_explicit(&x, memory_order_acquire); }
  ||| { atomic_store_explicit(&x, 1, memory_order_release);
        z2 = atomic_load_explicit(&y, memory_order_acquire); }  }-};
  return z1 + (2 * z2);
}
```

WRC: Write to read causality

If `z1` and `z2` both read 1, then it is necessarily the case that `z3` also reads one. There are seven possible return values (`(z1,z2,z3)=(1,1,0)` is not allowed).

```
#include <stdatomic.h>

int main() {
  _Atomic(int) x = 0;
  _Atomic(int) y = 0;
  int z1; int z2; int z3;

  {-{ { atomic_store_explicit(&x, 1, memory_order_release); }
  ||| { z1 = atomic_load_explicit(&x, memory_order_acquire);
        atomic_store_explicit(&y, 1, memory_order_release); }
  ||| { z2 = atomic_load_explicit(&y, memory_order_acquire);
        z3 = atomic_load_explicit(&x, memory_order_acquire);
      }
  }-}
  return (z1 + 2 * (z2 + 2 * (z3)));
}
```

IRIW: Independent reads of independent writes

This tests whether the reading threads must see the writes to `x` and `y` in the same order. For release-acquire pairs, the writes can be reordered and there are 16 possible return values. For the sequentially-consistent memory order, the writes can not be reordered and there are 15 possible return values (`(z1,z2,z3,z4) = (1,0,0,1)`) is not allowed.

```
#include <stdatomic.h>

int main(void) {
  _Atomic(int) x = 0;
  _Atomic(int) y = 0;
  int z1; int z2; int z3; int z4;
  {-{ atomic_store_explicit(&x, 1, memory_order_release);
  ||| atomic_store_explicit(&y, 1, memory_order_release);
  ||| { z1 = atomic_load_explicit(&x, memory_order_acquire);
        z2 = atomic_load_explicit(&y, memory_order_acquire); }
  ||| { z3 = atomic_load_explicit(&y, memory_order_acquire);
        z4 = atomic_load_explicit(&x, memory_order_acquire); }
  }-};
  return (z1 + 2 * (z2 + 2 * (z3 + 2 * z4))); }
```