Modeling Non-deterministic C Code with Active Objects

Nathan Wasser, Asmae Heydari Tabar, Reiner Hahnle

Modeling Non-deterministic C Code with Active Objects*

Nathan Wasser, Asmae Heydari Tabar, and Reiner Hähnle[0000−0001−8000−7613]

Technische Universität Darmstadt, Department of Computer Science,
64289 Darmstadt, Germany
{wasser,heydaritabar,haehnle}@cs.tu-darmstadt.de
https://www.informatik.tu-darmstadt.de/se

Abstract. Cheap and ubiquitous availability of multi-processor hardware provides a strong incentive to parallelize existing software. We aim to annotate existing sequential applications written in C with OpenMP directives that can be processed by compilers on high performance parallel computers. We adopt a model-based approach, where from sequential C-code a software model is extracted in a largely automatic fashion. The target is the modeling language ABS (Abstract Behavioral Specification), an active objects-language with formal semantics. ABS has been designed to be statically analyzable. We focus on the first stages of model-based parallelization: model extraction and validation. We define a behavior-preserving, fully automatic translation of a large fragment of sequential C that explicitly renders all possible execution sequences, then use automated test case generation to produce validation test cases.

Keywords: Model extraction · Model validation · Parallelization.

1 Introduction

The context of this paper is a project[1] concerned with the adaptation of legacy software due to changed requirements and technical advances. Specifically, cheap and ubiquitous availability of multi-processor hardware provides a strong incentive to parallelize existing software. In the long term we aim to annotate existing sequential applications written in C with OpenMP directives [14].

We adopt a *model-based* approach as illustrated in Fig. 1. From given sequential C-code a software model is extracted in a largely automatic fashion. The target is the modelling language ABS (Abstract Behavioral Specification) [7], an active objects-language [4] with formal semantics [9]. ABS is formally defined, free from ambiguity, and it has been designed to be statically analyzable [17]. Therefore, it is possible to use software tools for exhibiting opportunities for parallelization and to generate suitable directives. In this paper we focus on the first stage: *model extraction* and *model validation*.

* This work was funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project.

[1] Software-Factory 4.0, see http://www.software-factory-4-0.de/.

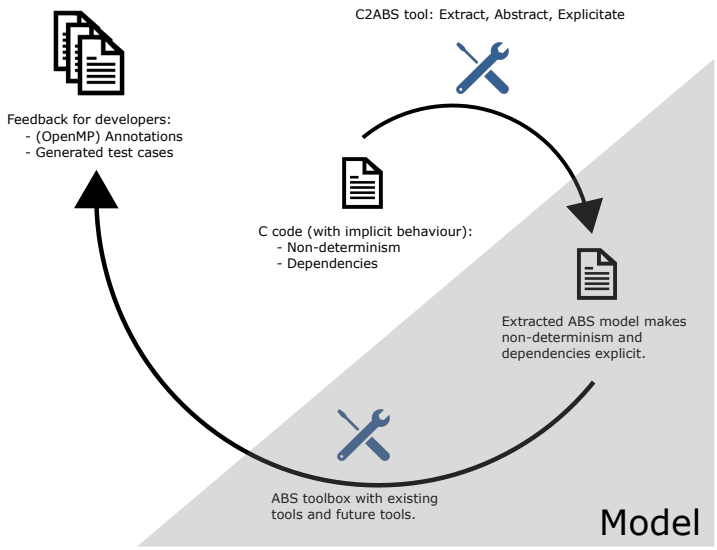

Fig. 1. Model-based parallelization

While abstraction of source code to a modelling language is a standard ingredient of many model checking tool chains (for example, [8]), here we pursue different goals: 1. we don't abstract away from behavior, but make non-deterministic behavior (a consequence of underspecification in C) explicit in the model; 2. non-deterministic execution sequences[2] and variable dependencies are precisely represented in a formal language and amenable to symbolic analysis; 3. the formal model with explicit non-determinism makes it possible to *validate* the model via automatically generated test cases and to give feedback to the author of the C-code about possibly unintended ambiguity.

Our main contributions are: 1. A behavior-preserving, fully automatic translation of a large fragment of sequential C that explicitly renders all possible execution sequences in ABS, and 2. application and adaptation of the ABS test case generator SYCO [3] to generate validation test cases. In Sect. 2 we define the C-fragment that we currently support and introduce a running example. In Sect. 3 we show how we extract an outline of the model based on the declarations of global variables and functions; how we extend the function-modelling classes with required helper methods in order to make non-determinism contained in C expressions within the function definition explicit in the model; and finally how we model the execution of the function call itself. In Sect. 4 we report on

[2] Most C compilers decide the evaluation order of subexpressions and side-effects at compile-time, but the C standard does not require this, so deciding at runtime is possible. Hence, this underspecified behavior is classified here as non-deterministic.

experiments performed with our tool, for model validation. Finally, we discuss related and future work in Sect. 5 and conclude in Sect. 6.

2 C-Fragment and Active Object Language

2.1 Input Language: C

The supported C-fragment is closely related to MISRA-C [12], a C subset widely used in embedded systems. We don't cover all features of MISRA-C (yet) which is not caused by principal limitations, but down to the fact that our tool is a research prototype rather than a commercial product. More importantly, in contrast to MISRA-C we explicitly *permit* non-deterministic computations and programs with underspecified C semantics that may lead to different behavior. In fact, our goal is to make such behavior explicit, so that it can be analyzed and taken into account in the parallelization stage.

Fig. 2 contains the subset of C we use as an input language to explain our model extraction process.[3] A program is a list of declarations containing a function definition for main(). In addition to the assignment operator =, we restrict ourselves to the operator set $\{ +, -, *, ==, !=, >, >=, <, <= \}$. The semantics of a program from this subset of C are the same as the semantics of the C99 standard for the given program. In particular the unspecified evaluation order for side effects of assignments, as well as evaluation of arguments and subarguments to operators[4] and functions are preserved. Following the standard, evaluation of all function arguments and side effects caused by these is sequenced before the actual function call, while evaluation of arguments and side effects outside of the function call are indeterminately sequenced to it.[5]

Listing 1.1. A C program

```
int d = 0; int x = 2;
int reset(int p) {
  return x = d;
}
int main() {
  int y = -1;
  while (x > reset(1))
    reset(d = y);
  return x;
}
```

Example 1. We consider an execution of the program in Listing 1.1. Execution of a C program always begins in the function main. First, a local variable y is initialized with the value -1. Then the condition of the **while** loop (x > reset(1)) is evaluated. The C standard imposes no order on the evaluation of the arguments x and reset(1) of the operator >. Therefore, either of the following executions follow the standard:

1. reset(1) is called, setting x to 0 while returning the value 0, then x is evaluated to 0. Finally, $0 > 0$ is evaluated to 0,[6] thus the condition is deemed false, the **while** loop is exited and the program returns 0 (the value of x).

[3] Our model extraction tool C2ABS can process a much larger subset of C. The given subset, however, is sufficient to demonstrate the key focus of this paper: making non-deterministic unspecified behavior of a C program explicit through active objects.

[4] The subset of C under consideration does not contain operators which introduce sequence points, such as the comma operator $(\cdot,\cdot)$ or the ternary operator $(\cdot ? \cdot : \cdot)$.

[5] This means that the evaluation of arguments and side effects outside of the function call may happen before or after—but not during—the execution of the function call.

[6] In C relational operators return 1 for true, 0 for false; **if** and **while** treat the condition 0 as false, everything else as true.

Fig. 2. Syntax for a subset of C

$$
\begin{aligned}
Decl_c &::= GlobalVarDecl_c \text{ ';' } \mid FuncDecl_c \text{ ';' } \mid FuncDef_c \\
GlobalVarDecl_c &::= \textbf{`int'}\ GlobalId_c \text{ '=' } \mathbb{Z} \\
FuncDecl_c &::= \textbf{`int'}\ FuncId_c \text{ '(' } ParamDecls_c \text{ ')'} \\
ParamDecls_c &::= \epsilon \mid \textbf{`int'}\ LocalId_c \{ \text{ ',' } \textbf{`int'}\ LocalId_c \} \\
FuncDef_c &::= FuncDecl_c \text{ '\{' } \{ Stmt_c \} \textbf{`return'}\ Expr_c \text{ ';' '\}'} \\
Stmt_c &::= \text{';'} \mid \text{'\{'} \{ Stmt_c \} \text{'\}'} \mid LocalVarDecl_c \mid If_c \mid While_c \mid Expr_c \text{ ';'} \\
LocalVarDecl_c &::= \textbf{`int'}\ LocalId_c \text{ '=' } Expr_c \text{ ';'} \\
If_c &::= \textbf{`if'}\ \text{'('}\ Expr_c \text{ ')'}\ Stmt_c\ [\ \textbf{`else'}\ Stmt_c\ ] \\
While_c &::= \textbf{`while'}\ \text{'('}\ Expr_c \text{ ')'}\ Stmt_c \\
Expr_c &::= \text{'('}\ Expr_c \text{ ')'} \mid \mathbb{Z} \mid GlobalId_c \mid LocalId_c \mid Expr_c\ Operator\ Expr_c \mid \\
&\quad\ GlobalId_c \text{ '=' } Expr_c \mid LocalId_c \text{ '=' } Expr_c \mid FuncId_c \text{ '(' } Args_c \text{ ')'} \\
Args_c &::= \epsilon \mid Expr_c \{ \text{ ',' } Expr_c \}
\end{aligned}
$$

2. x is evaluated to 2, reset (1) is called, setting x to 0 while returning the value 0. Finally, $2 > 0$ is evaluated to 1, thus the condition is deemed true and the **while** loop entered. The expression statement reset (d = y); is executed by evaluating the expression. It is ensured that the value and side effect of d = y are evaluated before the function reset is called. Therefore d is set to -1 and reset (-1) is called, setting x to -1 (the value of d). Now the condition of the **while** loop is checked again and will evaluate to 0 regardless of evaluation order, thus exiting the loop. The program returns -1 (the value of x).

Execution of the program is thus underspecified, due to *implicit* non-determinism.[7]

2.2 Output Language: Active Objects

Languages such as Java or C feature low-level concurrency where a thread can be preempted at any time by another process running on the same processor and heap space. This leads to myriads of possible interleavings that cause complex data races being hard to contain and to characterize. On the opposite side of preemptive scheduling is actor-based, distributed programming [16], where all methods are executed atomically and concurrency occurs only among distinct processors with disjoint heaps. In this scenario it is possible to specify behavior completely at the level of interfaces, typically in the form of behavioral invariants jointly maintained by an object's methods. The drawback is: this restrictive form of concurrency forces one to model and to specify systems at a highly abstract level, essentially in the form of protocols. It precludes modeling of concurrent

[7] Potential results of unspecified behavior in C often go unnoticed by the programmer.

behavior that is closer to real programs, such as waiting for results computed asynchronously on the same processor and heap.

Recently, *active object* languages [4] attempt to occupy a middle ground between preemption and full distribution. We focus on ABS [9] which is based on *cooperative scheduling* and has been used to model complex, industrial concurrent systems [2]. Cooperative scheduling implies that tasks cannot be preempted, but they may explicitly and voluntarily *suspend* their execution to allow a required result to be provided by another task: concurrent methods on the same processor and heap *cooperate* with each other to achieve a common goal.

The ABS language construct realizing this behavior has the form `await f?`, where `f` is a reference (called *future*) to the result of a method that may not have completed. Its effect is that the current task suspends itself and only resumes once the value of `f` is available. However, there might be more tasks except the one computing `f`'s value waiting for execution at this point. It is not determined in which sequence these waiting tasks are scheduled. Since they share the same memory, data races among them are possible.

Crucially, since the only ABS statement that can suspend execution is `await`, data races are *localized* in that they can *only* occur at await statements (or at the start of a method). Likewise, since all ABS methods run uninterruptedly either to completion or until they encounter an await statement, only the *final state* reached at the end of a method or before an await statement needs to be known when analyzing local data races. Hence, it suffices to reason about a very specific form of data race at few, explicitly specified code locations.

Given a program from our C subset we extract an ABS$_{lite}$ model from it. Fig. 3 shows the syntax of ABS$_{lite}$.[8] For a brief overview of the semantics of ABS$_{lite}$, consider the model in Listing 1.2. The main block at the end is executed when the model is run. A new object o of class C is created with an initial value of 5 for the implicitly defined field **this**.x. Then two asynchronous calls are made to the object o: one call to add 2 to the field x and one call to return the value of field x.An asynchronous call immediately returns a future value, which can be polled through an **await** statement to see if the method call has returned. The **await** statement ensures that no further code in the main block is executed until both asynchronous calls have returned. In the meantime the active object o has received the two asynchronous calls. It begins to execute one of these calls. Once that call has returned, it will execute the other. Depending on the order it executes these calls, the value returned by getX() is either 5 or 7. The **get** returns the value of a future, blocking if neccessary until the value is available. Here the **await** ensures that the return value from the call

Listing 1.2. A model in ABS$_{lite}$

```
class C(Int x) {
  Unit add(Int y) {
    this.x = this.x + y;
    return Unit;
  }
  Int getX() {
    return this.x;
  }
}
{ // main block
  C o = new C(5);
  Fut<Unit> se = o!add(2);
  Fut<Int> fx = o!getX();
  await se? & fx?;
  Int z = fx.get;
}
```

[8] C2ABS produces a model in ABS with additional features. ABS$_{lite}$ described here is chosen to show only what is actually required to extract a model from the C subset.

Fig. 3. Syntax for ABS_{lite}

$$
\begin{aligned}
Model_a &::= \{\ ClassDecl_a\ \}\ Block_a \\
ClassDecl_a &::= \text{`}\mathbf{class}\text{'}\ ClassId_a\ \text{`(`}\ ParamDecls_a\ \text{`)'}\ \text{`\{'}\ \{\ Decl_a\ \}\ \text{`\}'} \\
Decl_a &::= FieldDecl_a \mid MethodDecl_a \\
FieldDecl_a &::= Type_a\ FieldId_a\ \text{`=`}\ PureExpr_a\ \text{`;'} \\
Type_a &::= \text{`}\mathbf{Int}\text{'} \mid \text{`}\mathbf{Bool}\text{'} \mid \text{`}\mathbf{Unit}\text{'} \mid \text{`}\mathbf{Fut}\text{'}\ \text{`<'}\ Type_a\ \text{`>'} \mid ClassId_a \\
MethodDecl_a &::= Type_a\ MethodId_a\ \text{`(`}\ ParamDecls_a\ \text{`)'}\ RetBlock_a \\
ParamDecls_a &::= \epsilon \mid Type_a\ ParamId_a\ \{\ \text{`,'}\ Type_a\ VarId_a\ \} \\
RetBlock_a &::= \text{`\{'}\ \{\ Stmt_a\ \}\ \text{`}\mathbf{return}\text{'}\ PureExpr_a\ \text{`;'}\ \text{`\}'} \\
Stmt_a &::= \text{`;'} \mid Block_a \mid VarDecl_a \mid Assign_a \mid If_a \mid While_a \mid Await_a \mid Expr_a\ \text{`;'} \\
Block_a &::= \text{`\{'}\ \{\ Stmt_a\ \}\ \text{`\}'} \\
VarDecl_a &::= Type_a\ VarId_a\ \text{`=`}\ Expr_a\ \text{`;'} \\
Assign_a &::= (VarId_a \mid \text{`}\mathbf{this}\text{'}\ \text{`.'}\ FieldId_a)\ \text{`=`}\ Expr_a\ \text{`;'} \\
If_a &::= \text{`}\mathbf{if}\text{'}\ \text{`(`}\ PureExpr_a\ \text{`)'}\ Stmt_a\ [\ \text{`}\mathbf{else}\text{'}\ Stmt_a\ ] \\
While_a &::= \text{`}\mathbf{while}\text{'}\ \text{`(`}\ PureExpr_a\ \text{`)'}\ Stmt_a \\
Await_a &::= \text{`}\mathbf{await}\text{'}\ PureExpr_a\ \text{`?'}\ \{\ \text{`\&'}\ PureExpr_a\ \text{`?'}\ \}\ \text{`;'} \\
Expr_a &::= \text{`}\mathbf{new}\text{'}\ ClassId_a\ \text{`(`}\ Args_a\ \text{`)'} \mid AsyncCall_a \mid GetExpr_a \mid PureExpr_a \\
AsyncCall_a &::= (\text{`}\mathbf{this}\text{'} \mid VarId_a \mid \text{`}\mathbf{this}\text{'}\ \text{`.'}\ FieldId_a)\ \text{`!'}\ MethodId_a\ \text{`(`}\ Args_a\ \text{`)'} \\
Args_a &::= \epsilon \mid PureExpr_a\ \{\ \text{`,'}\ PureExpr_a\ \} \\
GetExpr_a &::= (VarId_a \mid \text{`}\mathbf{this}\text{'}\ \text{`.'}\ FieldId_a)\ \text{`.'}\ \text{`}\mathbf{get}\text{'} \\
PureExpr_a &::= \text{`(`}\ PureExpr_a\ \text{`)'} \mid VarId_a \mid \text{`}\mathbf{this}\text{'}\ \text{`.'}\ FieldId_a \mid OpExpr_a \mid Literal_a \\
OpExpr_a &::= \text{`!'}\ PureExpr_a \mid PureExpr_a\ Operator\ PureExpr_a \\
Literal_a &::= \mathbb{Z} \mid \text{`}\mathbf{True}\text{'} \mid \text{`}\mathbf{False}\text{'} \mid \text{`}\mathbf{Unit}\text{'}
\end{aligned}
$$

to getX() is available. It is stored in the local variable z. Through the *explicit* non-determinism[9] of active objects (realized by the two asynchronous calls) the value of z is underspecified.

3 Model Extraction

An overview of the model extraction process is in Fig. 4. Each function definition is modelled as a class, while each executing function call is modelled as an active object of that class. Evaluation of (sub)expressions and side effects take place in asynchronous method calls to the same active object, while **await** statements at which forked asynchronous calls are joined model the sequencing rules of the

[9] ABS code is atomically and deterministically executed by default. Non-determinism occurs only at scheduling points that are syntactically explicit in the code.

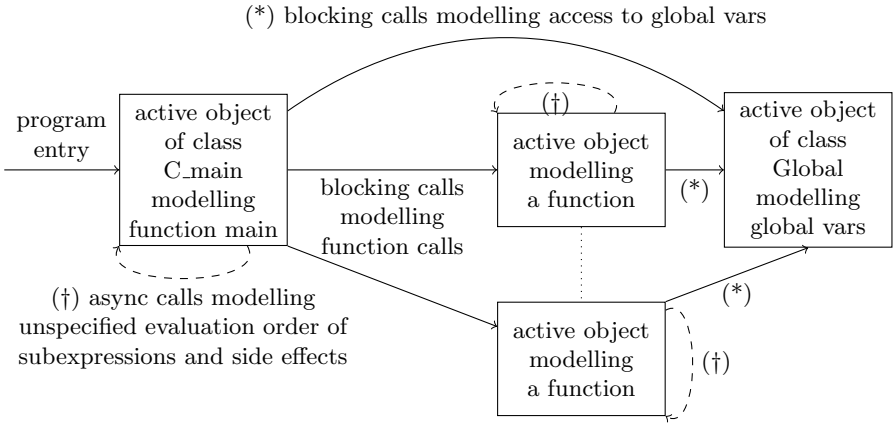

Fig. 4. Overview of model extraction

C standard. If a function is called multiple times (whether recursively or iteratively), each of these calls is modelled by its own active object. As all functions have access to the global variables[10], a single active object which all other active objects have access to is used to model the state of all global variables. Blocking calls to the global object are used to access/modify the global variables. Additionally, blocking calls are used to pass control from one function call to a nested function call being executed, as the C standard ensures that subexpressions and side effects outside of a function call are indeterminately sequenced to it and, therefore, cannot occur during execution of the function call.

3.1 Modelling global variable declarations and initial call to main

Given a program p we construct the model shown in Listing 1.3. The function *extractFunctions* is described in Sect. 3.3 and *extractGlobalVars* is defined to create a class Global, which contains all global variables with their initial values as fields, with getter and setter methods for these fields:

$$
\begin{aligned}
&\mathit{extractGlobalVars} : \mathit{Decl_c}^* \to \mathit{Decl_a}^* \\
&\qquad\qquad \epsilon \mapsto \epsilon \\
&\qquad\qquad \mathit{decl}\ \mathit{decls} \mapsto \mathit{extractGlobalVars}(\mathit{decls}), \text{ if } \mathit{decl} \notin \mathit{GlobalVarDecl_c} \\
&\quad \textbf{int}\ \mathit{gv} = z;\ \mathit{decls} \mapsto \textbf{Int}\ \mathit{gv} = z;\ \textbf{Int}\ \mathrm{get\_}\mathit{gv}()\ \{\ \textbf{return}\ \mathit{gv};\ \} \\
&\qquad\qquad\qquad\qquad \textbf{Unit}\ \mathrm{set\_}\mathit{gv}(\textbf{Int}\ \mathrm{x})\,\{\,\textbf{this}.\mathit{gv} = \mathrm{x};\ \textbf{return Unit};\}\\
&\qquad\qquad\qquad\qquad \mathit{extractGlobalVars}(\mathit{decls})
\end{aligned}
$$

In the main block, we create an active object of class Global and pass this to an active object modelling the program entry. Whenever new active objects

[10] We ignore the potential for variable shadowing.

modelling function calls are created, we pass the Global object along, such that every modelled function call has access to the global variables. As an example, Listing 1.4 shows the extracted Global class from Listing 1.1.

Listing 1.3. Extracted model of p

```
class Global {
  extractGlobalVars(p)
}
extractFunctions(p)
{
  Global global = new Global();
  C_main o = new C_main(global);
  Fut<Int> fv = o!call();
  Int v = fv.get;
}
```

Listing 1.4. Example Global class

```
class Global {
  Int d = 0;
  Int get_d() { return d; }
  Unit set_d(Int x) {
    this.d = x;
    return Unit;
  }
  Int x = 2;
  Int get_x() { return x; }
  Unit set_x(Int x) {
    this.x = x;
    return Unit;
  }
}
```

3.2 Modelling unspecified evaluation order within expressions

Evaluating an expression in C can exhibit unspecified behaviour due to the lack of a rigid evaluation order for subexpressions and side effects offered by the typical C standards (as opposed to, e.g., the Java language specification). To correctly model this unspecified behavior, we take advantage of the explicit non-determinism of active objects with respect to the execution order of asynchronous calls. Execution of a function call in C is modelled by an active object executing its call method. Within this method multiple asynchronous calls can be made to other methods of this active object followed by an **await** statement, such that these other methods can be executed in a non-deterministic fashion.

$$
\begin{aligned}
(\ e_1\ ) &\mapsto \mathit{convert}(e_1)\\
z &\mapsto (\textbf{Fut<Int>}\ \mathrm{x} = \textbf{this}!\mathrm{id}(z);,\ \epsilon,\ \mathrm{x})\\
lv &\mapsto (\textbf{Fut<Int>}\ \mathrm{x} = \textbf{this}!\mathrm{get\_}lv();,\ \epsilon,\ \mathrm{x})\\
gv &\mapsto (\textbf{Fut<Int>}\ \mathrm{x} = \textbf{this}!\mathrm{getGlobal\_}gv();,\ \epsilon,\ \mathrm{x})\\
lv = e_1 &\mapsto (\mathit{stmts}_1\ \textbf{Fut<Unit>}\ \mathrm{se} = \textbf{this}!\mathrm{set\_}lv(x_1);,\ \mathit{se}_1\ \mathrm{se},\ x_1)\\
gv = e_1 &\mapsto (\mathit{stmts}_1\ \textbf{Fut<Unit>}\ \mathrm{se} = \textbf{this}!\mathrm{setGlobal\_}gv(x_1);,\ \mathit{se}_1\ \mathrm{se},\ x_1)\\
e_1 \oplus e_2 &\mapsto (\mathit{stmts}_1\ \mathit{stmts}_2\ \textbf{Fut<Int>}\ \mathrm{x} = \textbf{this}!op_{\oplus}(x_1, x_2);,\ \mathit{se}_1\ \mathit{se}_2,\ \mathrm{x})\\
f(e_1, \ldots, e_n) &\mapsto (\mathit{stmts}_1 \cdots \mathit{stmts}_n\ \textbf{Fut<Int>}\ \mathrm{x} = \textbf{this}!\mathit{call\_f\_m}(\mathit{args});,\ \epsilon,\ \mathrm{x})\\
&\quad \text{where } \mathit{args} = x_1, \ldots, x_n, \mathit{se}_{1_1}, \ldots, \mathit{se}_{n_{|se_n|}} \text{ and } m = \textstyle\sum_{i=1}^{n} |\mathit{se}_i|
\end{aligned}
$$

Fig. 5. The function $\mathit{convert} : \mathit{Expr}_c \to \mathbb{EW}$

Definition 1. *A tuple* $(\mathit{stmts}, \mathit{se}, \mathit{futVar}) \in (\mathit{VarDecl}_a{}^* \times \mathit{VarId}_a{}^* \times \mathit{VarId}_a)$*, where se contains only local variables of type* **Fut<Unit>** *declared in stmts*

(the side-effects of the evaluated expression) and futVar is a local variable of type **Fut<Int>** *declared in stmts (the value of the evaluated expression) is defined as an* expression wrapper[11]*. The set of all expression wrappers is defined as* $\mathbb{EW}$*.*

We define the function *convert* in Fig. 5, which converts a C expression into an expression wrapper recursively, where $\mathrm{x}, \mathrm{se} \in \mathit{VarId}_a$ are fresh unused identifiers, $e_i \in \mathit{Expr}_c, z \in \mathbb{Z}, lv \in \mathit{LocalId}_c, gv \in \mathit{GlobalId}_c, (\mathit{stmts}_i, \mathit{se}_i, x_i) = \mathit{convert}(e_i), \oplus \in \mathit{Operator}$ and $f \in \mathit{FuncId}_c$.

```
Int id(Int x) { return x; }

Int get_lv() { return this.lv; }

Unit set_lv(Fut<Int> fx)
{
  await fx?;
  this.lv = fx.get;
  return Unit;
}

// for ⊕ ∈ {+,-,*}:
Int op⊕(Fut<Int> fx, Fut<Int> fy)
{
  await fx? & fy?;
  Int x = fx.get;
  Int y = fy.get;
  return x ⊕ y;
}
```

```
Int getGlobal_gv() {
  Fut<Int> fx = this.global!get_gv();
  // no await, blocking call
  Int result = fx.get;
  return result;
}

Unit setGlobal_gv(Fut<Int> fx) {
  await fx?; Int x = fx.get;
  Fut<Unit> se = this.global!set_gv(x);
  se.get; // no await, blocking call
  return Unit;
}

// for R ∈ { ==,!=,>,>=,<,<= }:
Int opR(Fut<Int> fx, Fut<Int> fy) {
  await fx? & fy?;
  Int x = fx.get; Int y = fy.get;
  Int result = 0;
  if (x R y) result = 1;
  return result;
}
```

```
Int call_f_m(Fut<Int> fx1,..., Fut<Int> fxn,
             Fut<Unit> se1,..., Fut<Unit> sem) {
  await fx1? & ... & fxn? & se1? & ... & sem?;
  Int x1 = fx1.get; ... Int xn = fxn.get;
  C_f o = new C_f(this.global, x1,..., xn);
  Fut<Int> fr = o!call();
  // no await, blocking call
  Int result = fr.get;
  return result;
}
```

Fig. 6. Families of required helper methods

As can be seen in the function *convert*, asynchronous calls to various methods of the current active object are made. The active object classes generated from a C function are thus required to implement the subset of methods in Fig. 6 which are used in the converted expression wrappers of all expressions contained in the function definition.

[11] In this paper we restrict expression wrappers to $(\mathit{VarDecl}_a{}^* \times \mathit{VarId}_a{}^* \times \mathit{VarId}_a)$, while in C2ABS they are in the superset $(\mathit{Stmt}_a{}^* \times \mathit{VarId}_a{}^* \times \mathit{PureExpr}_a)$.

Side effects are created only by assignments, while the side effects of an operator's operands are gathered and passed upwards. A function call has no side effects in this sense[12], but rather introduces a sequence point between evaluation of function arguments and any side effects produced therein, and the function call itself. For this reason the call to *call_f_m* contains the future values for all side effects of the function arguments, in addition to the arguments themselves. This allows an **await** statement to ensure that all side effects are completed, before the actual call to the function is modelled by creating a new active object of the appropriate type and calling its call method.

3.3 Modelling function definitions as classes

The function *extractFunctions* called in Listing 1.3 extracts ABS_{lite} classes modelling C function definitions and is defined in Fig. 7, together with *extractFunction* and *extractLocalVars*. Here $(stmts', se'_1 \cdots se'_n, x') = convert(e)$ and *extractStmts* (and helper functions *extract* and *varDeclToAssign*) are defined in Fig. 8.

Function parameters are modelled as class parameters (which are implicit fields), while local variables are modelled as explicit fields of the class. This allows access to them as required from the helper methods. For this reason a local variable declaration needs to be treated twice: once by creating a field to model this local variable and assigning it a witness term (**Int** $lv = 0$;) in *extractLocalVars* and once by modelling the initial value for the local variable by assignment (**this**.$lv = x'$.**get**;) in *extract*.

Treating **while** loops introduces an additional wrinkle: while in C the condition of a while loop can contain side effects, in ABS this is not possible. For this reason the auxiliary statements in the expression wrapper required to calculate the value of the pure expression must be performed twice: once before the **while** loop and once at the end of the loop body before re-evaluating the condition. We re-use the local variables declared in the auxiliary statements by replacing local variable declarations with assignment in *varDeclToAssign*.

4 Experiments

We developed an Eclipse plugin C2ABS which extracts an ABS model from a given C program, following the translation approach described in the previous sections.[13] To validate an extracted model we analyze it with SYCO[14], a systematic tester for ABS concurrent objects. The SYCO kernel includes state-of-the-art partial-order reduction techniques to avoid redundant computations during testing [3]. Two runs of an ABS program with the same main method are

[12] Obviously, a function call can have side effects, by changing the values of global variables, but these will be dealt with in the active object modelling the function call, rather than in the current active object.

[13] C2ABS with example inputs and outputs can be found at: `https://www.informatik.tu-darmstadt.de/se/se_research/se_projects/fsen_2019.en.jsp`

[14] `http://costa.fdi.ucm.es/syco/clients/web/`

$$
\begin{aligned}
&\mathit{extractFunctions} : \mathit{Decl_c}^* \to \mathit{ClassDecl_a}^* \\
&\epsilon \mapsto \epsilon \\
&\mathit{decl}\ \mathit{decls} \mapsto \begin{cases} \mathit{extractFunction}(\mathit{decl})\ \mathit{extractFunctions}(\mathit{decls}) & \text{, if } \mathit{decl} \in \mathit{FuncDef_c} \\ \mathit{extractFunctions}(\mathit{decls}) & \text{, otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{extractFunction} : \mathit{FuncDef_c} \to \mathit{ClassDecl_a} \\
&\texttt{int}\ f(\texttt{int}\ p_1, \ldots, \texttt{int}\ p_n)\{\mathit{stmts}\ \texttt{return}\ e;\} \\
&\qquad\qquad \mapsto \\
&\texttt{class}\ \mathrm{C\_}f(\mathrm{Global\ global}, \texttt{Int}\ p_1, \ldots, \texttt{Int}\ p_n)\ \{ \\
&\quad \mathit{extractLocalVars}(\mathit{stmts}) \\
&\quad \texttt{Int}\ \mathrm{call}()\ \{ \\
&\quad\quad \mathit{extractStmts}(\mathit{stmts}) \\
&\quad\quad \mathit{stmts}' \\
&\quad\quad \texttt{await}\ x'\ \&\ se'_1\ \&\ \ldots\ \&\ se'_n; \\
&\quad\quad \texttt{Int}\ \mathrm{result} = x'.\texttt{get}; \\
&\quad\quad \texttt{return}\ \mathrm{result}; \\
&\quad \} \\
&\quad \ldots\ //\ \mathrm{required\ helper\ methods}\ (\text{see Fig. 6}) \\
&\}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{extractLocalVars} : \mathit{Stmt_c}^* &\to \mathit{FieldDecl_a}^* \\
\epsilon &\mapsto \epsilon \\
;\ \mathit{stmts} &\mapsto \mathit{extractLocalVars}(\mathit{stmts}) \\
\{\ \mathit{stmts}_1\ \}\ \mathit{stmts}_2 &\mapsto \mathit{extractLocalVars}(\mathit{stmts}_1\ \mathit{stmts}_2) \\
e;\ \mathit{stmts} &\mapsto \mathit{extractLocalVars}(\mathit{stmts}) \\
\texttt{int}\ \mathit{lv} = e;\ \mathit{stmts} &\mapsto \texttt{Int}\ \mathit{lv} = 0;\ \mathit{extractLocalVars}(\mathit{stmts}) \\
\texttt{if}\ (e)\ \mathit{st}\ \mathit{stmts} &\mapsto \mathit{extractLocalVars}(\mathit{st}\ \mathit{stmts}) \\
\texttt{if}\ (e)\ \mathit{st}_1\ \texttt{else}\ \mathit{st}_2\ \mathit{stmts} &\mapsto \mathit{extractLocalVars}(\mathit{st}_1\ \mathit{st}_2\ \mathit{stmts}) \\
\texttt{while}\ (e)\ \mathit{st}\ \mathit{stmts} &\mapsto \mathit{extractLocalVars}(\mathit{st}\ \mathit{stmts})
\end{aligned}
$$

Fig. 7. The functions *extractFunctions*, *extractFunction* and *extractLocalVars*

redundant relative to each other when any possible difference in the scheduling of tasks cannot possibly lead to a data race. Obviously, this is an undecidable property. SYCO safely under-approximates redundant computations.

Table 1 contains C programs that contain expressions with unspecified evaluation order. The programs two-unspec, Schrödinger and one-to-fib are based on an idea by Derek Jones[15], where the C standard allows two-unspec to re-

[15] `http://shape-of-code.coding-guidelines.com/2011/06/18/fibonacci-and-jit-compilers/`

$$
\begin{aligned}
&\mathit{extractStmts} : \mathit{Stmt_c}^* \to \mathit{Stmt_a}^* \\
&\qquad \epsilon \mapsto \epsilon \\
&\qquad \mathit{st}\ \mathit{stmts} \mapsto \mathit{extract}(\mathit{st})\ \mathit{extractStmts}(\mathit{stmts})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{extract} : \mathit{Stmt_c} \to \mathit{Stmt_a}^* \\
&\qquad ; \mapsto \epsilon \\
&\qquad \{\ \mathit{stmts}\ \} \mapsto \mathit{extractStmts}(\mathit{stmts}) \\
&\qquad e; \mapsto \mathit{stmts}'\ \mathbf{await}\ x'\ \&\ se'_1\ \&\ \ldots\ \&\ se'_n; \\
&\qquad \mathbf{int}\ \mathit{lv} = e; \mapsto \mathit{stmts}'\ \mathbf{await}\ x'\ \&\ se'_1\ \&\ \ldots\ \&\ se'_n;\ \mathbf{this}.\mathit{lv} = x'.\mathbf{get}; \\
&\qquad \mathbf{if}\ (e)\ \mathit{st} \mapsto \mathit{stmts}'\ \mathbf{await}\ x'\ \&\ se'_1\ \&\ \ldots\ \&\ se'_n;\ \mathbf{Int}\ \mathrm{x} = x'.\mathbf{get}; \\
&\qquad\qquad \mathbf{if}\ (\mathrm{x}\ \mathrm{!}= 0)\ \{\ \mathit{extract}(\mathit{st})\ \} \\
&\qquad \mathbf{if}\ (e)\ \mathit{st}_1\ \mathbf{else}\ \mathit{st}_2 \mapsto \mathit{stmts}'\ \mathbf{await}\ x'\ \&\ se'_1\ \&\ \ldots\ \&\ se'_n;\ \mathbf{Int}\ \mathrm{x} = x'.\mathbf{get}; \\
&\qquad\qquad \mathbf{if}\ (\mathrm{x}\ \mathrm{!}= 0)\ \{\ \mathit{extract}(\mathit{st}_1)\ \}\ \mathbf{else}\ \{\ \mathit{extract}(\mathit{st}_2)\ \} \\
&\qquad \mathbf{while}\ (e)\ \mathit{st} \mapsto \mathit{stmts}'\ \mathbf{await}\ x'\ \&\ se'_1\ \&\ \ldots\ \&\ se'_n;\ \mathbf{Int}\ \mathrm{x} = x'.\mathbf{get}; \\
&\qquad\qquad \mathbf{while}\ (\mathrm{x}\ \mathrm{!}= 0)\ \{ \\
&\qquad\qquad\quad \mathit{extract}(\mathit{st}) \\
&\qquad\qquad\quad \mathit{varDeclToAssign}(\mathit{stmts}') \\
&\qquad\qquad\quad \mathbf{await}\ x'\ \&\ se'_1\ \&\ \ldots\ \&\ se'_n;\ \mathrm{x} = x'.\mathbf{get}; \\
&\qquad\qquad \}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{varDeclToAssign} : \mathit{VarDecl_a}^* \to \mathit{Assign_a}^* \\
&\qquad \epsilon \mapsto \epsilon \\
&\qquad T\ x = e;\ \mathit{stmts} \mapsto x = e;\ \mathit{varDeclToAssign}(\mathit{stmts})
\end{aligned}
$$

Fig. 8. The functions *extractStmts*, *extract* and *varDeclToAssign*

turn either 1 or 2, Schrödinger tests if two calls to two-unspec are equal and one-to-fib(n) returns a value between 1 and the n-th Fibonacci number. Too many false positives are often a problem with static code checkers, so no-reliance is a test case which does not rely on unspecified evaluation order, calculating the same result despite *different* execution paths. Finally, assign-chain returns (x = y = z = 5) + f(), where f returns the sum x + y + z, to test unspecified evaluation order of side effects.

We compared the result of model extraction with C2ABS followed by analysis with SYCO to program analysis using Cerberus[16], a tool for developing a semantic model for a substantial fragment of C [11]. It takes a similar approach than we do by cross-compiling C into a Lisp dialect and performing analysis on that program. Table 1 contains the number of explored states during anal-

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

Program	Extraction w/C2ABS, validation w/SYCO				Cerberus (45s timeout)	
	Explored states	Time (ms)	Results	Executions	Results	Executions
two-unspec	42	19	1, 2	2	1, 2	7
Schrödinger	148	190	0, 1	4	0, 1	98
one-to-fib(3)	58	35	1, 2	2	1, 2	7
one-to-fib(4)	382	972	1, 2, 3	12	timeout	
no-reliance	104	120	0	2	0	2
Listing 1.1	208	570	-1, 0	5	-1, 0	9
assign-chain	4609	12838	11, 13, 14, 15, 16, 17, 18, 20	480	11, 13, 16, 20	42

Table 1. Model validation with SYCO compared to program analysis with Cerberus

ysis and the total time spent for the SYCO web interface. The Cerberus web interface has a 45 second timeout and does not give exact run times. We also show the different possible results for the programs and the number of execution paths deemed different by the tools. In the case of SYCO, it shows only those executions that lead to a different configuration after partial order reduction [1].

While Cerberus times out after 45 seconds for one-to-fib(4), SYCO manages to completely validate the model extracted by C2ABS in less than a second. SYCO recognizes that there are only 4 different paths in the Schrödinger model, while Cerberus claims 98. But most interesting are the different results for assignment-chain: here the difference seems to be that Cerberus assumes the order of the side effects is set (first assign z, then y, then x) and only allows the evaluation of f() to interleave. However, this does not match the C standard which clearly states that the evaluation order of side effects is unspecified. Our model faithfully reflects this, allowing the side effects and function call to occur in any order, resulting in additional possible results.

In addition to the C programs where SYCO could fully analyze the extracted model, we considered programs where the extracted model caused SYCO to time out after 45 seconds when attempting to analyze all possible execution paths. The one-to-fib function for inputs greater than 4 is such a case, as well as a nested **for** loop example with 10,000 inner iterations. Partial validation of these larger models was possible, by enabling constraints in SYCO to only consider certain paths, and by using a simulation tool that creates an Erlang program from an ABS model and executes that.[17] With these we can partially validate one-to-fib with inputs up to 19 in less than 10 seconds.

5 Related and Future Work

We discussed the Cerberus tool in the previous section. Apart from it, there is not much published work on model extraction. The SPIN model checker contains the model extractor Modex from C to ProMeLA [8]. Unfortunately, we did not manage to get it to work on our examples. MISRA-C is a well-known subset of the C language widely used in the development of safety-critical systems [13]. One of its rules checks whether the value of an expression is the same under any

[17] `http://samir.fdi.ucm.es:8080/ei/clients/web/`

order of evaluation that the standard permits. It stipulates that no unspecified behavior is caused by the order of evaluation of subexpressions. There are several, mostly commercial, static code analyzers equipped with a MISRA-C compliance checker, for example, Astrée [6], Polyspace[18], Axivion Bauhaus Suite [15], and ECLAIR[19]. All of these are based on abstract interpretation [5]. Also, some compilers like Green Hills, IAR, TASKING and TI are equipped with a MISRA-C compliance checker. In contrast to MISRA-C compliance checkers we want to analyze and detect also non-compliant behavior and we give detailed feedback to the developer about differing computations.

In the future we intend to add operators that introduce sequencing (in particular the ternary operator), as well as tracking sequencing information to recognize undefined behavior, such as changing a value multiple times between sequence points. We will also extend the types C2ABS can deal with. ABS has a formally defined semantics [9], while a semantics for C is given by the K framework[20], allowing a formal proof of the correctness of the translation in future. *Common continuation region analysis* [10] allows recognizing and optimizing asynchronous calls which can be performed in parallel. Finding parallelization potential in the ABS model could then be transferred back to the C program.

6 Conclusion

We described how to extract an ABS model from a C program to make the implicit non-deterministic behavior explicit. There exist a number of tools built to analyze ABS models [17], because the language was designed to be analyzable. This will help us extend the ABS toolbox with tools built to localize parallelizable parts of the model and thus give feedback to the C developers. We implemented our model extraction approach and validated the models thus extracted using SYCO. In doing so, we have found differences in results between our modelling of the C standard and that chosen by developers of the related tool Cerberus. We feel confident that our results are correct. Our approach also seems to scale better. Additionally, we found areas where SYCO can be optimized and relayed this to the developers.

Acknowledgments. We would like to thank the SYCO development team for their support, in particular, Samir Genaim and Miky Zamalloa.

References

1. Albert, E., Arenas, P., Gómez-Zamalloa, M.: Actor- and task-selection strategies for pruning redundant state-exploration in testing. In: Ábrahám, E., Palamidessi, C. (eds.) 34th Conf. on Formal Techniques for Distributed Objects, Components, and Systems. vol. 8461, pp. 49–65. Springer (2014)

[18] See `https://www.mathworks.com/products/polyspace.html`
[19] See `http://www.bugseng.com/eclair-0`
[20] See `https://github.com/kframework/c-semantics`

2. Albert, E., de Boer, F.S., Hähnle, R., Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L., Wong, P.Y.H.: Formal modeling of resource management for cloud architectures: An industrial case study using Real-Time ABS. Journal of Service-Oriented Computing and Applications **8**(4), 323–339 (Dec 2014)
3. Albert, E., Gómez-Zamalloa, M., Isabel, M.: SYCO: a systematic testing tool for concurrent objects. In: Zaks, A., Hermenegildo, M.V. (eds.) Proc. 25th Intl. Conf. on Compiler Construction, CC, Barcelona, Spain. pp. 269–270. ACM (2016)
4. de Boer, F., Din, C.C., Fernandez-Reyes, K., Hähnle, R., Henrio, L., Johnsen, E.B., Khamespanah, E., Rochas, J., Serbanescu, V., Sirjani, M., Yang, A.M.: A survey of active object languages. ACM Computing Surveys **50**(5), 76:1–76:39 (Oct 2017), article 76
5. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Fourth POPL, Los Angeles. pp. 238–252. ACM Press, New York (Jan 1977)
6. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE analyzer. In: European Symposium on Programming. vol. 3444, pp. 21–30. Springer (2005)
7. Hähnle, R.: The Abstract Behavioral Specification language: A tutorial introduction. In: Bonsangue, M., de Boer, F., Giachino, E., Hähnle, R. (eds.) Formal Models for Components and Objects. LNCS, vol. 7866, pp. 1–37. Springer (2013)
8. Holzmann, G.J., Smith, M.H.: An automated verification method for distributed systems software based on model extraction. IEEE Trans. Software Eng. **28**(4), 364–377 (2002)
9. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F., Bonsangue, M.M. (eds.) 9th FMCO. LNCS, vol. 6957, pp. 142–164. Springer (2011)
10. Kim, W., Agha, G.A., Panwar, R.B.: Efficient compilation of concurrent call/return communication in actor-based programming languages. In: Proc. 3rd Intl. Conf. High Performance Computing (HiPC). pp. 62–67 (Dec 1996). https://doi.org/10.1109/HIPC.1996.565798
11. Memarian, K., Matthiesen, J., Lingard, J., Nienhuis, K., Chisnall, D., Watson, R.N.M., Sewell, P.: Into the depths of C: elaborating the de facto standards. In: Krintz, C., Berger, E. (eds.) 37th PLDI. pp. 1–15. ACM (2016)
12. MISRA Consortium: MISRA-C: 2004 — Guidelines for the use of the C language in critical systems (2004)
13. Motor Industry Research Association: MISRA C 2012: Guidelines for the Use of the C Language in Critical Systems (2013)
14. OpenMP Architecture Review Board: OpenMP Application Programming Interface, 4.5 edn. (November 2015), `https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf`
15. Raza, A., Vogel, G., Plödereder, E.: Bauhaus - A tool suite for program analysis and reverse engineering. In: Pinho, L.M., Harbour, M.G. (eds.) 11th Ada-Europe Intl. Conf. on Reliable Software Technologies. LNCS, vol. 4006, pp. 71–82. Springer (2006)
16. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: Modeling and verification of reactive systems using Rebeca. Fundamenta Informatica **63**(4), 385–410 (2004)
17. Wong, P.Y.H., Albert, E., Muschevici, R., Proença, J., Schäfer, J., Schlatte, R.: The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. STTT **14**(5), 567–588 (2012)