The Right Kind of Non-Determinism: Using Concurrency to Verify C Programs with Underspecified Semantics

Eduard Kamburjan
University of Oslo, Oslo, Norway
eduard@ifi.uio.no

Nathan Wasser
Sharpmind, Frankfurt, Germany
nate@sharpmind.de

We present a novel and well automatable approach to formal verification of C programs with underspecified semantics, i.e., a language semantics that leaves open the order of certain evaluations. First, we reduce this problem to non-determinism of concurrent systems, automatically extracting a distributed Active Object model from underspecified, sequential C code. This translation process provides a fully formal semantics for the considered C subset. In the extracted model every non-deterministic choice corresponds to one possible evaluation order. This step also automatically translates specifications in the ANSI/ISO C Specification Language (ACSL) into method contracts and object invariants for Active Objects. We then perform verification on the specified Active Objects model, using the `Crowbar` theorem prover, which verifies the extracted model with respect to the translated specification and ensures the original property of the C code for all possible evaluation orders. By using model extraction, we can use standard tools, without designing a new complex program logic to deal with underspecification. The case study used is highly underspecified and cannot be handled correctly by existing tools for C.

1 Introduction

Verification of programs relies on the availability of a formal, or at least a formalizable, semantics of the used programming language. However, the semantics of mainstream programming languages contain challenges that require special attention from programmers and verification tools alike.

In this work we consider the semantics of the C language, which in addition to fully specified behavior contains *undefined*, *unspecified* and *implementation defined* behavior: these semantics describe not exactly what should happen, but leave crucial decisions to the implementing compiler and/or the runtime environment. Our focus here is on the unspecified evaluation order within the C standard, which we refer to as *underspecified*. Importantly, the semantics for underspecified behavior is not *undefined*, as the semantics limits the possible choices. This is not merely a fringe case, but is observable already in natural and small programs. Consider the C program in Fig. 1. The C99 standard [23] does not specify the order of evaluation of the subexpressions in the addition.[1] Indeed, the two main compilers for C return different values: `gcc` 7.4.0 returns 2 (evaluating the second summand first), `clang` 6.0.0 returns 1 (evaluating the first summand first). The reason is that `gcc` uses a stack-based translation of expressions, while `clang` uses a queue-based one.

Verification of underspecified C code is still an open problem and merely *fixing* the choice is not enough for verification: As the semantics is underspecified, compilers are not required to be consistent in their choice *even during the run of a single program* and optimizations are not obligated to preserve the choice of the compiler.

[1] This unspecified evaluation order is also prevalent in other C standards.

C. Aubert, C. Di Giusto, L. Safina & A. Scalas (Eds.):
15th Interaction and Concurrency Experience (ICE 2022)
EPTCS 365, 2022, pp. 1–16, doi:10.4204/EPTCS.365.1

```c
int x;
int id_set_x(int val){
  x=1;
  return val;}
int main(void){
  x=0;
  return x + id_set_x(1);}
```

Figure 1: Addition with side-effect.

This effect is further amplified from a software engineering perspective, when program equivalence becomes a problem: For one, changing, updating the compiler, or indeed barely changing its parameters may result in different program behavior. For another, reengineering legacy software, a critical activity to, e.g., enable parallelization [18] cannot rely on analyses proving functional equivalence, if these analyses are not considering underspecification. Before attempting to prove program equivalence, one must be able to reason about functional behavior of programs in a language with underspecified semantics.

Approach. At the core of this work is the idea to transform non-determinism in sequential programs arrising due to *underspecification* to non-determinism due to *concurrency* and then use tools to specify and verify concurrent behavior, which are more advanced and investigated in more detail. Each possible evaluation order is one possible interleaving order.

More precisely, this work presents an approach to *automatically* verify functional behavior of C programs with underspecified semantics, which is based on reducing *underspecification* to *non-determinism* in a fully specified language: We are able to verify functional properties of C programs without undefined behavior with respect to every possible standard-compliant semantics. In this work we build upon the model-extraction approach by Wasser et al. [37] for a subset of the C language and give an *implemented* system that verifies the functional behavior of the extracted model. The extracted model gives a *fully formal* and analyzable semantics for C in terms of an Active Object framework.

We translate C code into an *Active Objects* language [9] and regard sequential C programs as parallel programs, in which the non-determinism arises from parallelism and not from underspecified semantics. Conceptually, this is a rare case where a problem of *sequential* programs is transformed to a problem of *parallel* programs, because the support for analysis of parallel systems is better than the support for reasoning about underspecified semantics.

For Active Objects there are program logics [26] that enable modular reasoning and we are able to employ method contracts for asynchronous calls [27]. The expected behavior under all possible semantics is annotated with ACSL [8] and automatically translated into cooperative contracts and object invariants of Active Objects. Using this approach we give a case study to verify that a highly underspecified recursive function that computes the nth Fibonacci number in one semantics returns a value between 1 and the nth Fibonacci number in every standard-adhering semantics.

Contributions. Our contributions are (1) an implemented approach to *automatically* verify functional behavior of C programs with underspecified semantics, and a deductive verification case study of underspecified C code which is (2) the biggest verification case study of such code that cannot be handled by existing approaches (see next section) (3) the biggest deductive verification case study for Active Objects (in lines of code) to date. The case study can be proven *fully automatically*. Additionally to the conceptual approach and case study, we also contribute a translation of ACSL specifications for C into BPL specifiations for ABS.

State-of-the-Art. Underspecified (and to a lesser degree undefined) semantics are a rarely approached challenge for deductive verification. Here, we review the tools that consider these kinds of semantics.

Frama-C [13] can find (some) *undefined* behavior related to read-write or write-write accesses *between* sequence points. However, it does not recognize *unspecified* behavior when these accesses occur *indeterminately sequenced* as in our examples here, instead only examining a single fixed evaluation order[2] [11, p.40]. Further, while most of ACSL is utilized in Frama-C, this does not include global invariants, which we are able to handle. Additionally, new tools must be built specifically for the C intermediate representation only used within Frama-C, while our approach can profit from all tools available for ABS, which has included so far model checking, simulation, deadlock analysis and deductive verification. RV-Match [1]—based on C semantics formalized [2, 19] in the K framework [3, 36]—is able to find (some) *undefined* and *implementation defined* behavior in C programs, but like Frama-C chooses only a single evaluation order when faced with *underspecified* behavior. This in turn prevents both from finding undesired behavior that is only obvious when a different evaluation order is chosen. While our approach currently works only with an admittedly smaller subset of C containing underspecification than that allowed in RV-Match and Frama-C, it faithfully considers all possible evaluation paths allowed by the standard. Cerberus [4, 33] is an analysis tool for undefined and underspecified behavior; however, it cannot utilize any specifications and its treatment of unspecified evaluation order of *side effects* does not match the C standard, as demonstrated in [37]. The separation logic system of Frumin et al. [17], based on small-step semantics in Coq [30] correctly treats underspecification. They give a formal system to verify a program in their toy language λMC and check effects of underspecified behavior with a modified separation logic. In contrast to the subset of C we consider, λMC is emphatically *not* a subset of C and is described as merely a C-style language[3]. Verification of *any* C program therefore requires manual translation into an equivalent λMC program and manual specification of the λMC program in Coq. Our model-extraction based approach is fully automated, can be used with standard program logics and analyses for Active Objects and does not rely on complex rule modifications to handle underspecified behavior. We stress that this automation includes the verification, which needs not be performed by the user in an interactive prover such as Coq [5].

Holzmann and Smith [22] attempt to reuse the SPIN model checker by extracting Promela code from a C program. However, their approach requires *manual* translation/adjustment (flattening) of the underspecified parts. Furthermore, Promela/SPIN only support model checking and cannot be applied to unbounded inputs. Concerning semantics, several formalizations [16, 34, 35] of the C semantics deal with underspecified evaluation order without giving a reasoning system.

To conclude the overview of the state-of-the-art, there is no satisfying approach to verify underspecified C code and the partial approaches are not suited for automatization.

Structure. In Sec. 2 we investigate the program in Fig. 1 in more detail. In Sec. 3 we give preliminaries: the basics of ABS [24], the Active Object language used, and its contracts. In Sec. 4 we describe the model-extraction, which we then use in Sec. 5 to verify the Fibonacci case study. We conclude in Section 6. The accompanying technical report with formal details, proofs and a link to the implementation is not referred to for the double-blind review.

[2] E.g., value analysis in Frama-C claims that the program in Fig. 1 can only return 2.

[3] Even this is debatable, but underspecified C-style behavior is present.

2 Overview over Workflow

Before we introduce the used systems, we illustrate our approach using the code in Fig. 2, which adds ACSL specifications to the previous example. The strong global invariant specifies a condition that must hold at every point during execution, while the requires/ensures clauses are standard pre/postconditions.

Specified C-code is translated into specified ABS-code. ABS is object-oriented and uses the following concurrency model: (1) An object cannot access the fields of another object. (2) Every method call is asynchronous (i.e., does not block the caller) and returns a future. A future can be used to synchronize on the called method and read its eventual return value. (3) Only one process is active per object and a process can only be interrupted when executing an **await** `g` statement. An **await** `g` statement waits until all futures in the guard `g` are resolved, i.e., their process has terminated. There are no global variables and for specification, ABS supports object invariants and method contracts.

The code in Fig. 3 shows a (prettified) part of the translation of Fig. 2. The global variables are handled by a special (singleton) class `Global`. In `Global`, each global variable is a field and the global invariant becomes the object invariant of this class. Similarly, the global invariant is also added as pre/postcondition to the setter and getter method handling the fields.

Each C-function `f` is translated into an ABS-class `C_f` and an interface `I_f` with a `call` method that models its execution. The function contract of `id_set_x` becomes the method contract of `I_id_set_x.call`. We only show the translation of **main** in detail. Again, the function contract becomes the method contract of `call`. The other methods in the class `C_main` model memory accesses to global variable `x`, calling function `id_set_x` and addition with the `+` operator.

The `call` method is a translation of the **main** function. It first sets `x` to 0 and than waits for this operation to finish — the **await** at line 21 models synchronization at the sequence point `;`. The next three lines translate the addition operation and contain *no* **await**, because the C-expression contains no sequence point. The two calls to model evaluation of the subexpressions are called in one order, but may be executed in a different one.

The method `op_plus_fut_fut` models evaluation of the addition expression. It takes two futures, i.e., two references to *yet unfinished executions*. It then synchronizes with both of them, i.e., it waits until *both* are resolved (line 34) and then adds the corresponding return values. It depends on the global scheduling which method is executed first and therefore whether the read triggered in `C_main` or the write in `C_id_set_x` takes place on `Global` first. Note that the specification of `C_main` is also automatically derived from the ACSL specification. The translated model can now be passed to the `Crowbar` verification system, which checks that the code adheres to its specification. It indeed does so and, as expected, fails to close the proof if the specification is wrong, i.e., if the results is specifed as only 1 or only 2.

```
int x; //@ strong global invariant x == 0 || x == 1;
int id_set_x(int val)
/*@ requires val == 1; ensures \result == 1; @*/ {
  x=1; return val;}
int main(void)
/*@ ensures \result == 1 || \result == 2; @*/ {
  x=0; return x + id_set_x(1);}
```

Figure 2: Specified addition with side-effect.

```
[Spec:ObjInv(this.x == 0||this.x == 1)]
class Global implements Global {
 Int x = 0;
 [Spec:Ensures(result == 0||result == 1)]
 Int get_x() { return this.x; }
 [Spec:Requires(value == 0||value == 1)]
 Unit set_x(Int value) { this.x = value; }
}
class C_id_set_x(Global global)
      implements I_id_set_x {
 [Spec: Requires( val == 1 )]
 [Spec: Ensures( result == 1 )]
 Int call(Int val){...}// executes id_set_x(val)
 ... }
class C_main(Global global)
      implements I_main {
 [Spec:Ensures(result == 1||result == 2)]
 Int call() { // executes main()
  Fut<Unit> tmp_4 =
    this!set_global_x_val(0); // sets x to 0
  await tmp_4?; // introduces sequence point ‘‘;’’
  Fut<Int> tmp_5 =
    this!get_global_x(); // reads x
  Fut<Int> tmp_6 =
    this!call_id_set_x_val_0(1);//calls id_set_x
  Fut<Int> tmp_7 =
    this!op_plus_fut_fut(tmp_5, tmp_6);//add
  await tmp_7?; // introduces sequence point ‘‘;’’
  return tmp_7.get; // returns result of addition
 }
 [Spec: Ensures(valueOf(fut_arg1) + valueOf(fut_arg2) == result)]
 Int op_plus_fut_fut(Fut<Int> fut_arg1,
                     Fut<Int> fut_arg2) {
  await fut_arg1? & fut_arg2?;
  Int arg1 = fut_arg1.get;
  Int arg2 = fut_arg2.get;
  return ( arg1 + arg2 );
 }
 ... }
```

Figure 3: Partial translation of Fig. 2.

3 Active Objects and Their Verification

In this section we give the preliminaries for our work: the ABS language and cooperative contracts. For space reasons, we refrain from introducing the full formalisms and refer to [26] for a full definition of the underlying program logic and to [27] for a definition of the used ABS semantics and cooperative contracts. We stress, however, that the approach is fully formal.

ABS [24] is an executable, object-oriented modeling languages based on Active Objects [9], designed to model and analyze distributed systems. It has been applied to model a wide range of concurrent software systems, such as cloud-based services [14, 31], YARN [32] or memory systems [28].

Overview. ABS syntax is largely based on Java and we refrain from describing the full language here. Instead, we introduce ABS in an example-driven way to demonstrate its concurrency model and formal semantics. The main features of the concurrency model can be summarized with the points below:

Strong Encapsulation. Every object is strongly encapsulated at runtime, such that no other object can access its fields, not even objects of the same class.

Asynchronous Calls with Futures. The ABS language combines actors [21] with futures [6]. Each method call is asynchronous and generates a future. Futures can be passed around and are used to synchronize on the process generated by the call. Once the called process terminates, its future is *resolved* and the return value can be retrieved. We say that the process *computes* its future.

Cooperative Scheduling. At every point in time, at most one process is active in an object. Active Objects are preemption-free: A running process cannot be interrupted unless it *explicitly* releases control over the object. This is done either by termination with a **return** statement or with an **await** `g` statement that waits until guard `g` holds. A guard polls a set of futures and holds iff all futures in it are resolved.

These features ensure that a process has exclusive control over the heap memory of its object between syntactically marked statements. This vastly simplifies deductive verification, as between such statements techniques from sequential program verification carry over directly.

Example 1. *As the extracted models from C code are rather unintuitive, we demonstrate the concurrency model of ABS with a more natural program.*

Fig. 4 gives an ABS model with two objects that folds some binary operation over three numbers: one object that performs the operation and a second object that performs the folding. Interface `Fold` *defines an interface for the fold. Lines 2 and 3 give the specification, which we discuss in more detail below. Here, we specify that the input values must be positive (*`Requires`*) and that the result is positive (*`Ensures`*). Interface* `Comp` *specifies a single method, which performs some operation that also operates only on positive numbers. Class* `FoldC` *implements the folding and has a field* `comp` *that points to a* `Comp` *instance. We specify that the field is initialized with a non-null value (*`Requires`*) and stays non-null (*`ObjInv`*). It has a field* `last` *to store the intermediate result. ABS uses a main block to initialize the system, which here creates one instance of each class, starts two* `fold`*-processes and synchronizes on both. There is no await in the class – the processes executing* `C.fold` *do not overlap, so the value of* `last` *cannot change before it is returned and it is safe to save the intermediate value in this field.*

Cooperative Method Contracts. Here, we give the used fragment of the specification language for ABS: cooperative method contracts [27] and object invariants for Active Objects [15]. We recap the Behavioral Program Logic [26] used to verify cooperative method contracts.

Cooperative Method Contracts use two kinds of preconditions for methods: *parameter preconditions*, which describe the expected parameters; and *heap preconditions*, which additionally describe the class fields. Splitting the precondition is necessary, because the parameters are controlled by the *caller process* (and must be guaranteed by the caller), while the fields are controlled by the last active process in the *callee object* (and must be guaranteed by this process). There are also two postconditions: the heap postcondition defines the final state upon termination of the method in terms of its fields and local variables plus a special program variable `result` for the return value; the parameter postcondition defines the return value in terms of the parameters. The parameter postcondition can be used upon reading from the future if the call parameters are known.

We also use object invariants, which must hold at every point a method loses or regains control over the object: at method start, termination and **await** statements. The initial state of classes is specified with *creation conditions*.

```
interface Fold {
  [Spec: Requires(a>0 && b>0 && c>0)]
  [Spec: Ensures(result>0)]
  Int fold(Int a, Int b, Int c);
}
interface Comp {
  [Spec: Requires(a>0 && b>0)]
  [Spec: Ensures(result>0)]
  Int op(Int a, Int b);
}
class CompC implements Comp { ... }
[Spec: Requires(comp != null)]
[Spec: ObjInv(comp != null)]
class FoldC(Comp comp, Int last)
      implements Fold{
  Int fold(Int a, Int b, Int c){
    Fut<Int> f = comp!op(a, b); last = f.get;
    f = comp!op(last, c); last = f.get;
    return last;
  }
}
{ Comp a = new CompC();
  Fold c = new FoldC(a,0);
  Fut<Int> f1 = c!fold(1,2,5);
  Fut<Int> f2 = c!fold(1,2,4);
  await f1? & f2?; }
```

Figure 4: Simple ABS Model, slightly beautified.

Specification. Method signatures in interfaces may be annotated with parameter preconditions of the form `[Spec:Requires(e)]` and postconditions (`[Spec:Ensures(e)]`), where `e` is an expression of Boolean type. Similarly, method implementations in classes may be annotated with heap pre- and postconditions. A heap precondition that could be a parameter precondition is automatically transformed. Classes may be annotated with object invariants `[Spec: ObjInv(e)]` and creation conditions `[Spec: Requires(e)]`. Loops may be annotated with loop invariants `[Spec: WhileInv(e)]`. The specifications in Fig. 4 are explained in Example 1.

Full cooperative contracts have mechanisms to specify and verify **await** statements with suspension contracts and **get** statements with resolving contracts [27]. Similarly, so called *context sets* [27] are used to specify and analyze the heap preconditions. As neither heap preconditions nor suspension or resolving contracts are used by the extracted models, we refrain from introducing them in detail.

Verification `Crowbar` [29] is a verification system for `ABS` that implements symbolic execution (SE) i.e., the step-wise execution of statements to generate a set of first-order logic formulas. Validity of all generated formulas implies safety of the method. The resulting formulas are output in SMT-LIB [7] format and passed to solvers such as Z3.

Additionally to verifying cooperative contracts, `Crowbar` implements a lightweight deadlock checker for ABS that contrary to existing deadlock checkers for ABS [25, 20], requires no main block: The structural deadlock analysis deduces which methods cannot be part of a deadlock *for any program*: A deadlock is a cycle of dependencies caused by future (and condition) synchronizations [25] and is analyzed in terms of cycles in dependency graphs between synchronizations, objects and methods. Any method that contains no synchronization cannot be part of any dependency cycle, it is *structurally deadlock-free*.

Similarly, all methods that only call deadlock-free methods and synchronize only on their futures are not part of any deadlock.

Example 2. *Consider Ex. 1. If the implementation of* `CompC.op` *contains no blocks or call, e.g., the statement* **return** `a*b`*, then we can show deadlock freedom.*

`CompC.op` *is structurally deadlock-free: it contains no synchronization or suspension.* `C.fold` *depends only on* `CompC.op` *and is thus not part of any deadlock.*

4 Extraction of Annotated Model

In order to extract an ABS model annotated with appropriate specifications from a (specified) C program, we extend the approach from [37] (which extracts a non-deterministic Active Objects model from C code containing underspecified behavior) by automatically generating some specifications which are sound by construction and generating all other specifications by translation of the specifications in the underlying C program. In order to translate ACSL function contracts into method contracts it was also required to slightly change the manner in which function parameters were modeled, from parameters of the class to parameters of the `call` method within the class. Otherwise, simple functional properties would have required reasoning about heap properties.

ACSL

The *ANSI/ISO C Specification Language (ACSL)* [8] is a behavioral specification language for C programs, used by the state-of-the-art *Frama-C* [13] tool suite. ACSL can be used to specify function contracts (pre- and postconditions), data invariants over global variables and some further constructs, such as loop invariants, statement contracts (pre- and postconditions for a single statement or block of statements), assertions or ghost code.

Function contracts consist of a `requires` clause for the precondition and an `ensures` clause for the postcondition. Both clauses can be simple C expressions of arithmetic type[4], with the postcondition allowed to contain `\result` to refer to the return value. Additionally, an `assigns` clause to specify which locations may be accessed can be given. We ignore `assigns` clauses for now as they are not directly relevant for underspecified semantics.

ACSL allows two types of data invariants on global variables: 1. *strong global invariants*, which hold at all times; and 2. *weak global invariants*, which hold before and after each execution of a function call and can thus equivalently be added as a requires and ensures clause to all functions. We therefore focus here only on strong global invariants, in particular as these cannot be easily dealt with in Frama-C. Furthermore, we restrict strong global invariants to properties about single variables and thus exclude relational properties.

4.1 From C Code to ABS (`C2ABS`)

`C2ABS` [37] is an Eclipse plugin which extracts an ABS model from a C program. Here we describe how this extraction takes place. In the next subsection we describe the novel extension of this model extraction: synthesizing specification annotations for the extracted model. Table 1 details how C concepts are translated into ABS. The basic idea is to have one Active Object which models access to global variables and further model each executed function call as its own Active Object. Within these function

[4]Full ACSL allows more operators, which we ignore for now.

C	ABS
Top-level declarations	
Global variables	Class `Global` with methods to get/set variable values
definition of function `f`	class `C_f` with parameter `global`
Execution of function `f`	Execution of `call` method on object of class `C_f`
Parameters and local variables	
const parameter	parameter of `call` method
non-const parameter	parameter of `call` method stored in field
const local variable	local variable
non-const local variable	field
Local const read	Direct variable/parameter access
Other (sub-)expressions	Methods awaiting parameters and:
global read/write	synchronous call to `global` object (write is side effect)
local non-const read/write	get/set value of field
C built-in operators $\oplus$	return result of performing $\oplus$
invocation of function `f`	await side effects, create `C_f` object, make synchronous call to method `call` of object
Unspecified evaluation order	Asynchronous method calls to **this** object
Sequence points	**await** statements

Table 1: Translation of C concepts into ABS

call objects each (sub)expression being evaluated is modeled as an asynchronous method call to itself with **await** statements modeling *sequence points*: the point between evaluation of all arguments and side effects of a function call, and the call itself; the semicolon at the end of an expression statement; etc. Access to global variables is modeled by methods making blocking calls to the `global` object, while (potentially recursive) function calls are modeled by creating new Active Objects for the appropriate function and making blocking calls to these new objects.

Example 3. *Consider the function* **main** *in Fig. 1 and the statement* **return** `x + id_set_x(1);` *inside, where there is a sequence point between evaluation of the expression and returning from the function. The ABS class extracted is shown in Fig. 5, where the method* `call` *models function execution and lines 5-9 model the unspecified evaluation order of the the expression* `x + id_set_x(1)` *with the* **await** *at line 10 allowing non-deterministic choice in which order the methods to* **this** *are executed in. Once all futures have been resolved, the* **await** *regains control, modeling the sequence point before returning. The method* `call` *then returns the value of the addition. The method* `get_global_x` *models the memory access, by making a synchronous call*[5] *to the* `global` *parameter of the class, requesting the value of* `x`*. The method* `call_id_set_x_val_0` *models a call to the function* `id_set_x` *with an argument evaluated at compile time and zero side effects from evaluating its argument*[6]*. This is done by first creating a new* `C_id_set_x` *object with access to the same* `global` *object and then making a synchronous call to the* `call` *method of that object with the evaluated function arguments as parameters. Finally, the method* `op_plus_fut_fut` *models the addition of two subexpressions evaluated at runtime and therefore modeled as futures. First, the method awaits the resolution of its subexpressions, then returns the sum. While the three methods can be executed in arbitrary (and interleaving) order, the only visible difference depends on the order of* `get_global_x` *and* `call_id_set_x_val_0`*, as* `op_plus_fut_fut` *immediately awaits resolution of the*

[5] An asynchronous call to an object in another object immediately followed by a **get**.

[6] If the argument were a future or side effects (modeled as futures) were present, the method would immediately await resolution of all these futures.

```
class C_main(Global global)
      implements I_main {
 Int call() {
  ...
  Fut<Int> fut_x = this!get_global_x();
  Fut<Int> fut_set =
    this!call_id_set_x_val_0(1);
  Fut<Int> fut_add =
    this!op_plus_fut_fut(fut_x, fut_set);
  await fut_x? & fut_set? & fut_add;
  return fut_add.get;
 }
 Int get_global_x() {
  Fut<Int> f = global!get_x();
  return f.get;
 }
 Int call_id_set_x_val_0(Int arg1) {
  I_id_set_x o = new C_id_set_x(global);
  Fut<Int> f = o!call(arg1); return f.get;
 }
 Int op_plus_fut_fut(Fut<Int> fut_arg1,
                     Fut<Int> fut_arg2) {
  await fut_arg1? & fut_arg2?;
  Int arg1 = fut_arg1.get;
  Int arg2 = fut_arg2.get;
  return arg1 + arg2;
 }
}
```

Figure 5: Class `C_main` extracted from function **main** in Fig. 1

other two methods.

4.2 Automatically Synthesizing Specifications

Due to the automated nature in which function-modelling classes and helper methods are generated, we can synthesize some specifications directly. For others we require ACSL specification of the underlying C program.

Auto-generate specifications related to global object As each function-modelling class receives the `global` object as a parameter, uses it to access global variables and passes it on when instantiating any further function-modelling classes, we must (at least) specify that this class parameter (and field) is never **null**. To this end all function-modelling classes are specified with:

```
[Spec : Requires(global != null)]
[Spec : ObjInv(global != null)]
```

Auto-generate precise postconditions for operator methods C2ABS-generated methods from C built-in operators $\oplus$ all perform the same basic steps: await resolution of all future parameters and then return the result of performing $\oplus$ on the (resolved) parameters. Precise postcondition specifications for each of these methods can therefore be generated automatically, by ensuring that the result of the method is equal to the result of performing $\oplus$ on the (resolved) parameters. All C operator method declarations in interfaces are thus automatically annotated with appropriate postcondition specifications.

Example 4. *The interface* `I_main` *in the model extracted from the program in Fig. 1 contains the following annotated method declaration:*

```
[Spec : Ensures(valueof(fut_arg1) + valueof(fut_arg2) == result)]
Int op_plus_fut_fut(Fut<Int> fut_arg1,
                    Fut<Int> fut_arg2);
```

Translate ACSL requires/ensures function contracts ACSL requires/ensures clauses specify (relational) restrictions upon the function arguments and functional guarantees for the result. Following similar steps to those for extracting C expressions—simplified somewhat due to lack of side effects—these can be converted into pre- and postconditions of the `call` method in the interface modelling the function. Additionally, similar pre- and postconditions are added to the indirect call methods in any interfaces modelling functions calling the specified function. When an argument to an indirect call is a future value, the pre- and postconditions must be formulated to hold for the resolved argument.

Example 5. *Given the specified function* `id_set_x` *at line 3 in Fig. 2:*

```
2 int id_set_x(int val)
3 /*@ requires val == 1; ensures \result == 1; @*/ {
```

We annotate both the `call` *method in* `I_id_set_x` *and the* `call_id_set_x_val` *method in* `I_main` *with the following specifications:*

```
[Spec : Requires(val == 1)]
[Spec : Ensures(result == 1)]
```

Translate ACSL strong global invariants While a strong global invariant must hold at every point in the program, it suffices to *check* that it holds at program start and whenever the global variable is changed. The ACSL invariant is translated as above and added as an object invariant in the `Global` class and as preconditions on the argument of all setter methods for said variable. When the argument to indirect setters outside of `Global` is a future value, the precondition must be formulated to hold for the resolved argument. In order to *use* the invariant, we add postconditions to all getter methods for the variable.

Example 6. *Given the strong global invariant at line 1 in Fig. 2 that* `x == 0 || x == 1`*, the global state is modeled as the code in Fig. 6. Additionally,* `I_id_set_x` *and* `I_main` *contain the annotated method declarations in the lower code in Fig. 6.*

Use ABS functions in lieu of ACSL logic functions ACSL allows pure *logic functions* to be defined (inductively or axiomatically) and called in ACSL specifications. Translating these definitions is outside of the scope of this work and we therefore instead allow ABS functions to be called directly in ACSL specifications. If the ABS function is not inside the standard library, it must be defined inside an ACSL-style comment in the C program.

Scope The C Standard lists 52 cases of unspecified behavior [23, Annex. J.1]. However, most of these cases are not relevant to functional verification of runtime semantics, e.g., unspecified behavior of macros; or concern well-investigated elements outside of the considered language fragment, such as

```
interface Global {
 [Spec : Ensures(result == 0||result==1)]
 Int get_x();
 [Spec : Requires(arg == 0||arg == 1)]
 Unit set_x(Int arg);
}
[Spec : ObjInv(this.x == 0 || this.x == 1)]
class Global implements Global {
 Int x = 0;
 Int get_x() { return this.x; }
 Unit set_x(Int arg) {
  this.x = arg;
  return unit;
 }
}
```

```
[Spec:Requires(arg == 0 || arg == 1)]
Unit set_global_x_val(Int arg);
[Spec:Requires(valueof(fut_arg) == 0||valueof(fut_arg) == 1)]
Unit set_global_x_fut(Fut<Int> fut_arg);
[Spec:Ensures(result == 0||result == 1)]
Int get_global_x();
```

Figure 6: Example for translating strong global invariants.

floating points and string literals; or concern deprecated features of old compilers for rare hardware, such as the use of negative zeros in integer types. Our focus is therefore on those cases that touch on core aspects of the runtime semantics and are relevant for almost all programs: order of subexpression and side effect evaluation (except for some operators such as &&) [23, 6.5], of function argument evaluation [23, 6.5.2.2] and of evaluation of complex assignments [23, 6.5.16]. All these aspects can be handled by our approach and reduced to non-determinism of concurrent systems.

5 Case Study

Underspecified behavior lurks at almost every binary operation[7] and can have subtle effects in larger systems. To evaluate our verification approach, we use an extreme case of underspecification, investigating the C program[8] in Fig. 7 containing a function whose result heavily depends on unspecified evaluation order. The function in question is declared as `int one_to_fib(int n)` and should calculate a number between 1 and the nth Fibonacci number. The base cases are for inputs 1 and 2 (as well as all non-positive inputs), which return 1; as well as for input 3, which returns either 1 or 2 in the same manner as the program in Figure 1. Otherwise, `one_to_fib(n)` returns the sum of `one_to_fib(n-2)` and `one_to_fib(n-1)` with a potential decrement of 1 in the function `pred_or_id` ensuring that 1 is always a potential result, as otherwise $\{1,\dots,Fib(n-1)\}+\{1,\dots,Fib(n-2)\} = \{2,\dots,Fib(n)\}$.

Verification of this program is a challenging task due to the extensive non-determinism. In [37] the extracted model for this program was exhaustively checked for inputs up to 5, validating that all possible outputs (and no outputs outside this range) could be produced. Later experiments with an enhanced

[7] Underspecified behavior also lurks at many function calls.

[8] Adapted from an idea on Derek Jones's *The Shape of Code* blog at: `shape-of-code.coding-guidelines.com/2011/06/18/fibonacci-and-jit-compilers/`

```
1 //@ ABS def Int fib(Int n) = if n <= 2 then 1
2 //@                         else fib(n−1) + fib(n−2);
3
4 /*@ strong global invariant x == 0 || x == 1; @*/ int x;
5
6 //@ ensures \result == val;
7 int id_set_x(const int val)
8 { x=1; return val; }
9 //@ ensures \result == 1 || \result == 2;
10 int one_or_two(void) {
11     x=0;
12     return x + id_set_x(1);
13 }
14 //@ ensures \result == val – 1 || \result == val;
15 int pred_or_id(const int val) {
16     x=0;
17     return val - x + id_set_x(0);
18 }
19 //@ ensures \result >= 1 && \result <= fib(n);
20 int one_to_fib(const int n) {
21  if (n > 3)
22   return one_to_fib(n-2)
23          + pred_or_id(one_to_fib(n-1));
24  else if (n == 3) return one_or_two();
25  else return 1; }
```

Figure 7: Calculate a number between 1 and the `n`th Fibonacci number in C

model extraction process partially validated models for inputs up to 10. In this work we verify that no outputs outside of the range are produced for any (valid) inputs.[9] The annotated extracted model for this C program can be found in the technical report. The ABS function definition inside the ACSL-style specification in line 2 is copied verbatim into the model, the helper methods for `+` (used in lines 10, 15 and 23) and `-` (line 15) receive precise specifications, the strong global invariant on `x` at line 4 produces specifications throughout the model (`Global` interface and class, plus indirect getter and setter methods of other interfaces), while the `call` methods and their indirect callers are specified with translations of the contracts for the matching functions. As the program does not contain a **main** method and is not executable, so the model it produces is therefore also not executable: the main block in the extracted model is empty. As we are focused on proving a property of `one_to_fib` in general, rather than for a specific actual call, this non-executability is not a problem. This shows an additional strength of our approach, in that we can analyze *library* calls in isolation, rather than only being able to analyze a complete program. `Crowbar` can close all proof obligations of the extracted model *automatically*. Note that we prove the following *for all inputs* to `one_to_fib`.

Theorem 1. *The extracted model is safe with respect to its specification.*

In particular, the proof cannot be closed if we change the specification to express that `one_to_fib` returns a value from a smaller range.

[9] The semantics of the program are underspecified but *not* undefined.

Deadlock Freedom. Running `Crowbar` performs a simple analysis for structurally deadlock-free methods and returns all methods for which it cannot deduce it. For the extracted model it returns 9 such methods. These are the methods that take futures as parameters, which is not supported by the deadlock analysis in `Crowbar`, and methods depending on these methods. However, all futures that are passed as parameters are always futures of free methods. Thus we can state the following lemma, which is proven in the technical report.

Lemma 1. *The extracted model is deadlock free for every extractable main block.*

Applying State-of-the-Art Tools. As detailed in Sec. 1, other automatic tools cannot handle the example correctly. They either fix an evaluation order and can (wrongly) prove a stronger result, i.e., that the result is always the *n*th Fibonacci number (Frama-C, RV-match), do not support specification of global invariants of ACSL (Frama-C) or do not support verification at all(Cerberus). We do not compare our approach explicitly with the theory presented by Frumin et al. [17], which does treat underspecification correctly, but not for C and requires manual translation and manual specification of the translated program in the target formalism and an interactive proof.

6 Conclusion

We have demonstrated a novel approach combining model extraction with deductive verification of a distributed active objects model in order to verify C programs with underspecified behavior by reducing the non-determinism of underspecification to non-determinism of parallelism. We have extended the `C2ABS` tool—which already gives C a formal semantics in terms of Active Objects— to automatically translate a large subset of ACSL specifications into BPL specifications and implemented the `Crowbar` tool based on [26] in order to verify the specified model and analyze it for deadlock freedom. Using a complex case study that exemplifies the challenges for verification of underspecified programs we showed that our approach of model extraction and verification is *fully automatic*. We reused a standard logic and deadlock analysis for ABS and did not need special amendments for underspecified behavior after the extraction.

Future Work. For formalized parallelization of C code, we plan to integrate a formal, logic-based dependences analysis [10] and to consider further cases of underspecification of a larger fragment of C, e.g., in list initializers. The newest version of `C2ABS` uses different model extraction strategies [38] and we will investigate using `Crowbar` to verify these models as well. In cases where the input C program is not completely specified, we envisage generating the missing object invariants and method contracts automatically via counter-example guided refinement techniques [12] using the failed `Crowbar` proofs.

Finally, it is worth investigating how our model extraction approach compares to an explicit handling of underspecifiation by branching for every possible evaluation order.

Acknowledgements

This work was partially funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project and partially by the Research Council of Norway via `PeTWIN` (294600) and `SIRIUS` (237898).

References

[1] `https://runtimeverification.com/match/`.

[2] `https://github.com/kframework/c-semantics`.

[3] `http://kframework.org/`.

[4] `https://cerberus.cl.cam.ac.uk/`.

[5] `https://coq.inria.fr/`.

[6] Henry G. Baker & Carl E. Hewitt (1977): *The Incremental Garbage Collection of Processes*. In: *Proceeding of the Symposium on Artificial Intelligence Programming Languages*, *SIGPLAN Notices* 12, p. 11, doi:10.1145/872734.806932.

[7] Clark W. Barrett, Aaron Stump & Cesare Tinelli (2010): *The SMT-LIB Standard Version 2.0*. Https://homepage.cs.uiowa.edu/ tinelli/papers/BarST-SMT-10.pdf.

[8] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy & Virgile Prevosto (2018): *ACSL: ANSI/ISO C Specification Language Version 1.14*. `https://frama-c.com/acsl.html`.

[9] Frank S. de Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes & Albert Mingkun Yang (2017): *A Survey of Active Object Languages*. *ACM Comput. Surv.* 50(5), pp. 76:1–76:39, doi:10.1145/3122848.

[10] Richard Bubel, Reiner Hähnle & Asmae Heydari Tabar (2019): *A Program Logic for Dependence Analysis*. In: *IFM*, *LNCS* 11918, pp. 83–100, doi:10.1007/978-3-030-34968-4_5.

[11] David Bühler, Pascal Cuoq & Boris Yakobowski (2020): *Eva – The Evolved Value Analysis plug-in, Manual v21.1*. Available at `https://frama-c.com/download/frama-c-eva-manual.pdf`.

[12] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu & Helmut Veith (2000): *Counterexample-Guided Abstraction Refinement*. In E. Allen Emerson & Aravinda Prasad Sistla, editors: *CAV*, Springer, pp. 154–169, doi:10.1007/10722167_15.

[13] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles & Boris Yakobowski (2012): *Frama-C: A Software Analysis Perspective*. In: *SEFM*, Springer-Verlag, p. 233–247, doi:10.1007/978-3-642-33826-7_16.

[14] Crystal Chang Din, Richard Bubel, Reiner Hähnle, Elena Giachino, Cosimo Laneve & Michael Lienhardt (2015): *Deliverable D3.2 Verification of project FP7-610582 (ENVISAGE)*. Available at `http://www.envisage-project.eu`.

[15] Crystal Chang Din & Olaf Owe (2015): *Compositional reasoning about active objects with shared futures*. *Formal Asp. Comput.* 27(3), pp. 551–572, doi:10.1007/s00165-014-0322-y.

[16] Chucky Ellison & Grigore Rosu (2012): *An executable formal semantics of C with applications*. In: *POPL'12*, ACM, pp. 533–544, doi:10.1145/2103656.2103719.

[17] Dan Frumin, Léon Gondelman & Robbert Krebbers (2019): *Semi-automated Reasoning About Non-determinism in C Expressions*. In: *ESOP*, *LNCS* 11423, pp. 60–87, doi:10.1007/978-3-030-17184-1_3.

[18] Reiner Hähnle, Asmae Heydari Tabar, Arya Mazaheri, Mohammad Norouzi, Dominic Steinhöfel & Felix Wolf (2020): *Safer Parallelization*. In: *ISoLA (2)*, *Lecture Notes in Computer Science* 12477, Springer, pp. 117–137, doi:10.1007/978-3-030-61470-6_8.

[19] Chris Hathhorn, Chucky Ellison & Grigore Roşu (2015): *Defining the Undefinedness of C*. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, ACM, pp. 336–345, doi:10.1145/2813885.2737979.

[20] Ludovic Henrio, Cosimo Laneve & Vincenzo Mastandrea (2017): *Analysis of Synchronisations in Stateful Active Objects*. In Nadia Polikarpova & Steve Schneider, editors: *Integrated Formal Methods*, Springer International Publishing, Cham, pp. 195–210, doi:10.1007/978-3-319-66845-1_13.

[21] Carl Hewitt, Peter Bishop & Richard Steiger (1973): *A universal modular ACTOR formalism for artificial intelligence*. In: *IJCAI'73*, Morgan Kaufmann Publishers Inc., pp. 235–245. Available at `http://dl.acm.org/citation.cfm?id=1624775.1624804`.

[22] Gerard J. Holzmann & Margaret H. Smith (2002): *An Automated Verification Method for Distributed Systems Software Based on Model Extraction*. IEEE Trans. Software Eng. 28(4), pp. 364–377, doi:10.1109/TSE.2002.995426.

[23] ISO (1999): *ISO C Standard 1999*. Available at `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf`. ISO/IEC 9899:1999 draft.

[24] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte & Martin Steffen (2010): *ABS: A Core Language for Abstract Behavioral Specification*. In Bernhard K. Aichernig, Frank S. de Boer & Marcello M. Bonsangue, editors: *FMCO 2010*, *LNCS* 6957, Springer, pp. 142–164, doi:10.1007/978-3-642-25271-6_8.

[25] Eduard Kamburjan (2018): *Detecting Deadlocks in Formal System Models with Condition Synchronization*. *ECEASST* 76, doi:10.14279/tuj.eceasst.76.1070.

[26] Eduard Kamburjan (2019): *Behavioral Program Logic*. In: *TABLEAUX*, *LNCS* 11714, Springer, pp. 391–408, doi:10.1007/978-3-030-29026-9_22.

[27] Eduard Kamburjan, Crystal Chang Din, Reiner Hähnle & Einar Broch Johnsen (2020): *Behavioral Contracts for Cooperative Scheduling*, doi:10.1007/978-3-030-64354-6_4.

[28] Eduard Kamburjan & Reiner Hähnle (2018): *Prototyping Formal System Models with Active Objects*. In: *ICE*, *EPTCS* 279, pp. 52–67, doi:10.4204/EPTCS.279.7.

[29] Eduard Kamburjan, Marco Scaletta & Nils Rollshausen (2021): *Crowbar: Behavioral Symbolic Execution for Deductive Verification of Active Objects*. *CoRR* abs/2102.10127. Available at `https://arxiv.org/abs/2102.10127`.

[30] Robbert Krebbers (2014): *An operational and axiomatic semantics for non-determinism and sequence points in C*. In: *POPL'14*, ACM, pp. 101–112, doi:10.1145/2535838.2535878.

[31] Torgeir Lebesbye, Jacopo Mauro, Gianluca Turin & Ingrid Chieh Yu (2021): *Boreas - A Service Scheduler for Optimal Kubernetes Deployment*. In: *ICSOC*, *Lecture Notes in Computer Science* 13121, Springer, pp. 221–237, doi:10.1007/978-3-030-91431-8_14.

[32] Jia-Chun Lin, Ming-Chang Lee, Ingrid Chieh Yu & Einar Broch Johnsen (2020): *A configurable and executable model of Spark Streaming on Apache YARN*. International Journal of Grid and Utility Computing 11(2), pp. 185–195, doi:10.1504/IJGUC.2020.105531.

[33] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson & Peter Sewell (2016): *Into the depths of C: elaborating the de facto standards*. In: *37th PLDI*, ACM, pp. 1–15, doi:10.1145/2908080.2908081.

[34] Michael Norrish (1998): *C formalised in HOL*. Technical Report UCAM-CL-TR-453, University of Cambridge, Computer Laboratory. Available at `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-453.pdf`.

[35] Nikolaos S Papaspyrou (2001): *Denotational semantics of ANSI C*. Computer Standards & Interfaces 23(3), pp. 169–185, doi:10.1016/S0920-5489(01)00059-9.

[36] Grigore Roşu & Traian Florin Şerbănuţă (2010): *An Overview of the K Semantic Framework*. Journal of Logic and Algebraic Programming 79(6), pp. 397–434, doi:10.1016/j.jlap.2010.03.012.

[37] Nathan Wasser, Asmae Heydari Tabar & Reiner Hähnle (2019): *Modeling Non-deterministic C Code with Active Objects*. In: *FSEN*, *LNCS* 11761, pp. 213–227, doi:10.1007/978-3-030-34968-4_5.

[38] Nathan Wasser, Asmae Heydari Tabar & Reiner Hähnle (2021): *Automated model extraction: From non-deterministic C code to active objects*. Science of Computer Programming 204, p. 102597, doi:10.1016/j.scico.2020.102597.