A Dynamic Logic for Deductive Verification of C Programs with KeY-C

Authors Oleg Mürk, Daniel Larsson, Reiner Hähnle

Document Id 033

Date July 2007

Availability Restricted CEDES
Status Final

A Dynamic Logic for Deductive Verification of C Programs with KeY-C

Oleg Mürk, Daniel Larsson, and Reiner Hähnle

Chalmers University of Technology, Department of Computer Science and Engineering
S-412 96 Gothenburg, Sweden
oleg.myrk@gmail.com, danla@chalmers.se, reiner@chalmers.se

Abstract. We present KeY-C: a tool for deductive verification of C programs. KeY-C allows verification of C programs w.r.t. operation contracts and invariants. It is based on an earlier version of KeY that supports Java Card. In this paper we outline syntax, semantics, and calculus of C Dynamic Logic (CDL) that were adapted from their Java Card counterparts. Currently, the tool is in an early development stage. As a side-product of this work we expect to generalize KeY architecture for easily adding the support for new programming languages. This paper is a further development of our work described in [11].

1 Introduction

We present KeY-C, a variant of the software verification tool KeY [2], that supports a subset of C as its target language. KeY is an interactive theorem proving environment and allows to prove properties of imperative/object-oriented sequential programs. The central concept is an axiomatization of the operational semantics of the target language in the form of a sequent calculus for dynamic logic, i.e., a program logic. The rules of the calculus that axiomatize program formulas define a symbolic execution engine of programs. The system provides heuristics and proof strategies that automate large parts of proof construction, for example, first-order reasoning, arithmetic simplification, and symbolic execution of loop-free programs are performed mostly automatically. The remaining user input typically consists of occasional existential quantifier instantiations. The main creative part is to define a program contract and supplying loop (in)variants. KeY was designed with ease of interactive proof construction in mind (see screenshot in Figure 1) and to lower the initial learning curve. In particular, the concepts and the syntax of the language specific dynamic logic are as close as possible to the target languages.

The existing KeY system can handle Java Card and most of sequential Java, allowing verification of complex programs [2, Part IV]. Its calculus contains about 1000 rules of which about half are language-independent Dynamic Logic (DL) rules. We are working on adding gradual support for a portable type-safe subset of C, axiomatized in C Dynamic Logic (CDL). In the future we plan to

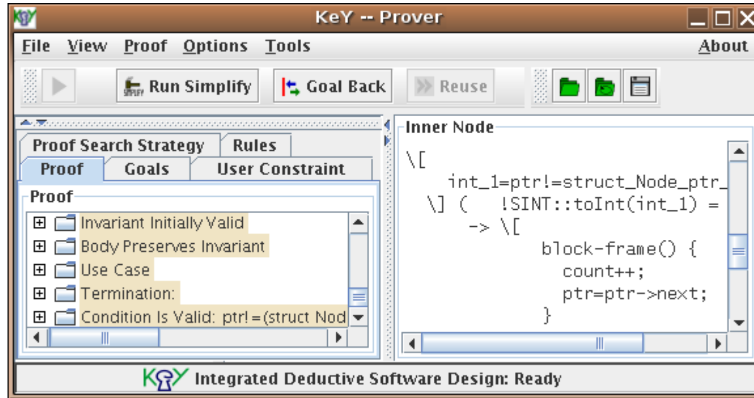

Fig. 1. KeY graphical user interface.

create specializations of this calculus for particular platforms such as MISRA-C [10] or IA-32 (C on Intel 32 bit processors). As a side-product of this work we expect to generalize KeY architecture for easily adding the support for new programming languages.

In Section 2 we outline the principles of the Dynamic Logic (DL), in Section 3 we present an example C program that is used in Sections 4-8 of this paper to illustrate the C Dynamic Logic (CDL) and its calculus. In Section 9 we describe the current status of the KeY-C system and further directions, Section 10 describes related work and we summarize this paper in Section 11.

2 Principles of Dynamic Logic in KeY

Dynamic Logic (DL) is based on First-Order Logic (FOL) extended with a *type system* — function and predicate arguments and function results are equipped with a type. In order to arrive at a reasonable calculus, the subtyping relationship $\subset$ must form a lattice. In the formal semantics all elements of the semantic domain also receive a type. In order to represent different states during program execution, function and predicate symbols are split into *rigid* and *non-rigid* symbols. Rigid symbols behave the same as in FOL, while non-rigid symbols can have different values in different execution states. Basic DL has two types — int with signature and semantics of mathematical integers $\mathbb{Z}$ and boolean that models boolean values true and false.

DL instantiated for a particular programming language is a modal logic with two parametric modalities [P] and $\langle$P$\rangle$ for every syntactically correct and type-checked program P. Its semantics is defined in terms of Kripke structures, where states (worlds) are determined by the values of non-rigid symbols and transitions are defined by the semantics of the programming language. A formula is *valid* iff it is true in all possible states.

The formula $[\mathrm{P}]\phi$ is true in a state s iff ϕ holds in the final state reached when P is started in s provided that P terminates at all. In other words, $[\mathrm{P}]\phi$ asserts *partial correctness* of P w.r.t. postcondition ϕ. The formula $\langle \mathrm{P} \rangle \phi$ is true in a state s iff there exists a final state s' reached when P is started in s, such that ϕ holds in s'. In other words, $\langle \mathrm{P} \rangle \phi$ asserts *total correctness* of P w.r.t. postcondition ϕ. Note that in the case of indeterministic programs there can be more than one final state and ϕ must hold just in one of them. For this reason, in practice, working with non-deterministic programs in DL requires first converting them to deterministic counter-parts at the level of formal semantics, e.g. by feeding in indeterministic choices as an additional input.

A *sequent calculus* is used for performing deduction. A sequent is of the form $\Gamma \vdash \Delta$, where Γ and Δ are sets of formulas. The semantics of a sequent is the same as that of the formula $\bigwedge \Gamma \Rightarrow \bigvee \Delta$. The DL calculus builds upon standard FOL with equality and arithmetic. DL calculus rules that work on program modalities always modify the first active statement of the modalities.

Locations are special non-rigid functions that can have an arbitrary value in different states of the Kripke structure and are used to model modifiable memory locations of the program. I.e. there exists a state for every combination of the values of the locations, while the value of other non-rigid symbols in some state may, for instance, depend on the values of some locations in this state.

The main principle of DL is to reduce program modalities into so-called *updates*. An *atomic update* has the form $\mathcal{U} = \{loc := val\}$, where *loc* is a *location* expression and *val* is its new value term. Semantically, the validity of $\mathcal{U}\phi$ in state s is defined as the validity of ϕ in state s', which is state s where the values of locations are modified according to the update $\mathcal{U}$. There are operations for sequential and parallel composition of updates as well as for quantification, where the update is quantified by a free variable satisfying some condition. Any composition of updates can be automatically transformed to a normal form consisting of a parallel composition of quantified atomic updates (the details are inessential for this paper). An update applied to a pure FOL formula can be automatically transformed into a pure FOL formula without an update. Such FOL formula has to be proven using the standard rules of FOL sequent calculus and the rules expressing the properties of the particular Kripke structure. Most loop-free sequences of program statements can be turned into updates fully automatically.

Typically, the contracts for the programs are represented in the form of proof obligations $pre \Rightarrow [\mathrm{P}]post$ (for partial correctness) and $pre \Rightarrow \langle \mathrm{P} \rangle post$ (for total correctness), where P defines the program being reasoned about (typically a function body), precondition *pre* specifies assumptions about the initial state and postcondition *post* specifies the requirements for the final state.

Of course, there are many more details, some of which will be handled later in this text. A full treatment of the subject can be found in [2, Part IV], although the constructions there are slightly over-specialized for Java.

3 Example C Program

To illustrate the details of C Dynamic Logic (CDL) we use the function list2arr on Figure 2 that converts a linked list pointed to by ifirst into an array and stores the computed length of the linked list into *ocount and the pointer to the first element of the created array into *oarr. The linked list itself is deleted. If the array length is zero or memory allocation failed then *oarr is set to the null pointer. In this paper we mostly work with the program code in the first column, but we also look at some interesting statements in the second column.

```
struct Node {
 struct Node* next;
 int elem;
};

void list2arr(
        struct Node* ifirst,
        int* ocount,
        int** oarr
        )
{
 int count = 0;
 struct Node* ptr = ifirst;
 while (ptr !=
        (struct Node*)0) {
   count = count + 1;
   ptr = (*ptr).next;
 }
 *ocount = count;
```

```
 int* arr = (int*)0;
 if (count > 0) {
   arr = (int*)
     calloc(count, sizeof(int));
   if (arr != (int*)0) {
     struct Node* ptr = ifirst;
     int index = 0;
     while (index < count) {
       arr[index] = ptr->elem;
       struct Node* oldPtr = ptr;
       ptr = ptr->next;
       free(oldPtr);
       index++;
     }
   }
 }
 *oarr = arr;
}
```

Fig. 2. Example C program

4 CDL Syntax and Semantics

The type system and the signature of CDL reflect the peculiarities of the C language. To represent C integer *rvalues* we introduce logic integer types CHAR for **char**, SINT for **signed int**, UINT for **unsigned int**, etc. The reason why we cannot use mathematical integers int for this purpose is because C allows multiple bit-representations of the same mathematical integer value (e.g. so-called *negative zero*). In our example we use only logic type SINT as the C type **signed int** is a synonym for C type **int**.

Further, we need to reason about *objects*. Objects are memory areas that hold values and can be referenced by pointers (and consequently are *lvalues*). We introduce a supertype of all *object* types Void. Symbols representing pointer

rvalues will have a logic type which is either Void (in the case of C type **void**$*$) or one of its subtypes. To represent null pointer rvalues we introduce a subtype of all object types Null. The interpretation of this type consists of exactly one element represented by the constant null. All concrete object types are a subtype of Void and a supertype of Null.

In order to represent C scalar objects (arithmetic or pointer) we introduce a logic *scalar* object type $\mathsf{T@} \subset \mathsf{Void}$ for any type T. This object type is equipped with a location $\mathsf{T@::value} : \mathsf{T@} \to \mathsf{T}$, representing the value of the scalar object.

Further, for every C **struct** S we introduce a logic *struct* object type $\mathsf{\$S}$ with rigid member accessor functions allowing to navigate from the object instance to its members. In the case of **struct** Node from our example these accessor functions are $\mathsf{Node::next} : \mathsf{Node} \to \mathsf{Node@}$ and $\mathsf{Node::elem} : \mathsf{Node} \to \mathsf{int@}$.

Finally, in order to represent arrays we introduce for each logic object type T a logic *array* object type $\mathsf{T[]} \subset \mathsf{Void}$ with two rigid functions. The first function is array element accessor function $\mathsf{T[]::elem} : \mathsf{T[]}, \mathsf{int} \to \mathsf{T}$. The second is rigid array size function $\mathsf{size}(a)$ defined for all arrays.

Please note that both struct and array accessor functions are *rigid*, so their value is the same in all states and so is array size function. The only non-rigid symbols so far that can have different values in different states are the values of scalar objects. In the case when a C scalar variable (e.g. **int** i) cannot be referenced by any pointer we can avoid the extra level of indirection provided by the scalar type $\mathsf{T@}$ (e.g. $\mathsf{SINT@}$) and model such variables just as locations of the value type (e.g. of type SINT). For instance, most local variables in C programs are never referenced by pointers. In our system we mark such variables with C specifier **register**.

In order to model object allocation we introduce for each logic object type T a rigid object repository function $\mathsf{T}::\langle\mathsf{lookup}\rangle : \mathsf{int} \to \mathsf{T}$ and a rigid array repository function $\mathsf{T[]}::\langle\mathsf{lookup}\rangle : \mathsf{int}, \mathsf{int} \to \mathsf{T[]}$. Further we introduce an object counter location $\mathsf{next} : \mathsf{int}$ that is used to allocate objects $\mathsf{T}::\langle\mathsf{lookup}\rangle(\mathsf{next})$ and arrays $\mathsf{SINT@[]}::\langle\mathsf{lookup}\rangle(\mathsf{next}, l)$ of size l and is incremented on each allocation $\{\ \mathsf{next} := \mathsf{next} + 1\ \}$. The repository functions return distinct objects for distinct values of next.

In order to reason about programs we need to express the properties of our heap structure. It turns out that the heap objects form trees rooted in repository functions with parent relationship defined by struct member and array element accessor functions and scalar objects serving as the leaves of the tree. Note that this structure is rigid and does not change in different states. Consequently we need axiomatization of trees containing such rules as

$$\begin{gathered}
\mathsf{\$Node::elem}(o_1) \neq \mathsf{\$Node::next}(o_2), \\
\mathsf{\$Node::elem}(o_1) = \mathsf{\$Node::elem}(o_2) \Leftrightarrow o_1 = o_2, \\
\mathsf{\$Node::elem}(o) \neq \mathsf{SINT@}::\langle\mathsf{lookup}\rangle(i), \\
\mathsf{\$Node::next}(o) \neq \mathsf{\$Node@}::\langle\mathsf{lookup}\rangle(i)\ \ .
\end{gathered}$$

Further we need to define function $\mathsf{rootObj} : \mathsf{Void} \to \mathsf{Void}$ that returns for given object the root of the tree this object belongs to. In order to work with

pointers we need to define predicate isRealObj : Void that expresses that the path from the object to the root does not exceed the bounds of the arrays on this path. Such objects represent valid C memory areas that can be accessed. C language allows pointers to array *tails* — one element past the last element of the array. The predicate isValidObj : Void is supposed to express this property by being true iff the object is real or it is the tail element of a real array.

Finally, we need to express whether the objects are currently allocated or not. This is achieved by associating with each tree of objects a *block* that can be accessed with a rigid function objBlock : Void → Block such that

$$\mathsf{rootObj}(o_1) = \mathsf{rootObj}(o_2) \Leftrightarrow \mathsf{objBlock}(o_1) = \mathsf{objBlock}(o_2) \ .$$

Now we define a location function storage : Block → StorageMode that can have one of the four possible values ST_AUTOMATIC, ST_ALLOCATED, ST_STATIC and ST_TRAP, where the first three denote possible storage modes and the last denotes that the object is not allocated yet or has been already destroyed.

5 CDL Modality Calculus

Now let's consider our example from Figure 2. In this example we assume that no local variables or formal parameters of the function list2arr can be referenced by a pointer, which means that we prefix their declarations with the specifier **register**, as described earlier. We introduce one exception to this rule by not prefixing the declaration of the local variable count with **register** in order to illustrate how things become more laborious when local variables can be referenced by pointers.

The proof obligation for total correctness of function list2arr has the form $pre \Rightarrow \langle \mathrm{P} \rangle post$, where P is the function body and *pre* and *post* are respectively pre- and postconditions expressed in terms of locations ifirst : $Node, ocount : SINT@ and oarr : SINT@@ that model the formal parameters of the function. Parts of the pre- and postconditions will be discussed in Section 8. Here we focus on how to reduce the program part of the proof obligation. According to FOL sequent calculus $\vdash pre \Rightarrow \langle \mathrm{P} \rangle post$ can be reduced into $pre \vdash \langle \mathrm{P} \rangle post$.

The main principle of CDL modality calculus is to process the first active statement of the modality and either reduce into an update or replace it with simpler expressions. Now let's consider the beginning of our function body P

```
{
  int count = 0;
  register struct Node *ptr = ifirst;
  ...
}
```

The variable count has to be allocated in the beginning of the compound statement and destroyed in the end of the compound statement. We achieve this by first introducing a *block frame* and then registering the local variable count to be destroyed at the end of the frame execution

```
block-frame(count) {
  allocate count;
  count = 0;
  ...
}
```

Now statement **allocate** count; is reduced into a sequential update

$$\left\{\begin{array}{l}\mathsf{count} := \mathsf{SINT@}::\langle\mathsf{lookup}\rangle(\mathsf{next});\\ \mathsf{storage}(\mathsf{objBlock}(\mathsf{count})) := \mathsf{ST_AUTOMATIC};\\ \mathsf{next} := \mathsf{next}+1\end{array}\right\} \quad .$$

Further, statement $\mathsf{count} = \mathsf{0}$; is reduced into update

$$\{\ \mathsf{SINT@}::\mathsf{value}(\mathsf{count}) := \mathsf{SINT}::\mathsf{fromInt}(0)\ \}\ ,$$

where $\mathsf{SINT}::\mathsf{fromInt} : \mathsf{int} \to \mathsf{SINT}$ converts mathematical integers into CDL integer representation (if there are many possible representations, the the one chosen by the compiler for the constants is selected). Now, the declaration of ptr is reduced into

```
register struct Node *ptr;
ptr = ifirst;
```

that is reduced into sequential update composition

$$\{\ \mathsf{ptr} := s_0\ \}\{\ \mathsf{ptr} := \mathsf{ifirst}\ \} \quad .$$

Here s_0 is a fresh skolem symbol of type $\mathsf{\$Node}$ that can have any value. According to the C specification local variables are created in *indeterminate* state that can be either an *unspecified valid* value or a *trap* value. Accessing trap values leads to *undefined* behavior. We model pointer trap values by objects that have storage mode $\mathsf{ST_TRAP}$, while integer trap values must be representable by the CDL integer types (like SINT). We model indeterminate and unspecified values with fresh skolem symbols and our calculus is designed in such way that proofs about programs that can perform actions that lead to undefined behavior cannot be closed (we have skipped proving the validity of the values when reducing value assignments above).

The result of symbolically executing the first two statements of the function body has the form $pre \vdash \mathcal{U}\langle \mathrm{P1}\rangle post$, where P1 is the rest of the program contained within the block frame

```
block-frame(count) {
  while (ptr != (struct Node*)0) {
    count = count + 1;
    ptr = (*ptr).next;
  }
  *ocount = count;
  ...
}
```

and update $\mathcal{U}$ is the sequential composition of the updates described above. After all the statements in the block frame are executed and its body becomes empty, the variable count can be destroyed

```
block–frame() {
  destroy count;
}
```

Destruction statement results in an update

$$\{ \mathsf{storage(objBlock(count))} := \mathsf{ST_TRAP} \} \ .$$

An empty block frame with an empty variable list

```
block–frame() {}
```

can be removed from the modality. An empty modality without any code inside can also be removed

$$\mathcal{U}\langle\rangle post \equiv \mathcal{U} post.$$

6 CDL Selection and Iteration Calculus

Let's first consider the rule for the **while** statement in the sequent $pre \vdash \mathcal{U}\langle\mathrm{P1}\rangle post$ that we derived in the previous section. This sequent is reduced into four new sequents.

- *Invariant Initially Valid*:

$$pre \vdash \mathcal{U}(inv \wedge variant \geq 0) \ .$$

- *Body Preserves Invariant*:

$$pre \vdash \mathcal{U}\mathcal{V}(inv \Rightarrow ([\ \mathsf{int\ i0 = C\ != 0}\ ](\mathsf{SINT::toInt(i0)} = 1 \Rightarrow ([\mathsf{B}]inv)))) \ .$$

- *Use Case*:

$$pre \vdash \mathcal{U}\mathcal{V}(inv \Rightarrow ([\ \mathsf{int\ i0 = C\ != 0}\ ](\mathsf{SINT::toInt(i0)} = 0 \Rightarrow ([\mathsf{P2}]post)))) \ .$$

- *Termination*:

$$pre \vdash \mathcal{U}\mathcal{V} \begin{pmatrix} inv \wedge variant \geq 0 \wedge variant = variantSaved_0 \Rightarrow \\ \langle\ \mathsf{int\ i0 = C\ != 0}\ \rangle \\ \begin{pmatrix} \mathsf{SINT::toInt(i0)} = 1 \Rightarrow \\ \langle\mathsf{B}\rangle(variant < variantSaved_0 \wedge variant \geq 0) \end{pmatrix} \end{pmatrix} \ .$$

Here C is the condition of the while statement, B is the while statement body, P2 is P1 with while statement removed, i0 is a temporary variable, *inv* is a loop *invariant*, *variant* is a non-negative loop variant that decreases with each execution of the loop body, and $variantSaved_0$ is a fresh skolem symbol needed to save the value of the variant before executing the loop body. $\mathcal{V}$ is an *anonymous*

update that resets the locations, which can be modified by C and B, with the values of fresh skolem symbols

$$\mathcal{V} = \{\ \mathsf{ptr} = s_1;\ \mathsf{SINT@::value(count)} = s_2\ \}\ .$$

In principle we should reset all locations as the loop body must preserve invariant $\mathcal{I}$ for any initial state satisfying $\mathcal{I}$. However, by resetting only those locations that are modified within the loop body, we get a much more user-friendly rule that is still sound. This rule is a variation on the work described in [3].

Leaving the example program for a moment, **if** statements

$$\langle ..\ \textbf{if}\ \text{(E) B1 } \textbf{else} \text{ B2 ...}\rangle\phi$$

are reduced into

$$\mathsf{if}\ (\mathsf{SINT::toInt(E)} \neq 0)\ \mathsf{then}\ (\langle ..\ \text{B1 ...}\rangle\phi)\ \mathsf{else}\ (\langle ..\ \text{B2 ...}\rangle\phi)$$

when E is a variable with register specifier in its declaration and into

$$\langle ..\ \textbf{register int}\ \text{i0} = \text{E};\ \textbf{if}\ \text{(i0) B1 } \textbf{else} \text{ B2 ...}\rangle\phi$$

otherwise. Notice the pattern $\langle ..\ \ ...\rangle$ to match the surrounding context (currently only the block frames). Here

$$\mathsf{if}\ (C)\ \mathsf{then}\ (B_1)\ \mathsf{else}\ (B_2)$$

is equivalent to

$$(C \Rightarrow B1)\ \wedge\ (\neg C \Rightarrow B2)$$

Such formulae in the succedent of a sequent lead eventually to case distinctions and two branches of the proof, unless one can prove that one of the branches is never taken.

7 CDL Expression Calculus

In this section we discuss how C expressions are reduced into updates by looking at the while statement body B.

Lets consider the first statement count = count + 1;. We introduce a special *value-of* operator @ that denotes the value of a scalar object, so the statement becomes @count = @count + 1;. The main principle of expression calculus is first reducing expression statements into simpler ones

```
register int i1 = @count;
register int i0 = i1 + 1;
@count = i0;
```

Now i1 = @count; is reduced into update

$$\{\ \mathsf{i1} := \mathsf{SINT@::value(count)}\ \}\ ,$$

but before that we have to prove that the value assigned is valid $\mathsf{isValidVal}(\mathsf{SINT@}::\mathsf{value}(\mathsf{count}))$, where

$$\mathsf{isValidVal}(i) \equiv \mathsf{SINT}::\mathsf{MIN} \leq \mathsf{SINT}::\mathsf{toInt}(i) \leq \mathsf{SINT}::\mathsf{MAX} \ .$$

Here SINT::toInt : $\mathsf{SINT} \to \mathsf{int}$ converts CDL integers into mathematical integers. Statement i0 = i1 + 1; is reduced into update

$$\{ \ \mathsf{i0} := \mathsf{SINT}::\mathsf{fromInt}(\mathsf{SINT}::\mathsf{toInt}(\mathsf{i1}) + 1) \ \} \ ,$$

which means we have to prove that the result does not overflow

$$\mathsf{SINT}::\mathsf{MIN} \leq \mathsf{SINT}::\mathsf{toInt}(\mathsf{i1}) + 1 \leq \mathsf{SINT}::\mathsf{MAX} \ .$$

Note that this rule works only if mathematical integers have at most one bit representation, because otherwise $\mathsf{SINT}::\mathsf{toInt}^{-1}(\mathsf{SINT}::\mathsf{toInt}(\mathsf{i1})+1)$ is not uniquely defined.

Creating a calculus for C integer expressions is complicated by the fact that mathematical integers can have different bit representations on different platforms and even during one run of a program. Moreover, in general it is undefined what happens in the case of arithmetic operation or conversion overflows. For now we have chosen CDL integer types to be isomorphic with mathematical integers. We have chosen CDL integer values to be valid iff they fit into the bounds of minimum and maximum values of the corresponding C type and only proofs about programs that do not produce overflows can be closed. This approach works in the case when mathematical integers have at most one bit representation (e.g. two's complement on IA-32) or when we want to abstract away from bit representations and thus programs with bit-operations are not supported. At the same time it is not hard to modify the integer expression calculus for any particular specialization of the C specification.

Now let's consider the second statement of the loop body ptr = @((*ptr).next);. First, this statement is reduced into simpler statements

```
o1 <- *ptr;
o0 <- o1.next;
ptr = @o0;
```

Here o1 is of C type **struct** Node (CDL type $\mathsf{\$Node}$) and o0 is of C type **struct** Node* (CDL type $\mathsf{\$Node@}$). In order to navigate object trees we have introduced an additional object reference assignment operator o <− oexp similar to JAVA reference assignments. Statement o1 <− *ptr; is reduced into update

$$\{ \ \mathsf{o1} := \mathsf{ptr} \ \} \ ,$$

which implies we have to prove that we can dereference the pointer $\mathsf{objExists}(\mathsf{ptr})$, where

$$\mathsf{objExists}(o) \equiv \mathsf{isRealObj}(o) \land \mathsf{storage}(\mathsf{objBlock}(o)) \neq \mathsf{ST_TRAP} \ .$$

Statement o0 <− o1.next; is reduced into update

$$\{ \ \mathsf{o0} := \mathsf{\$Node::next(o1)} \ \} \ ,$$

while statement ptr = @o0; is reduced into update

$$\{ \ \mathsf{ptr} := \mathsf{\$Node@::value(o0)} \ \} \ ,$$

but before that we have to prove that the pointer assigned is valid $\mathsf{isValidPtr}(\mathsf{\$Node@::value(o0)})$, where

$$\mathsf{isValidPtr}(o) \equiv o = \mathsf{null} \lor \mathsf{isValidObj}(o) \land \mathsf{storage}(\mathsf{objBlock}(o)) \neq \mathsf{ST_TRAP} \ .$$

Recall that $\mathsf{isValidPtr}$(o) allows references to array tails.

Creating a calculus for C pointer expressions has many technical challenges. Dynamic arrays can only be referenced by pointers to the first element of the array, so pointer indexing p0 = arr + index; in our example program (arr [index] is reduced into ∗(arr + index)) results in a cumbersome sequential update

$$\left\{ \ \mathsf{p0} := \mathsf{SINT@[]::elem} \begin{pmatrix} \mathsf{(SINT@[])objParent(arr)}, \\ \mathsf{objParentIdx(arr)} + \mathsf{SINT::toInt(index)} \end{pmatrix} \right\} \ .$$

Here $\mathsf{objParent} : \mathsf{Object} \rightarrow \mathsf{Object}$ gives the parent object of the argument (in this case the parent array) and $\mathsf{objParentIdx} : \mathsf{Object} \rightarrow \mathsf{int}$ gives the index of the argument within its siblings (in this case the array elements). When comparing pointers for equality, a pointer to the first child of a struct or an array is equal to the pointer to their parent. Moreover, a pointer to an array tail *may* be equal to a pointer to a root object of any allocated block. In the case when pointer equality is undecidable we use a fresh skolem symbol as the result of the pointer comparison.

Now consider statement p0 = calloc(i0 , **sizeof**(**int**)); in our example program. As we work with a type-safe subset of C then such statement allocates objects of logic array type $\mathsf{SINT@[]}$ and with array size equal to $\mathsf{SINT::toInt(i0)}$, while it's return value is a pointer to the first element of this array cast into type Void. The resulting sequential update is

$$\left\{ \begin{array}{l} \mathsf{o0} := \mathsf{SINT@[]::\langle lookup\rangle(next, SINT::toInt(i0))}; \\ \mathsf{p0} := \mathsf{SINT@[]::elem(o0}, 0); \\ \mathsf{storage(objBlock(o0))} := \mathsf{ST_ALLOCATED}; \\ \mathsf{next} := \mathsf{next} + 1 \end{array} \right\} \ .$$

As calloc can also fail we have also to prove that postcondition holds in the case of update $\{ \ \mathsf{p0} := \mathsf{null}; \ \}$.

Statement free (p1); results in an update

$$\{ \ \mathsf{storage(objBlock(p1))} := \mathsf{ST_TRAP} \ \} \ ,$$

which means we have to prove that

$$\begin{array}{c} \mathsf{objExists(p1)} \ \land \\ \mathsf{storage(objBlock(p1))} = \mathsf{ST_ALLOCATED} \ \land \\ \mathsf{objAliased(rootObj(p1), p1)} \ , \end{array}$$

where $\mathsf{objAliased}(o_1, o_2)$ is a reflexive transitive closure of the relation of o_2 being the first child of o_1.

Finally, C supports *deep* value assignments o1 = o2 of objects when all member values of o2 are copied into o1. Such assignments can be modeled by just rewriting them into a sequence of scalar assignments, but we reduce them into a straightforward update.

The order of evaluating the subexpressions of C expressions is unspecified in many cases. For this reason we require checking by some *external* means that the different evaluation orders give the same result. Surprisingly, it is impossible to do type-checking of expressions in a completely portable way as the types of integer constants, the resulting types of integer promotions and of the usual integer conversions depend on knowing the exact minimum and maximum representable values of the types. For this reason we require all type conversions to be made explicit via type casts, that can be inserted automatically by a preprocessor, if needed.

8 Example Proof

Unfortunately, due to space limitations we cannot discuss the full proof of our example on Figure 2, however we present the constructions needed to prove that the first loop does what is intended. In order to express the precondition that the function argument ifirst refers to a linked list we need to introduce two fresh rigid functions $len : \mathsf{int}$ and $list : \mathsf{int} \to \mathsf{Node}$. Now

$$\begin{aligned} pre &\equiv L(\mathsf{ifirst}) \ \wedge\ \mathsf{objExists}(\mathsf{ocount}), \\ post &\equiv \mathsf{SINT@::value}(\mathsf{ocount}) = len, \\ inv &\equiv 0 \leq \mathsf{SINT::toInt}(\mathsf{SINT@::value}(\mathsf{count})) \leq len\ \wedge \\ &\quad\ \mathsf{ptr} = list(\mathsf{SINT::toInt}(\mathsf{SINT@::value}(\mathsf{count}))), \\ variant &= len - \mathsf{SINT::toInt}(\mathsf{SINT@::value}(\mathsf{count}))\ , \end{aligned}$$

where $L(\mathsf{ifirst})$ expresses that ifirst points to the beginning of the linked list

$$L(\mathsf{ifirst}) \equiv \begin{pmatrix} list(0) = \mathsf{ifirst}\ \wedge\ len \geq 0\ \wedge\ list(len) = \mathsf{null}\ \wedge \\ \forall\ \mathsf{int}\ i; ((0 \leq i)\ \wedge\ (i < len) \Rightarrow Q(i)) \end{pmatrix}$$

and $Q(i)$ expresses the properties of the linked list node i

$$Q(i) \equiv \begin{pmatrix} \mathsf{objExists}(list(i))\ \wedge \\ \$\mathsf{Node@::value}(\$\mathsf{Node::next}(list(i))) = list(i+1)\ \wedge \\ \mathsf{isValidVal}(\mathsf{SINT@::value}(\$\mathsf{Node::elem}(list(i)))) \end{pmatrix}\ .$$

The constructions for the second part of the function list2arr are much more involved because of the side-effects on the heap (allocating an array and deleting the linked list). For instance, $*$ocount should not point to any of the elem fields of the nodes of the linked list. We have also completely avoided the problem of what is *not modified* during the function execution.

9 Current Status and Further Work

At the moment of writing this text the type system, the signature, and the calculus described above are implemented in the KeY-C and we are at the stage of debugging the calculus and improving its usability. In essence, we can work with a large subset of C variable declarations, expressions, and we support **while** and **if** statements. However, recall that we only work with type-safe C programs. The program in our example can be reduced into FOL sequents, but we didn't have time to close all branches of the proof yet.

Supporting full C, of course, requires a lot more work to reason about numerous small and not-so-small features of the C language — **for** loops, **const** and **volatile** modifiers, string literals, typedefs, enumerations, const expressions, unions, bit-fields, varargs just to name a few. Of the more conceptual extensions we could name introducing modularity: translation units, extern and static variables, function calls, and function pointers. Luckily the C module system can be viewed as a special case of the JAVA module system, so taking over the calculus from the KeY for JAVA should be straightforward, although laborious. Calling function pointers can be implemented in the same way as polymorphic method calls in JAVA.

Another interesting direction is supporting C jump statements — **continue**, **break**, **return** and **goto**. Of those the first three are present in the KeY for JAVA and should be easy to take over. The calculus for the **goto** statement is yet to be developed, however, it seems that we can put together a reasonable calculus by combining the ideas of switch statements, exceptions and loop invariants, which after all are a **goto** statement in different disguises.

Further, an update calculus could be extended to support unions and deep value assignments in a more natural form. A deep update calculus would allow a surface syntax for deep object value updates of the form $\{\ o_1 :> o_2\ \}$ that would be semantically equivalent to copying all the members of object o_2 into o_1, e.g.

$$\left\{ \begin{array}{l} \mathsf{\$Node@::value(\$Node::next}(o_1)) := \mathsf{\$Node@::value(\$Node::next}(o_2)); \\ \mathsf{SINT@::value(\$Node::elem}(o_1)) \ := \mathsf{SINT@::value(\$Node::elem}(o_2)) \end{array} \right\} .$$

The former syntax is clearly more readable. The main challegne here is to develop rules for update simplification and application.

Similar ideas could be used to support reasoning about non-type safe raw memory access (as bytes). The idea is first to introduce a location representing individual bytes

$$\mathsf{byte} : \mathsf{Block}, \mathsf{int} \to \mathsf{UCHAR} \ ,$$

Now, for instance, ((**unsigned char**∗)o)[3] = 234; could be reduced into an update

$$\{\ \mathsf{byte}(\mathsf{objBlock}(o), \mathsf{objOffset}(o) + 3) := \mathsf{UCHAR::fromInt}(234)\ \} \ ,$$

where $\mathsf{objOffset}(o)$ is object's offset with respect to the beginning of the block. Now, we could introduce a surface syntax for deep object value updates of the

form $\{\ o_1 :> o_2\ \}$ that would be semantically equivalent to copying the bytes of object o_2 into o_1, i.e.

$$\left\{ \begin{array}{l} \mathsf{for}\ i;\ 0 \leq i < \mathsf{sizeof}(\mathsf{T}); \\ \ \ \mathsf{byte}(\mathsf{objBlock}(o_1), \mathsf{objOffset}(o_1) + i) := \\ \ \ \mathsf{byte}(\mathsf{objBlock}(o_2), \mathsf{objOffset}(o_2) + i) \end{array} \right\} .$$

What You see here is a quantified update over variable i satisfying the condition $0 \leq i < \mathsf{sizeof}(\mathsf{T})$, where $\mathsf{sizeof}(\mathsf{T})$ tells the size of object type T in bytes. The idea is that a type safe program would only require deep value updates of the former form, while in a non-type safe program only those places that work explicitly with bytes would require the latter kind of updates. Creating a dynamic logic for raw memory access without a deep update calculus would mean using only the latter kind of updates, which, we surmise, would be unusable for non-trivial programs.

Finally, in C any pointer can point to any memory location and consequently significant portions of program contracts and proofs are devoted to proving that object references cannot be aliased. We surmise that a static analysis could be developed to compute non-aliasing properties of the object references which could then be used to simplify the proofs. One possible way to use this information is to modify the CDL type system so that unalised references to the same C type receive distinct types in CDL that have the same structure (i.e. member or value accesor functions). Another very simple static analysis could detect scalar variable declarations that are never taken address of and prepend a **register** specifier to them.

10 Related Work

ACE (Assertion Checking Environment) [13], Caduceus [6], and the separation logic Isabelle/HOL approach described in [14] are all based on Hoare-style deductive verification using interactive theorem proving. Both ACE and Caduceus translate the C program (ACE expects a MISRA-C program while Caduceus can handle type-safe C programs) annotated with formal specifications to an intermediate language. In ACE, this intermediate product is given as input to the Stanford Temporal Prover (STeP). Caduceus uses it to generate the needed verification conditions which are then to be proved using one of several supported theorem provers (e.g. Coq and Simplify). In [14], the authors present a formal memory model for C programs and describe the implementation of this framework in the theorem prover Isabelle/HOL. The idea is to support both a low-level view of the heap as an array of bytes and two high-level views—multiple typed heaps and separation logic—and to unify these views in a sound way. This way, the verification approach can accurately reflect the real semantics of C, while well-behaved programs (that use pointers in a type-safe way) are still easy to reason about. Comparing these three methods to our approach reveals that, in ACE, Caduceus, and the Isabelle/HOL approach, interleaving of symbolic execution and logical reasoning is not possible and interaction does not take place

at the level of C source code but on intermediate code. As part of the Verisoft project (www.verisoft.de) a Hoare calculus and formal semantics of the C subset C0 on top of Isabelle/HOL were developed [12], however the verification of C programs is less automated than in KeY.

MAGIC [4], SLAM [1] and BLAST [7] are all verification approaches based on the abstract-verify-refine paradigm. An abstract model of the program is created using predicate abstraction and then model checking is used to check whether the abstraction conforms to the specification. If the verification step fails due to a too coarse abstraction, information gained from the failed verification step is used to refine the model and then the verification step is repeated. MAGIC verifies C programs with respect to finite state machine specifications and uses *simulation* as the notion of conformance. Both SLAM and BLAST check safety properties of C programs and use reachability analysis to check conformance. All three mentioned approaches assumes that the C program is type-safe. Another approach based on abstraction is Astrée [9], a static analyzer based on abstract interpretation [5]. Astrée focuses on subtle machine implementation aspects such as rounding errors introduced by floating-point arithmetics. Astrée can automatically discover all potential run-time errors of a certain class using a sound approximation of the C semantics. Compared to our approach, the abstraction-based methods described above are more automated than ours but less expressive. Verification of functional properties is not possible.

11 Summary

In this paper we briefly described the ongoing effort to develop KeY-C, the C target version of the verification system KeY. As a side-product of this work we expect to generalize KeY architecture for easily adding the support for new programming languages. The implementation is done in JAVA and we are using the Cetus framework [8] for parsing and analysing C programs. The JAVA target version of KeY is available from www.key-project.org. KeY-C is available from the authors on request. We would like to acknowledge the members of the KeY project for fruitful discussions.

References

1. Thomas Ball and Sriram K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Conference Record of POPL'02: The 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3, Portland, Oregon, January 2002.
2. Bernhard Beckert, Reiner Hähnle, and Peter Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2006.
3. Bernhard Beckert, Steffen Schlager, and Peter H. Schmitt. An improved rule for while loops in deductive program verification. In Kung-Kiu Lau, editor, *Proc. 7th International Conference on Formal Engineering Methods (ICFEM), Manchester, UK*, volume 3785 of *LNCS*, pages 315–329. Springer-Verlag, 2005.

4. Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering (TSE)*, 30(6):388–402, June 2004.
5. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM Press.
6. Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In Jim Davies, Wolfram Schulte, and Michael Barnett, editors, *Formal Methods and Software Engineering, 6th Intl. Conf. on Formal Engineering Methods, ICFEM, Seattle, USA*, volume 3308 of *LNCS*, pages 15–29. Springer-Verlag, 2004.
7. T.A. Henzinger, R. Jhala, R. Majumdar, and G. SUTRE. Software verification with BLAST. In *Proc. 10th Int. SPIN Workshop (SPIN2003), Portland, OR, USA, May 2003*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. Springer, 2003. Tool paper.
8. Sang Ik Lee, Troy A. Johnson, and Rudolf Eigenmann. Cetus—an extensible compiler infrastructure for source-to-source transformation. In Lawrence Rauchwerger, editor, *Languages and Compilers for Parallel Computing, 16th Intl. Workshop, College Station, TX, USA, Revised Papers*, volume 2958 of *LNCS*, pages 539–553, 2004.
9. Laurent Mauborgne. ASTRÉE: Verification of absence of run-time error. In René Jacquart, editor, *Building the information Society (18th IFIP World Computer Congress)*, pages 384–392. Kluwer Academic Publishers, Aug 2004.
10. MISRA Consortium. *MISRA-C: 2004 — Guidelines for the use of the C language in critical systems*, 2004.
11. Oleg Mürk and Daniel Larsson and Reiner Hähnle. KeY-C: A tool for verification of C programs. In *Proceedings of 21st Conference on Automated Deduction (CADE-21)*, July 2007. Accepted.
12. Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.
13. Babita Sharma, S. D. Dhodapkar, and S. Ramesh. Assertion checking environment (ace) for formal verification of c programs. In *SAFECOMP '02: Proceedings of the 21st International Conference on Computer Safety, Reliability and Security*, pages 284–295, London, UK, 2002. Springer-Verlag.
14. Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editors, *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 97–108, Nice, France, January 2007.