Extended High-Level C-Compatible Memory Model with Limited Low-Level Pointer Cast Support for Jessie Intermediate Language

Alexey Khoroshilov
Institute for System Programming of the RAS
Moscow, 109004, 25, A. Solzhenitsyn st.
khoroshilov@ispras.ru

Mikhail Mandrykin
Institute for System Programming of the RAS
Moscow, 109004, 25, A. Solzhenitsyn st.
mmandrykin@ispras.ru

Abstract—The paper presents an intermediate language which is intended to serve as a target analyzable language for verification of real-world production GNU C programs (Linux kernel modules). The language represents an extension of the existing intermediate language used by the JESSIE plugin for the FRAMA-C static analysis framework. It is compatible with the C semantics of arrays, initially supports hierarchical (prefix) pointer casts and discriminated unions, and extended with limited support for low-level pointer casts. The approach to translation of the original C code into the intermediate language and translation of the intermediate language into the input language of the Why3 deductive verification platform are explained by examples. The examples illustrate the expressive power of the extended intermediate language and efficiency of the resulting axiomatic representation.

Keywords—deductive verification, memory model, C programming language semantics, discriminated unions, hierarchical pointer casts, low-level pointer casts

I. INTRODUCTION

There are many techniques and tools for static verification of C source code. Among them are deduction techniques. The deduction techniques are based on translating of the original C source code supplied with specifications of the properties being verified into a set of logical formulas whose validity is equivalent to the validity of the original program with respect to the specified properties. These logical formulas a.k.a. verification conditions (VCs) or proof obligations can be proved valid with theorem provers, either completely automatic (SAT, SMT solvers, saturation-based provers) or involving some user interaction (proof assistants). Thus any deductive verification tool for C that aims fully automatic verification should somehow translate the semantics of the original code and provided specifications into a set of verification conditions that the existing automatic theorem provers are capable to discharge.

There are, however, intrinsic features of the C programming language that significantly complicate such translation. First, C is an imperative language and thus the implicit program state should be somehow modeled in the resulting logical formulas. It can be modeled in several ways which usually involve theory of arrays for representing the state of program memory. Second, C is a low-level language with generally untyped semantics, which influences the choice of an appropriate memory model in at least two contradictory ways.

On the one hand, C allows low-level type casts and reinterpretations of the same chunk of memory in many almost arbitrary ways. This motivates the choice of an untyped memory model based on a single or several arrays of 8-bit vectors precisely simulating the memory of a real computer. On the other hand, C also potentially allows almost arbitrary pointer aliasing. But while low-level pointer casts are quite often used in practice, at least in some common cases such as buffering or byte reordering, pointer aliasing is almost always restricted in a way that there is no unaligned pointers (i.e. pointers of some type that point to an address inside an object of that type) and objects in memory are disjoint (they can be embedded, but otherwise they don’t intersect). This encourages effective encodings of pointer aligning and object disjointness, which apart from specific logics (usually variants of separation logic [8]) often implies separate representation of differently-typed or knowingly disjoint memory locations by different arrays (a.k.a. the Burstall-Bornat model [21], [24], [25], e.g. one array per type, structure field or memory region). These arrays usually have logical value types corresponding to the original source program types they represent. Thus the essentially typed C memory model arises.

As a result, there exist C deductive verification tools implementing either an untyped or a typed memory model. In particular, the SIMPL [4] tool used in L4 [5] micro-kernel verification uses the untyped memory model while the VCC verifier (being used to verify the Microsoft Hypervisor) [6], [7] uses the typed one.

The JESSIE plugin [2] for the FRAMA-C static analysis framework [1] uses the typed memory model similar to that of VCC with several significant modifications.

First, the JESSIE memory model supports memory protection, including allocation and deallocation operations for dynamic (flexible) arrays of arbitrary length. The model was developed with intent to both ease the burden put on the theorem provers by the additional memory safety VCs generated for each memory access as well as to reduce the significant specification overhead put on the verification engineer by the need to provide additional memory safety specifications. For these purposes the model uses the combinations of the theories of linear integer arithmetic (LIA) with uninterpreted...
functions (UF) much more intensively than the theories of quantifiers and arrays. The model also reduces the number of variables used in the resulting conjunctions of inequalities. This both makes the resulting VCs easier for the theorem provers and potentially allows for simpler and more effective automatic annotation inference. JESSIE implements so-called local encoding of memory blocks to facilitate pointer validity checks in presence of pointer arithmetic.

Second, JESSIE maps structure/union fields to separately updatable arrays (memories) [21], [23], which significantly helps with anti-aliasing, discriminated unions\(^1\) and prefix (a.k.a. hierarchical) casts between pointers to structures\(^2\) [23]. Some researchers claim that hierarchical casts and discriminated unions together constitute about 99% of pointer casts in the majority of real-world C programs [9].

Both features make the JESSIE memory model suitable for many real world C codebases, but some important fragments in operating system kernel source code remain disregarded by the model. Unlike the model used in VCC, the JESSIE memory model lacks support for operations on bit-fields and data type reinterpretation. These features are vital for verification of Linux kernel modules, in particular the ones implementing file systems and network protocols where a lot of low-level encoding/decoding operations involve reinterpretations of memory chunks between arrays of bytes and arrays of structures back and forth.

The original thesis [2] describing the methods underlining the current implementation of the JESSIE tool suggests exploiting region inference [10]–[12] techniques together with a combination of high-level mathematical and low-level bit-wise memory models, each one restricted to the corresponding subset of inferred memory regions. So the memory of the program is divided into several disjoint parts a.k.a. regions i.e. sets of memory locations such that any two pointers aliased necessarily belong to the same region. This allows several different memory models to coexist in the same verification framework and to be still used independently for reasoning about properties of different memory regions.

The weak point of this approach lies in the principles behind the state-of-the-art automated theorem provers. They usually implement each of supported logical theories separately [13] and establish the interaction between the theories by propagating disjunctions of predicates among which only equalities are properly interpreted by all the theories involved. This implies that typically logical predicates or functions involving application of several theories at once, e.g. conversion of a mathematical integer to a fixed-length bit-vector or vice versa, are either entirely absent from the natively supported set of particular solver’s features or their support is very limited and inefficient. With such restrictions all the interaction between the involved low-level and high-level memory models expressed in compound inter-theory predicates lays upon the verification engineer (both in terms of providing specifications of such predicates and manually proving the VCs significantly dependent on their meaning). Besides the fact that the presence of pointer aliasing can propagate bit-wise regions quite far causing a significant part of memory locations to be encoded as bit-vectors that are in general significantly more complex to reason about. Another ubiquitous case of spreading the bit-wise encoding across the program is pointer arithmetic. Whenever a pointer is added an offset expressed as a bit-vector, the offsets added to this pointer in all other places are preferably encoded as bit-vectors as well. In addition to that, bit-vectors are currently unsupported by the latest version of the WHY3 [3] verification platform targeted by the JESSIE tool and the implementation of the bit-vector regions support in JESSIE itself is still mostly raw and incomplete.

Another shortcoming of the original JESSIE memory model is that its treatment of pointer shift operation turned out to be incompatible with the corresponding C counterpart in presence of prefix cast support presented in the thesis [2]. While in C shifting a pointer into an array of derived structures becomes wrong after casting the pointer to the type of the parent structure, the original JESSIE intermediate language semantics totally ignores this restriction and thus significantly limits the amount of programs correctly translatable into the intermediate language.

So the decision finally came to modify and extend the existing basic high-level and efficient JESSIE memory model. In order to make the further presentation of the memory model and its extension easy, we don’t preset the full JESSIE intermediate language, its syntax, typing rules and semantics, that are fairly redundant, complicated and not at all minimal. More thorough presentation of the full language can be found in [2]. We rather present the intermediate language, as well as the original and the suggested new memory models in a significantly reduced (simplified) form. The simplified intermediate language still allows us to emphasize the basic aspects of the full language and show how it can be extended with partial support for low-level memory access (pointer type reinterpretation), covering the most important use cases such as encoding/decoding operations and conversions between different byte orderings.

The paper first introduces the simplified intermediate language, a small toy analogue to the real JESSIE intermediate language (with its original memory model, as described in [2]), and then the extended simplified intermediate language, which extends the simplified language (and its memory model) with intent to provide enough support for some low-level pointer casts, while maintaining all previous advantages of the model. So the extension part of the presented language and its memory model corresponds to our contribution in regard to the full JESSIE intermediate language (and the respective implementations). The suggested approach is extensible to structures with bit-fields and interacts well with improvements made in the existing implementation of the Jessie plugin, i.e. bounded pointers, multiple inheritance hierarchies, and region inference.

II. THE SIMPLIFIED INTERMEDIATE LANGUAGE
A. Abstract syntax

The JESSIE translator is the tool that stands in the middle of the verification toolchain working sequentially in the following order: FRAMA-C frontend – JESSIE plugin – JESSIE translator – WHY3 IDE. The translator accepts the input program as a single file in its own specific input representation, the JESSIE intermediate language, which contains all the translation units

\(^1\)Discriminated unions are unions whose field addresses are not taken and in which only the last field written should be subsequently read.

\(^2\)Up- and downcasts (or prefix casts) are casts between two structure types, one of the structures being composed of the fields forming a prefix of the fields comprising the other structure.
of the original program merged together along with the spec-

Fig. 1. Abstract syntax of the simplified intermediate language

main()

be translated into the simplified intermediate language in order
dynamic memory allocation and a flexible array member can
annotated C program with a union, automatic (i.e. stack) and
the simplified intermediate language we consider how a sample
Now in order to explain the choice of features presented in
unions or structures are provided in Figure 1 with the corre-
value.
function calls, in particular their memory footprint.
Non-determinate value can be substituted with an
arbitrary integer value during evaluation, but it natu-
ally acquires precise semantics when the language is
analyzed using a deductive reasoning technique such as
weakest precondition calculus [14], [15]

• * stands for any binary operation on integer values;
• p stands for a pointer variable;
• f ∈ F is a unique structure field or union field label
from a finite set F of such labels;
• t ∈ T is a unique tag of a structure, a union or a field
of a union from a finite set of tags T. The finite sets
of labels and tags share no common elements;
• ∞ stands for any binary relation on integer values;
• ◊ stands for any binary logical connective;

The predicates omin(t_p) ∞ t_n, omax(t_p) ∞ t_n,
tag(t_p) = t, tag(t_p) ≤ t and operators assume ¯p
and assert ¯p (they are marked with a star) only apply
for the analyzable part of the language, so they have
no influence on its evaluation and so don’t have any
evaluation semantics.

All terms, predicates and operators involving pointers to
unions or structures are provided in Figure 1 with the corre-
ponding short explanations on the right (in italics).

We additionally make the following assumptions about the
sets F and T:

• The function ¯t : F → T is defined for every element
f ∈ F. It maps composite (structure or union) field
labels to the tags of the corresponding composites.
• The function τ : F → T differs from the function ¯t
on and only on the fields of unions. It maps union fields
(i.e. their labels) into their own tags rather than the
tags of the corresponding unions.
• The unary relation (predicate) ν ⊆ F divides all fields
belonging to F into two disjoint subsets of integer
(satisfying ν(f)) and pointer (satisfying ∼ν(f)) fields.
• The partial order [16] (reflexive, antisymmetric, tran-
sitive) relation ι ⊆ T × T is defined on the set T.
With respect to this relation partially ordered set T
constitutes a bounded join-semilattice [16] with top
element †. The relation ʻι corresponds to inheritance
(prefix) relation between structures which extends also
to unions and union fields so that (1) if the fields of
any structure with tag t_1 form a prefix of the fields of
any structure with tag t_2, we have t_2 ≤ t_1, and (2) for
the fields of unions we have τ(f) = t ⇒ t ≤ ʻτ(f).
So finally we obtain τ(f) ʻ≤ ʻτ(f) for any f ∈ F.

Now in order to explain the choice of features presented in
the simplified intermediate language we consider how a sample
annotated C program with a union, automatic (i.e. stack) and
dynamic memory allocation and a flexible array member can
be translated into the simplified intermediate language in order
to verify the postcondition of the function main().

The sample C program is as follows (see Figure 2 below):

Fig. 1. Abstract syntax of the simplified intermediate language

of the original program merged together along with the spec-
ifications provided by the user.

The JESSIE intermediate language is quite sophisticated
and not at all minimal since it’s designed to simplify the
translation of the original C program into it preserving as much
of the program’s initial structure as possible while performing
a number of important transformations primarily concerning
simplification of the program’s memory layout. So in the paper
we aren’t going to introduce the JESSIE intermediate language
itself, but rather use its dramatically simplified counterpart
capturing the basic capabilities of the full language that refer to
its memory model. The abstract syntax [19] of this simplified
intermediate language is presented in Figure 1.

In this figure the following notation is assumed:
• v stands for an integer variable;
• n ∈ ℤ is an integer value;
• * designates the non-determinate integer value. This
facility is included in the language in order to simulate
function calls, in particular their memory footprint.
Non-determinate value can be substituted with an
arbitrary integer value during evaluation, but it natu-
rally acquires precise semantics when the language is
analyzed using a deductive reasoning technique such as
weakest precondition calculus [14], [15]

• * stands for any binary operation on integer values;
• p stands for a pointer variable;
• f ∈ F is a unique structure field or union field label
from a finite set F of such labels;
• t ∈ T is a unique tag of a structure, a union or a field
of a union from a finite set of tags T. The finite sets
of labels and tags share no common elements;
• ∞ stands for any binary relation on integer values;
• ◊ stands for any binary logical connective;

The predicates omin(t_p) ∞ t_n, omax(t_p) ∞ t_n,
tag(t_p) = t, tag(t_p) ≤ t and operators assume ¯p
and assert ¯p (they are marked with a star) only apply
for the analyzable part of the language, so they have
no influence on its evaluation and so don’t have any
evaluation semantics.

All terms, predicates and operators involving pointers to
unions or structures are provided in Figure 1 with the corre-
ponding short explanations on the right (in italics).

We additionally make the following assumptions about the
sets F and T:

• The function ¯t : F → T is defined for every element
f ∈ F. It maps composite (structure or union) field
labels to the tags of the corresponding composites.
• The function τ : F → T differs from the function ¯t
on and only on the fields of unions. It maps union fields
(i.e. their labels) into their own tags rather than the
tags of the corresponding unions.
• The unary relation (predicate) ν ⊆ F divides all fields
belonging to F into two disjoint subsets of integer
(satisfying ν(f)) and pointer (satisfying ∼ν(f)) fields.
• The partial order [16] (reflexive, antisymmetric, tran-
sitive) relation ι ⊆ T × T is defined on the set T.
With respect to this relation partially ordered set T
constitutes a bounded join-semilattice [16] with top
element †. The relation ʻι corresponds to inheritance
(prefix) relation between structures which extends also
to unions and union fields so that (1) if the fields of
any structure with tag t_1 form a prefix of the fields of
any structure with tag t_2, we have t_2 ≤ t_1, and (2) for
the fields of unions we have τ(f) = t ⇒ t ≤ ʻτ(f).
So finally we obtain τ(f) ʻ≤ ʻτ(f) for any f ∈ F.

Now in order to explain the choice of features presented in
the simplified intermediate language we consider how a sample
annotated C program with a union, automatic (i.e. stack) and
dynamic memory allocation and a flexible array member can
be translated into the simplified intermediate language in order
to verify the postcondition of the function main().

The sample C program is as follows (see Figure 2 below):

Fig. 1. Abstract syntax of the simplified intermediate language

of the original program merged together along with the spec-
ifications provided by the user.

The JESSIE intermediate language is quite sophisticated
and not at all minimal since it’s designed to simplify the
translation of the original C program into it preserving as much
of the program’s initial structure as possible while performing
a number of important transformations primarily concerning
simplification of the program’s memory layout. So in the paper
we aren’t going to introduce the JESSIE intermediate language
itself, but rather use its dramatically simplified counterpart
capturing the basic capabilities of the full language that refer to
its memory model. The abstract syntax [19] of this simplified
intermediate language is presented in Figure 1.

In this figure the following notation is assumed:
• v stands for an integer variable;
• n ∈ ℤ is an integer value;
• * designates the non-determinate integer value. This
facility is included in the language in order to simulate
function calls, in particular their memory footprint.
Non-determinate value can be substituted with an
arbitrary integer value during evaluation, but it natu-
rally acquires precise semantics when the language is
analyzed using a deductive reasoning technique such as
weakest precondition calculus [14], [15]

• * stands for any binary operation on integer values;
• p stands for a pointer variable;
• f ∈ F is a unique structure field or union field label
from a finite set F of such labels;
• t ∈ T is a unique tag of a structure, a union or a field
of a union from a finite set of tags T. The finite sets
of labels and tags share no common elements;
• ∞ stands for any binary relation on integer values;
• ◊ stands for any binary logical connective;

The predicates omin(t_p) ∞ t_n, omax(t_p) ∞ t_n,
tag(t_p) = t, tag(t_p) ≤ t and operators assume ¯p
and assert ¯p (they are marked with a star) only apply
for the analyzable part of the language, so they have
no influence on its evaluation and so don’t have any
evaluation semantics.

All terms, predicates and operators involving pointers to
unions or structures are provided in Figure 1 with the corre-
ponding short explanations on the right (in italics).

We additionally make the following assumptions about the
sets F and T:

• The function ¯t : F → T is defined for every element
f ∈ F. It maps composite (structure or union) field
labels to the tags of the corresponding composites.
• The function τ : F → T differs from the function ¯t
on and only on the fields of unions. It maps union fields
(i.e. their labels) into their own tags rather than the
tags of the corresponding unions.
• The unary relation (predicate) ν ⊆ F divides all fields
belonging to F into two disjoint subsets of integer
(satisfying ν(f)) and pointer (satisfying ∼ν(f)) fields.
• The partial order [16] (reflexive, antisymmetric, tran-
sitive) relation ι ⊆ T × T is defined on the set T.
With respect to this relation partially ordered set T
constitutes a bounded join-semilattice [16] with top
element †. The relation ʻι corresponds to inheritance
(prefix) relation between structures which extends also
to unions and union fields so that (1) if the fields of
any structure with tag t_1 form a prefix of the fields of
any structure with tag t_2, we have t_2 ≤ t_1, and (2) for
the fields of unions we have τ(f) = t ⇒ t ≤ ʻτ(f).
So finally we obtain τ(f) ʻ≤ ʻτ(f) for any f ∈ F.

Now in order to explain the choice of features presented in
the simplified intermediate language we consider how a sample
annotated C program with a union, automatic (i.e. stack) and
dynamic memory allocation and a flexible array member can
be translated into the simplified intermediate language in order
to verify the postcondition of the function main().

The sample C program is as follows (see Figure 2 below):
```c
#include <stdlib.h>

struct parent {
    int id;
};

union data {
    char c;
    int n;
};

struct child {
    struct parent p;
    union data data;
    char flex[1];
};

/*@ requires \offset_min(flex) \leq 0 &&
@ \\ offset_max(flex) \geq 0;
@ requires
@ \\typeof(flex[0]) \lt \typeof(char); @
@ assigns flex[0]; @
@ ensures flex[0] \geq 0;
@ */
void init_flex(char *flex)
{
    flex[0] = 0;
}
//@ ensures \result = 0;
int main()
{
    struct child ch[2];
    ch[0].p.id = 0;
    struct parent *pp =
        ((struct child *)pp)[1].p.id = 1;
    ((struct child *)pp)[1].data.c = 'c';
    pp =
        malloc(sizeof ch[0] +
            2 * sizeof(char));
    init_flex(((struct child *)pp)->flex);
    return
        ((struct child *)pp)->flex[1] - 1;
}
```

Fig. 2. The sample C program

The simplified intermediate language doesn’t support embedding. In the example program there are three cases of embedding, namely:

- each structure with tag parent is embedded into each structure with tag child as the first field with label p;
- each union with tag data is embedded into each structure with tag child as the field data (of the same name);
- each structure with tag child has a final flexible array member labeled flex. Flexible array members are in fact embedded arrays of variable length, so this case can also be treated as an embedding of an array into a structure.

The last two cases of embedding can be automatically translated into indirectly-accessed and explicitly allocated and deallocated union and array respectively (a flexible array member of default zero size can be translated into a null pointer).

Any array in JESSIE (as well as in our intermediate language) is always an array of structures. Therefore to represent the flexible array member of type char[] we introduce a dummy structure tag char with one field m of type char.

In the first case of embedding, however, the better option is to take advantage of the prefix cast support in the intermediate language by embedding the field of structures with tag parent into the structures with tag child in place of the first field parent. After this transformation parent and child structures become hierarchically related so that fields of structure parent form a prefix of the fields of structure child. Prefix casts can be entirely omitted in the simplified
intermediate language (but not in the full JESSIE language), so the assignment with cast in lines 35–36 can be translated into the pointer assignment operator (→). Finally, the stack allocation of an array of structures in line 33 can be translated likewise the embedded fields into an explicitly allocated and deallocated array of structures.

The original sample program is annotated with ACSL [17], [18] specifications in lines 18–23 and line 30. The predicates used in the annotations of the sample program correspond directly to the predicates (omin, omux, tag = and tag ≤) presented in the abstract syntax of the intermediate language. So for now we translate the function call in line 43 into two assertions (assert operators) corresponding to the ACSL requires clauses (in lines 18–21) and an update with a non-deterministic value (upd operator) corresponding to the assigns clause (in line 22) followed by the assumption of the init_flex function postcondition specified by the ensures clause (in line 23), which is translated into corresponding assume operator. The postcondition of the function main (line 30) is translated into the final assert operator. We explain the meaning of the omin, omux and tag predicates in the next section after introduction of the pointer and memory model of the simplified intermediate language.

To finish the translation of the sample program into the intermediate language we need to establish the required sets T and F along with the functions τ : F → T and τ : F → T, the predicate ν ⊆ F and the partial order ⪯ ⊆ T × T. The finite set of tags T for the sample program consists of its structure, union and union field tags (including the dummy tag for structure char):

\[ T = \{ t_{\text{parent}}, t_{\text{data}}, t_{\text{data . c}}, t_{\text{data . n}}, t_{\text{child}}, t_{\text{char}} \} \].

The partial order ⪯ establishes the inheritance relation between the structures parent and child and between the union data and its fields c and n. It also includes its least upper bound T:

\[ \leq = \{ (t_{\text{child}}, t_{\text{parent}}), (t_{\text{data . c}}, t_{\text{data . n}}), (t_{\text{data . n}}, t_{\text{data}}), (t_{\text{parent}}, T), (t_{\text{char}}, T), (t_{\text{data}}, T) \} \].

The remaining functions τ and τ̄ and the predicate ν are defined straightforwardly by their corresponding definitions:

\[ τ = \{ f_{\text{parent . id}} \mapsto t_{\text{parent}}, f_{\text{data . c}} \mapsto t_{\text{data . c}}, f_{\text{data . n}} \mapsto t_{\text{data . n}}, f_{\text{child . data}} \mapsto t_{\text{child}}, f_{\text{child . flex}} \mapsto t_{\text{child}}, f_{\text{char . m}} \mapsto t_{\text{char}} \} \]

\[ τ̄ = τ[f_{\text{data . c}} \mapsto t_{\text{data . c}}, f_{\text{data . n}} \mapsto t_{\text{data}}] \]

\[ ν = \{ f_{\text{parent . id}} : f_{\text{data . c}} : f_{\text{data . n}} : f_{\text{char . m}} \} \]

Allocation and pointer shift operations (alloc and shift) of the intermediate language require explicit structure or union tags. The reasons for this are explained further in the next section. In general, this requirement conforms to the limitations of the underlying memory model, but in the case of the example program the required explicit tags are easily recoverable (the tag in the shift operation is recoverable in general by static typing). We also note here that actual full JESSIE intermediate language distinguishes between different integer types by modeling them with bounded mathematical integers. However, in our considerations we omit this typing along with the necessary bounds checks for the sake of simplicity.

So we finally obtain the program in the simplified intermediate language presented in Figure 3.

B. Memory model

According to its original description [2], the JESSIE memory model is based on byte-level block memory model, which is intended to model memory protection. In this model pointers are represented with triples \( (α, l, o) \), where \( α \) represents the absolute address of a pointer i.e. the value that can be potentially obtained by casting the pointer to the appropriate integral type (suppose an unsigned one for clarity), \( l \) is a unique label of a memory block, which is ascribed to a pointer at the point of allocation and remains invariant under pointer shifts, and \( o \) is the offset of a pointer from the beginning of its memory block \( l \). The model assumes that pointers that belong to different memory blocks are always unequal, since this property is ensured by the uniqueness of the block labels. However, the absolute addresses of two pointers from two different memory blocks can be equal to each other, including the cases with one or two invalidated pointers belonging to deallocated memory blocks (as shown in picture 4).

Each memory block is ascribed with its current length denoted by \( A[l] \) and varying during evaluation. A pointer is considered valid if and only if its offset satisfies the relations \( 0 \leq o < A[l] \). Only valid pointers can be dereferenced so that a program can only read from or write into the memory accessible through valid pointers. No distinction is made by the model between read-only and writable memory, static, stack or heap-allocated memory or between user-space and kernel-space memory. Pointer subtraction is only allowed between pointers to the same memory block and pointer comparison for equality is also allowed between two arbitrary valid pointers (where the model semantics matches that of C).

However, the memory model presented above is not compatible with both its support for hierarchical pointer casts and the semantics of pointer shifts in the C language.

Let’s consider the valid pointer \( ch \) of type struct child * in line 33 of the example program (Figure 2, recall that \( ch \) is retyped into a pointer after transformation, see Figure 3) pointing to the array of child structures of length 2. Let’s assume it’s modeled by the pointer \( p = (α, l, 0) \), \( A[l] = 2 \). Then the pointer \( pp \) of type struct
parent * initialized in line 35 with the pointer ch is modeled by the same pointer p. Here we get a contradiction: while in the original program the expression pp[2] represents a valid (though misaligned) pointer somewhere into the structure pointed by ch, in the model we get shift(p, t_{parent}, 2) = (α', l, o'), where α' = 0 + 2 = 2 = A[l], while a valid pointer should satisfy the relation o' < A[l].

For that reason we prevent misaligned pointers in our memory model by extending it with another attribute ascribed to pointers themselves i.e. pointer tag label denoted by $T[(α, l, o)]$. A tag label corresponds to the precise runtime type of a structure or a union field addressed by a pointer at the current point of the program evaluation. These labels are only changed as a result of allocation, deallocation or union field assignment operators (a.k.a. strong updates). So they remain invariant under hierarchical pointer casts that thus can be made entirely transparent. Pointer shifting, on the contrary, must be guarded by check of the corresponding pointer tag to prevent alignment violation. So that’s the reason why the pointer type tags are made explicit in the pointer shift operation of the intermediate language.

The transparency of hierarchical pointer casts determines the need of runtime checks at pointer dereferences. These checks are avoided in the implementation, where explicit pointers casts are made mandatory by static typing and runtime checks for hierarchical pointer casts are introduced.

The absence of misaligned pointers allows high-level memory representation with a separately updatable array of memory per each structure/union field [21], [23] as long as we refuse to maintain the original C language semantics in presence of non-hierarchical pointer casts.

The abstract syntax of the simplified intermediate language introduced in the previous section doesn’t provide any pointer-to-integer conversion functions, so from here onwards we omit the absolute addresses α when referring to pointers (as their absolute addresses thus have no effect on the evaluation).

C. Semantics

The simplified intermediate language (just as the original JESSIE) is supposed to be executed on an imaginary non-deterministic machine. All possible executions of a program on such an imaginary machine are restricted by the assumption operators (assume) obtained by translating the ACSL annotations provided by the user. Then the resulting set of all possible executions is analyzed by generating and discharging the verification conditions corresponding to the preconditions of the operations involved in the evaluation process as well as the assertion operators (assert) obtained from the provided annotations.

So the simplified intermediate language is devised with intent to have easily analyzable semantics rather than easily executable one (hence the point of translating the original C code into it). The semantics of the intermediate language provides the VC generator and thus also the theorem prover with efficient representation for most of the memory separation conditions by translating all memory read and write operations into select and store operations on distinct (logical) arrays, one array per a structure/union field [21], [23]. This is reflected in the semantics where we represent heap memory by the map $M : F → M_n ∪ M_p$, where

\[ ∀ M[f] ∈ M_n, M[f] : F ↪ Z \text{ and} \]
\[ ∀ M[f] ∈ M_p, M[f] : F ↪ P, \text{ where} \]
\[ P = \{ (l, o) | l ∈ L, o ∈ \mathbb{Z} \} \]

is the set of pointers and $L ≃ \mathbb{N} ∪ \{0\}$ is the infinite numerable set of distinct memory block labels (we can also directly use the set $\mathbb{N} ∪ \{0\}$ for labels).

However, the representation of mappings $A$ and $T$ in the semantics is not quite efficient for further analysis. To make their representation significantly more efficient the JESSIE tool implementation applies a number of important optimizations (at the stage of JESSIE-to-WHY3ML [20] translation) that we describe further.

The initial state of the imaginary machine is restricted by the following constraints:

\[ A[l_{null}] = 0, \forall p ∈ P, T[p] = T. \]

Any possible execution of the example program (Figure 3) started from this initial state does not get stuck [19] and its final state can be expressed by the following set of conditions:

$A \supseteq \{ (l_{ch} ⇒ 0, l_{ch}[0].data ⇒ 0, l_{ch}[1].data ⇒ 0, l_{malloc} ⇒ 1, l_{malloc}⇒ flex ⇒ 2) \}

$T \supseteq \{ (l_{child}, l_{ch}[0].data ⇒ 0 ⇒ t_{data}, c, (l_{ch}[0].data ⇒ 0 ⇒ t_{data}, l_{ch}[1].data ⇒ 0 ⇒ t_{data}, (l_{malloc}, l_{child}, l_{malloc}⇒ flex, ⇒ t_{char}) \}

$M \supseteq \{ f_{parent.id} ⇒ \{ (l_{ch}, 0 ⇒ 0, (l_{ch}, 1 ⇒ 1, \cdots, f_{child.data} ⇒ \{ (l_{ch}, 0 ⇒ (l_{ch}[0].data ⇒ 0, (l_{ch}, 1 ⇒ (l_{ch}[1].data ⇒ 0, \cdots, f_{data}.c ⇒ \{ (l_{ch}[0].data ⇒ 0 ⇒ 99, \cdots, f_{child.flex} ⇒ \{ (l_{ch}, 0 ⇒ (l_{null}, 0, (l_{ch}, 1 ⇒ (l_{null}, 0, \cdots, (l_{malloc}, 0 ⇒ (l_{malloc}⇒ flex, 0, \cdots) \}\}\}\}\}\}\}\}\}\}\}

To verify the example program with respect to the user-provided annotations we first apply two most important optimizations implemented in JESSIE: local encoding of pointers in order to optimize the encoding of the map $A$ and separation between pointers to unions and structures in order to optimize the encoding of the map $T$.

First, we notice that we can only consider mappings in the map $A$ for blocks that are accessible through pointers, as other mappings don’t influence evaluation. Then we can replace the map $A : L → Z$ with the map $A' : P → Z$ such that $∀(l, o) ∈ P, A'[l, o] = A[l]$. Next we can notice that now all the three mappings $A$, $T$, and $M$ are defined on the set $P$. Thus we can hide the internal structure of pointers as pairs behind an abstract type by introducing a function $o : P → Z : ∀(l, o) ∈ P, o(l, o) ⇒ o$. Then by substituting for any pointer $p$ the function $o(p)$ and the map $A'[p]$ with two functions $οmin(A, p) = −ο(p)$ and $οmax(A, p) = Α'[p] − o[p] − 1$ we obtain the local encoding for pointers. This encoding is called local because for analyzing a function that performs
some operations on pointers (e.g. \texttt{shift}, \texttt{psub}, \texttt{acc}, \texttt{upd}) it’s more efficient to operate with inequalities on the minimal
and maximal offsets of pointers than the identical inequalities
on their offsets and block lengths (e.g. \texttt{omax}(A,p) \geq 0 \text{ vs.}
\textit{A}[p] - \textit{o}(p) - 1 \geq 0). This encoding first appeared in the
CADECUS deductive verification tool [21], [22] and is also
used in JESSIE, where the functions \texttt{omin} and \texttt{omax}
are encoded as uninterpreted functions of two arguments with
the appropriate set of axioms.

Second, we notice that as we only allow moderated
unions\footnote{A moderated union in C is a union whose field addresses are not taken.}, pointers to structures are never aliased with point-
ers to unions. But for pointers to structures the following
invariant is always maintained: \( \forall p \in \mathbb{P}, \forall i \in \mathbb{Z}. \text{\texttt{T}[p]} = \text{\texttt{T[shift} \{p, \text{\texttt{T}[p][i]}\}} \). This suggests replacing the map \texttt{T} with a function \texttt{tag} for pointers to structures and a map \texttt{T}_\texttt{u} similar to the original \texttt{T} for unions. Then the function \texttt{tag} is encoded as an uninterpreted function of two arguments with several necessary axioms. This significantly
reduces function preconditions since instead of requiring ap-
propriate tag for each element of the range of pointers accessed
by a function we can only require it for just one arbitrary
pointer into the same array.

Now after applying these two optimizations we can analyze
the sample program in the simplified intermediate language (see Figure 3) with respect to annotations represented by the
assert and \texttt{assume} operators. The semantics of predicates
involving functions \texttt{omin}, \texttt{omax} and \texttt{tag} is now clear since
it exactly corresponds with the semantics of the uninterpreted
functions \texttt{omin}, \texttt{omax} and \texttt{tag}. As a result we can refine the
final state of the program (based on the \texttt{assume} operator)
by adding the constraint \( \text{\texttt{M[}f_{char.m}[\text{\texttt{p}}]} \geq 0. \)
The first two assertions in the program can be proven valid while the
last one can be easily refuted by a counterexample where
\( \text{\texttt{M[}f_{char.m}[\text{\texttt{p}}]} \neq 1. \)

II. THE EXTENDED SIMPLIFIED INTERMEDIATE
LANGUAGE

A. Motivation and abstract syntax

While we were able to successfully verify the sample inter-
mediate program in Figure 3, thus also offering a reasonable
(although not quite rigorous) judgment about the correctness
of the original C program in Figure 2 with respect to user-
provided annotations, many real C code examples involving
lower-level memory manipulation are not expressible in the
simplified intermediate language we presented. One example
of such program can be obtained by modifying functions
\texttt{init\_flex} and \texttt{main} of the original program in the fol-
lowing way:

```c
// main() {
    pp = malloc(sizeof char[0] + 5 * sizeof(char));
    init\_flex((struct parent *) ((struct child *)pp)->flex);
    ((struct child *)pp)->flex[4] = 'c';
}
```

So in order to get this modified program verified we suggest
extending the simplified intermediate language with two new
capabilities — memory \texttt{reinterpretation} and memory block
\texttt{ripping}.

The intuitive meaning of the former capability is transform-
ing a memory block allocated for one structure or union type
into a memory block for another such type provided that (1)
both structure/union types do not have pointer fields and that
either (2a) the size of the source type is a multiple of the size
of the destination one (\texttt{splitting} inside a memory block) or that
(2b) besides the reverse of this multiplicity (now the size of the
destination type is a multiple), the size of the original block
is also a multiple of the size of the destination type (\texttt{joining}
inside a memory block).

Being able to do this alone, however, is still insufficient
in case of our modified program as it involves \texttt{reinterpretation}
of a memory block of size 5 from type \texttt{char} (retyped to
\texttt{struct char} to type \texttt{struct parent} which is neither
a case of splitting (because the size of type \texttt{char} is less
than the size of type \texttt{struct parent} and obviously is not a multiple of it) nor a case of joining (because 5 is
not a multiple of \texttt{sizeof} \texttt{(struct parent)}, which is
typically 4). Here we can apply the latter capability, whose
intuitive meaning is in splitting the original memory block
just before the \texttt{reinterpretation} into two continuous parts —
an accessible and a ghost one and then joining these parts
back together after the necessary memory \texttt{reinterpretations}
are done and the types of the parts match again.

The rationale behind the second capability is suggested by
the ability to have ghost pointer variables that can address
(point to) some obscure memory blocks, inaccessible by the
original program, though almost indistinct from the ordinary
blocks in the translated intermediate language program. We use
the term “memory block \texttt{ripping}” rather than “memory block
\texttt{splitting}” because the latter term is rather associated with the
\texttt{corresponding} kind of memory \texttt{reinterpretation} operation. The
term “ripping” also suggests the essence of the operation that
is ripping the temporarily redundant part of a memory block
into a separate ghost memory block accessible through a ghost
variable and then “mending” this part back again in order to
restore the full capacity of the original memory block.

With this intuitive considerations in mind we can translate
the modified C program into the extended intermediate lan-
guage presented in Figures 1 and 5 (the most interesting part
of the resulting intermediate program is shown in Figure 6).

The extension introduces a new function \texttt{cast}, which is,
though not mandatory to syntactically place a pointer to one
composite type in place where a pointer to another arbitrary
one is semantically required. The corresponding restriction
unavoidably must be enforced in the extended simplified inter-
mediate language due to its untyped semantics. But the use of
the function is mandatory for the resulting program not to get
\[ t_p := \ldots | \text{cast}(t_p, t, t) \text{ pointer reinterpretation cast} \\
* \text{rip}(t_p, t_p) \text{ memory block ripping} \]

\[ o := \ldots | \text{rmem}(t_p, t, t) \text{ memory block splitting/joining} \\
* \text{mend}(t_p, t_p) \text{ memory block mending} \]

Fig. 5. Extension of the simplified intermediate language abstract syntax

The function **rip** and the operators **mend** and **rmem** correspond to the proposed memory “ripping/mending” and “splitting/joining” operations. The function **rip** accepts two arguments – a pointer into the destined memory block, usually subject for further joining, and a greater pointer into the same memory block pointing at the offset, where the “odd” (temporarily unnecessary, but hindering the join) part of the block starts. The result of the function is a (ghost) pointer to the start of a new memory block, representing the detached ending part of the original block. This pointer is intended to be saved in a ghost variable for future use in the corresponding **mend** operator. The pointer into the original memory block (the first argument) remains unchanged by this function. This pointer then can be used in the following **cast** operator. The pointer into the original memory block **mend** be saved in a ghost variable for future use in the corresponding **mend** ending part of the original block. This pointer is intended to the start of a new memory block, representing the detached ending part of the original block. This pointer is intended to be saved in a ghost variable for future use in the corresponding **mend** operator. The pointer into the original memory block (the first argument) remains unchanged by the function. This pointer then can be used in the following **cast** operator, which is analogous to *cast* except it reinterprets (modifies) the memory block and pointer attributes (the A and T maps) as a side effect rather than modifying the pointer. After the reinterpretation the pointer obtained from the *cast* function becomes a pointer into the new valid memory block and can be used for whatever needed. The original memory block, meanwhile, stays temporarily (or even permanently) invalidated. The subsequent another **rmem** operator can be used swap back the validity of the two blocks while simultaneously updating the corresponding memory along with the block and pointer attributes. Finally, the **mend** operator can be used to “stick back” the original block from its accessible (and possibly modified) and ghost parts. The resulting translation of the lines 1-6 from the modified fragment (see above) of the original program (Figure 2) is shown in Figure 6. Here we use the ghost pointer \( g \) to temporarily address the ripped part of the dynamically allocated memory block in the program.

So we extended the intermediate language presented in Figure 1 with two additional functions, **cast** and **rip**, and two operators, **rmem** and **mend**. The function **cast** expresses non-hierarchical (reinterpretation) pointer casts, the operation **rmem** represents both possible memory block reinterpretation operations — block joining and splitting, and the function **rip** and the operation **mend** correspond to memory ripping and mending respectively. With the help of these four new constructions we can express the intention of the modified sample program in the extended intermediate language as shown in Figure 6.

**B. Extended memory model**

The functions **cast** and **rip** and the operations **rmem** and **mend** acquire precise semantics in the context of the appropriately extended memory model.

To introduce the corresponding semantics we first need to extend the memory model presented above with block reinterpretation and ripping. To support memory reinterpretation we introduce a new model function \( \varphi : \mathbb{T} \times \mathbb{T} \rightarrow \mathbb{Z} \). The function is defined on ordered pairs of tags in the following way:

- if
  - the tags \( t_1 \) and \( t_2 \) are structure, union or union field tags;
  - the size \( s_{t_1} \) of the first structure or union with tag \( t_1 \) is a multiple of the size of the second one \( s_{t_2} \);
  - both of the tags correspond to either union fields or structures/unions without pointer fields, and
  - there is a logical predicate \( \text{rmem}_{t_1,t_2}(M,T,l,l',m) \) representing the high-level semantics of the low-level (i.e. bit-wise) equality between the values of the fields of a structure/union or a union field with tag \( t_1 \) and the fields of a structure/union or a union field with tag \( t_2 \). This predicate can be defined only as needed, e.g. for casts to and from dummy char structures, and left undefined for more complicated cases such as casts between structures with different field alignments.

then \( \varphi(t_1, t_2) = s_{t_1} \div s_{t_2} \);
- if, instead of the second condition, the size if the second structure or union field \( s_{t_2} \) is a multiple of...
the size of the first one \((s_1)\) and the three remaining conditions are met, then \(\varphi(t_1, t_2) = -(s_2 \oplus s_1)\);

- \(\varphi(t_1, t_2) = 2\), otherwise.

Another new model function \(\pi : L \times T \rightarrow L\). We supplement unique block labels with tags from \(T\). For allocation (\textit{malloc}) operations these label tags must be equal to the tags used in the corresponding operations assigning the labels to their corresponding memory blocks. So a unique label \(l_{t,t}\) with tag \(t\) can be assigned only to a memory block allocated for an array of structures or unions with the same tag \(t\) (a single structure/union is a special case of array). The labels are also grouped so that for each label \(l_{t,t}\) the is a right-adjacent ghost label \(l'_{t,t}\) and a collection of reinterpretation labels \(\{l_{t,t'} | t' \in T\}\), where any two labels \(l_{t,t} \neq l_{t',t'}, i \neq j \neq t \neq t'\). Then the function \(\pi\) maps a label \(l_{t,t}\) to its corresponding reinterpretation label \(l_{t,t'}\) so that \(\pi(l_{t,t'}) = l_{t,t'}\). The reinterpretation labels are distinguished from their original counterparts only in the precondition of the alloc operation, so in all other cases there’s no difference between a memory block obtained initially by actual allocation and a block reinterpreted from another one with a different tag by using the \textit{rmem} operation. A minimal predicate (binary relation) \(\beta \subseteq L \times T\) such that \(\beta(l_{t,t}, t)\) holds for any \(l_{t,t} \in L\) is used in the alloc operation to ensure that a freshly assigned label \(l_{t,t}\) belongs to a new distinct group (and so has the corresponding right-adjacent ghost label \(l'_{t,t}\) and all the necessary reinterpretation labels \(l_{t,t'}\)). The last function we introduce, \(\rho : L \rightarrow L\) maps a label \(l_{t,t}\) to its corresponding right-adjacent ghost label \(l'_{t,t}\) and is used in the \textit{rip} function and the \textit{mend} operation.

With the three new functions \(\varphi\), \(\pi\) and \(\rho\) and a predicate \(\beta\) we can formalize the semantics of the new functions cast and rip and the operations \textit{rmem} and \textit{mend}. The semantics is intended to preserve the invariant that exactly the memory addressed by valid pointers in the original program can be accessed in the corresponding intermediate language program. This is indeed true for our sample program if we also include memory accessible through the ghost pointer \(g\). Besides the evaluation semantics, the operation \textit{rmem} also has additional implicit semantics at the VC generation stage. The operation has a postcondition \textit{rmem} \((t_1, t_2) \rightarrow \{M, T, l', A[l]\}\) that is treated as an \textit{assume} operator inserted just after the \textit{rmem} operation.

For the modification of the sample program translated into the extended intermediate language as shown in Figure 6 we define the functions \(\varphi\), \(\pi\) and \(\rho\) and the predicate \(\beta\) in the following way:

\[
\begin{align*}
\varphi(t_{\text{char}}, t_{\text{parent}}) &= -4, \\
\varphi(t_{\text{parent}}, t_{\text{child}}) &= 4, \\
\varphi(t, t') &= 0, \\
\{t, t'\} &\not\subseteq \{t_{\text{parent}}, t_{\text{child}}\}, \\
\pi(l_{\text{malloc}}) &= \text{flex}, t_{\text{char}}, t_{\text{parent}}, t_{\text{parent}}, t_{\text{parent}}, t_{\text{parent}}, t_{\text{parent}}, t_{\text{parent}}, t_{\text{parent}}, t_{\text{parent}} \\
\pi(l_{\text{malloc}}) &= \text{flex}, t_{\text{char}}, t_{\text{parent}}, t_{\text{parent}}, t_{\text{parent}}, t_{\text{parent}}, t_{\text{parent}}, t_{\text{parent}}, t_{\text{parent}} \\
\rho(l_{\text{malloc}}) &= \text{flex}, t_{\text{char}}, t_{\text{parent}}, t_{\text{parent}}, t_{\text{parent}}, t_{\text{parent}}, t_{\text{parent}}, t_{\text{parent}}, t_{\text{parent}} \\
\beta &\subseteq \{(l_{\text{ch}, t_{\text{child}}, \text{child}},\ (l_{\text{ch}[0], \text{data}, t_{\text{data}}, \text{data}), \\
(l_{\text{ch}[1], \text{data}, t_{\text{data}}, \text{data}), \\
(l_{\text{malloc}}(l_{\text{ch}, t_{\text{child}}, \text{child}})), (l_{\text{malloc}}(\text{flex}, t_{\text{char}}, t_{\text{parent}}))\}
\end{align*}
\]

The final state of the maps \(A\), \(T\) and \(M\) at the end of any possible evaluation of the intermediate language program in Figure 6, if restricted by the given \textit{assume} operators and the \textit{rmem} operation postcondition, can be expressed by the following constraints:

\[
\begin{align*}
A &\supseteq \{(l_{\text{ch}, t_{\text{child}}}, \rightarrow 0), (l_{\text{ch}[0], \text{data}, t_{\text{data}}, \rightarrow 0), (l_{\text{ch}[1], \text{data}, t_{\text{data}}, \rightarrow 0), \\
& (l_{\text{malloc}}(l_{\text{ch}, t_{\text{child}}}), \rightarrow 1), (l_{\text{malloc}}(\text{flex}, t_{\text{char}}, t_{\text{parent}}), \rightarrow 5)\}
\end{align*}
\]

\[
\begin{align*}
T &\supseteq \{(l_{\text{ch}, t_{\text{child}}, \cdots} \rightarrow t_{\text{child}}, (l_{\text{ch}[0], \text{data}, t_{\text{data}}, \rightarrow 0) \rightarrow t_{\text{data}, c}, \\
& (l_{\text{ch}[0], \text{data}, t_{\text{data}}, \rightarrow 0), (l_{\text{ch}[1], \text{data}, t_{\text{data}}, \rightarrow 0), \\
& (l_{\text{malloc}}(\text{flex}, t_{\text{char}}, t_{\text{parent}, \cdots} \rightarrow t_{\text{char}, c}, (l_{\text{malloc}}(\text{flex}, t_{\text{char}}, t_{\text{parent}, \cdots} \rightarrow t_{\text{parent}}\}
\end{align*}
\]

\[
\begin{align*}
M &\supseteq \{(l_{\text{parent}, \text{id}} \rightarrow 0), (l_{\text{ch}, t_{\text{child}}, \rightarrow 0), (l_{\text{ch}, t_{\text{child}}, \rightarrow 1), \\
& (l_{\text{malloc}}(\text{flex}, t_{\text{parent}}, \rightarrow 0) \rightarrow 256, \cdots), \\
& f_{\text{child}, \text{data}} \rightarrow (l_{\text{ch}, t_{\text{child}}, \rightarrow 0), (l_{\text{ch}[0], \text{data}, t_{\text{data}, \rightarrow 0}, \\
& (l_{\text{ch}, t_{\text{child}}, \rightarrow 1), (l_{\text{ch}[1], \text{data}, t_{\text{data}, \rightarrow 0}, \\
& f_{\text{data}, c} \rightarrow (l_{\text{ch}[0], \text{data}, t_{\text{data}, \rightarrow 0) \rightarrow 99, \cdots), \\
& f_{\text{child}, \text{flex}} \rightarrow (l_{\text{ch}, t_{\text{child}}, \rightarrow 0), (l_{\text{null}, \rightarrow 0), \\
& (l_{\text{ch}, t_{\text{char}}, \rightarrow 1), (l_{\text{null}, \rightarrow 0), \\
& (l_{\text{malloc}}(l_{\text{ch}, t_{\text{child}}, \rightarrow 0), (l_{\text{malloc}}(\text{flex}, t_{\text{char}}, t_{\text{parent}, \rightarrow 0), \\
& f_{\text{char}, \text{m}} \rightarrow (l_{\text{malloc}}(\text{flex}, t_{\text{char}}, t_{\text{parent}, \rightarrow 0) \rightarrow 0, \\
& (l_{\text{malloc}}(\text{flex}, t_{\text{char}}, t_{\text{parent}, \rightarrow 1) \rightarrow 1, \cdots)\}
\end{align*}
\]

These constraints prove the final assertion of the intermediate language program (Figure 6) valid. So the extension of the intermediate language with block reinterpretation and ripping allows one to prove the validity of some programs.
involving low-level memory access. However, the semantics of the extended simplified intermediate language allows only ripping of the rightmost part of a memory block (with largest addresses), although in real-world kernel-space C code storing two or more structures of arbitrary size in one memory block is quite a common practice, which cannot be expressed in the intermediate language with the semantics defined this way. Fortunately, the local pointer encoding described in the previous section (that is used for the resulting axiomatic specification of the program eventually generated by JESSIE) lets us weaken the precondition of the memory ripping operation (rip) by eliminating the constraint \( o_1 < o_2 \) (introducing another inference rule for \( o_2 < o_1 \)) and thus allowing for ripping either the rightmost or the leftmost part of a memory block once at a time. After such transformation, the intermediate language semantics becomes powerful enough to express the majority of real-world lower-level memory operations such as encoding/decoding operations and byte re-orderings frequently encountered, for instance, in file system and network device drivers.

IV. CONCLUSION AND FUTURE WORK

In the paper we presented the simplified JESSIE intermediate language with simultaneous support for hierarchical pointer casts and discriminated unions. Its semantics is compatible with the semantics of the C programming language, particularly with regard to modeling array accesses in presence of hierarchical pointer casts. The language presented served both as a concise summary of the concepts underlying the original JESSIE intermediate language [2] and as a starting point for its further extension. We presented the extensions of the language that allow to express low-level pointer casts for some pointers to structures or unions without pointer fields. From practical perspective these contributions together allow significantly broader class of real-world C kernel-space code samples to be translated into the intermediate language in order to be analyzed and verified. Here a significant work for cleaning up the current (modified) JESSIE plugin implementation and deductive verification of some considerable kernel-space C codebase with it remains to be finished before we can present some meaningful results. Currently, the implementation of support for the presented \texttt{rmem} operation in mostly finished, so that we can already prove fragments involving byte reorderings, but not yet encoding/decoding operations (in general). From theoretical point of view, meanwhile, there remains many important directions for future work. On the one part, a rigorous formalization of the correspondence between the intermediate language semantics and that of the C programming language (at least in some limited form) is still to be done. On the other part, the formalization of the correspondence between the intermediate language and its axiomatic representation eventually generated by the tool implementation is also a subject for further research.

REFERENCES


