ISSN 0361-7688, Programming and Computer Software, 2015, Vol. 41, No. 4, pp. 197–208. © Pleiades Publishing, Ltd., 2015.
Original Russian Text © M.U. Mandrykin, A.V. Khoroshilov, 2015, published in Programmirovanie, 2015, Vol. 41, No. 4.

High-Level Memory Model with Low-Level Pointer Cast Support for Jessie Intermediate Language[1]

M. U. Mandrykin* and A. V. Khoroshilov**

The Institute for System Programming of the RAS, ul. A. Solzhenitsyna 25, Moscow, 109004 Russia
*e-mail: *mandrykin@ispras.ru, **khoroshilov@ispras.ru*
@

Abstract—The paper presents a target analyzable language used 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 F@ramaC static analysis framework. Compared to the original Jessie, the extension is fully compatible with the C semantics of arrays, initially supports discriminated unions and prefix (hierarchical) structure pointer casts and provides a limited, but reasonable support for low-level pointer casts (reinterpretations of the underlying bytes of memory). The paper describes the approaches to translation of the original C code into the analyzable intermediate language and of the intermediate language into Why3ML i.e. the input language of the Why3 deductive verification platform.

DOI: 10.1134/S0361768815040040

INTRODUCTION

There are many tools and techniques for static verification of C source code. Among them are deduction techniques, that are based on translation 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 correctness 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 (discharged) by various tools that can be either completely automatic, e.g. SAT- and SMT solvers [1, 2] (Z3, CVC3, CVC4, AltErgo et al.) and saturation-based provers (Vampire, EProver et al.), or interactive, e.g. Coq and PVS proof assistants.

Thus any deductive verification tool for C that aims the most completely automatic verification should somehow translate the semantics of the original code and provided specifications into a set of VCs that the existing state-of-the-art automatic theorem provers are capable to discharge.

Meanwhile the basic features of the C programming language significantly complicate such translation. First, C is an imperative programming language and thus the implicit program state should be somehow modeled in the resulting logical formulas. It can be modeled in several ways usually involving theory of arrays for representing the state of program memory. Second, C assumes manual memory management, which requires generation of additional memory safety checks, corresponding to possible memory mismanagements (premature deallocation, garbage, access violations and buffer overruns). Third, C is a low-level language with generally untyped semantics, which can substantially influence the choice of an appropriate memory model.

Let's assume the VC for the assertion check at line 10 ($a[n] = 1$) is required. The most straightforward and also precise solution would be to make use of the low-level memory model for constructing the corresponding logical formula. In this model each variable in the program memory is represented with a range of elements in the common logical byte array M indexed by 32-bit (or 64-bit) integer addresses. A logical array [2] (as a logical type supported by SMT-solvers) is a total map from the underlying set of the array index type to the underlying set of its element type that is associated with the corresponding pure (i.e. side-effect free) ternary update operation $\cdot[\cdot \leftarrow \cdot]$ defined on triples of the form (array, index, value). The result of the operation is a new logical array of the same (index and element) type, that differs from the original one on and only on the specified index mapped to the new specified element value (i.e. for any other distinct index the value from the original array is retained). Logical formulas can only constrain the values of logical arrays on certain indexes, while initially the values are treated as non-deterministic. The value of an array M on index i is further denoted as $M[i]$. For the array variable M representing the program memory we introduce a finite sequence of logical array constants M_1, M_2, ..., M_n corresponding to the states of program memory resulting from several consecutive memory updates. Assuming the variables a, b, c, d, e, n and m have 32-bit addresses represented with constants a, b, c, d, e, n

[1] The article was translated by the authors.

and m respectively, the required VC can be encoded within the low-level memory model as follows:

$$
\begin{aligned}
&M_1 = M_0[a \leftarrow M_0[e]] \wedge \\
&M_2 = M_1[b \leftarrow M_1[a]] \wedge \\
&M_3 = M_2[M_2[b] +_{32} M_2[n] \leftarrow 0_8] \wedge \\
&M_4 = M_3[M_3[b] +_{32} M_3[n] +_{32} 1_{32} \leftarrow 0_8] \wedge \\
&M_5 = M_4[M_4[b] +_{32} M_4[n] +_{32} 2_{32} \leftarrow 0_8] \wedge \\
&M_6 = M_5[M_5[b] +_{32} M_5[n] +_{32} 3_{32} \leftarrow 1_8] \wedge \\
&M_7 = M_2[M_6[d] +_{32} M_6[m] \leftarrow 0_8] \wedge \ldots \wedge \\
&M_{10} = M_5[M_9[d] +_{32} M_9[m] + 3_{32} \leftarrow 2_8] \wedge \\
&M_{11} = M10[M_{10}[c] \leftarrow 97] \wedge
\end{aligned}
$$

$$
\left.
\begin{aligned}
&(b \le a - 4 \vee b \ge a + 4) \wedge \ldots \wedge \\
&(e \le a - 16 \vee e \ge a + 4) \wedge \ldots \wedge \\
&(d \le d - 16 \vee e \ge d + 16)
\end{aligned}
\right\} 42 \text{ @@@@@@@.}
$$

The low-level memory model allows very precise bit-accurate modeling of C program semantics potentially (with even more precise modeling of CPU registers and caches) achieving nearly full conformance between the program model and the actual execution of the program on a real physical machine. The logical formulas involving both logical arrays and bit-vector reasoning are often amenable to automatic satisfiability checking with the state-of-the-art SMT solvers, though the existing decision procedures for the theory of logical arrays are generally incomplete [2] (i.e. the corresponding tools can loop forever or terminate without the satisfiability verdict). But even the simplest example above demonstrates the redundancy and nonoptimality of the low-level memory model. The model is poorly scalable in practice and is not usually used by deductive verification tools directly, but rather after applying a number of substantial optimizations.

Existing Jessie Memory Model

It worth to be mentioned here that in this paper we only consider logical formulas in first-order classical logic with equality modulo several theories (such as the theory of arrays, uninterpreted functions, bit vectors, and linear integer/real arithmetic), i.e. so-called *SMT* formulas. Alternative logics, including the ones based on separation logic [3], are usually rather insufficiently supported by automatic satisfiability checkers and therefore usually require significantly greater user intervention to determine the satisfiability of the resulting VCs. In the field of deductive verification the techniques based on SMT formulas are still the most often used in practice, particularly in the tools ESC/JAVA [4], SPEC# [5], VCC [6], FRAMAC/JESSIE [7, 8], FRAMAC/WP [7] and others.

The first, most prominent and obvious optimization used in SMT formula construction is called *pure variables* optimization, which consists in selective treatment of program variables that are only addressed by their names (while in general variables can also be addressed by pointers). In particular, the variables that never occur as arguments to the addressing operation (&), are often regarded as pure. This assumption is sound if the program under consideration is memory-safe (free of access violations and buffer overruns). As pure variables can only be addressed by their names, the corresponding access operations on these variables can be easily statically separated from each other and from accesses to other variables in program memory. Therefore each such variable can be represented with a separate finite sequence $v_1, v_2, \ldots, v_k$ of constants rather than an index in the common array sequence $M_1, \ldots, M_n$. This optimization alone allows to decrease the number of inequalities (stating the disjointness/separation of variables in program memory) in the above example from 42 to 6, also reducing the number of logical array constants (common memory states) $M_1, \ldots, M_n$ from 11 to 9.

The idea of separation for the common memory array M applied to the pure variables can be generalized to arbitrary a priori separated (up to some static approximation) memory regions. For that purpose we can separate the program memory into a number of disjoint sets of program variables, such that for any particular pointer expression in the program source code the unique corresponding set of program variables possibly addressed by that pointer can be determined once for all the states of the program execution corresponding to the source code location of the pointer. Thus we obtain a static approximation of program memory separation. The resulting disjoint sets of variables are called *regions* in direct analogy to the regions introduced in [9, 10] as a tool for automatic dynamic memory management without garbage collector (region-based memory management). These papers don't suggest any concrete algorithm for memory region inference (although they provide corresponding inexplicit inference rules), but the paper [11] suggests a practical algorithmic approach, allowing to compute context-sensitive (but flow-insensitive) approximation of region-based program memory separation statically. This technique is based on fix-point computation starting from the initial approximation assigning each pointer variable in the program with its own region and proceeding with unification of the regions according to the set of several simple unification rules. The result of the algorithm is a region label assignment, matching each pointer expression in the program with exactly one region label. To apply the suggested technique to arbitrary C programs the authors suggest using preliminary source code *normalization* that basically includes elimination of address operations (&) through retyping the addressed variables to pointers and reduction of the three C indirect access operations (namely, *, [] and .) to the single operation -> with the use of retyping, pointer arithmetic and dummy single-field structures for simple data types. After normalizing the sample annotated C

```
1   int *a, *b, d[4], e[4];
2   char c[4];
3   int n, m;
4   // ...
5   a = e;
6   b = a;
7   b[n] = 1;
8   d[m] = 2;
9   c[1] = 'a';
10  //@ assert a[n] ≡ 1;
```

Fig. 1. Sample annotated C code fragment.

```
struct intP { int intM; };
struct charP { char charM; };
// ...
struct intP *a, *b, d[4], e[4];
struct charP c[4];
int n, m;
// ...
a = e;
b = a;
(b + n)->intM = 1;
(d + m)->intM = 2;
(c + 1)->charM = 'a';
//@ assert (a + n)->intM ≡ 1;
```

Fig. 2. Normalized annotated C program fragment.

program fragment in Fig. 1 we obtain the normalized C program fragment shown in Fig. 2.

The separation of memory regions in the normalized program allows to significantly simplify (reduce) the earlier obtained VCs. In the given example, the whole program memory can be separated into three regions, namely: one common region assigned to pointers a, b and e, one region for pointer d, and a region for pointer ñ. By supplying each region with a separated logical array and taking into account the pure variables optimization considered above, we obtain the following VC:

$a_1 = e_0 \wedge$
$b_1 = a_1 \wedge$
$M_{a,b,e,1} = M_{a,b,e,0}[b_1 +_{32} n_0 \leftarrow 0_8]$ ^ . . . $\wedge$
$M_{a,b,e,4} = M_{a,b,e,3}[b_1 +_{32} n_0 +_{32} 3_{32} \leftarrow 1_8] \wedge$
$M_{d,1} = M_{d,0}[d_0 +_{32} m_0 \leftarrow 08] \wedge \ldots \wedge$
$M_{d,4} = M_{d,3}[d_0 +_{32} m_0 +_{32} 3_{32} \leftarrow 2_8] \wedge$
$M_{c,1} = M_{c,0}[c_0 \leftarrow 97]$

This VC makes use of only 4 logical array sequences for tracking the state of the corresponding distinct memory regions, and the inequalities stating the disjointness of memory addresses for different program variables are absent from the VC altogether since they rendered superfluous as a result of region-based memory separation.

Yet application of simple region-based memory separation alone faces scalability issues on sufficiently big input programs: In bigger C programs the memory regions are often unified, which is especially noticeable for function-local regions due to a large number of function calls with different combinations of pointer arguments. One solution for this scalability issue is suggested in paper [12] introducing the notion of *polymorphic region*, i.e. a region whose scope is restricted to a single strongly connected component in the function call graph of a program (i.e. a single function or several mutually recursive ones) and that is treated as an extra function parameter substituted with different region arguments at each call site. Polymorphic region separation is performed independently for each strongly connected component, which significantly facilitates the scalability of the overall approach. As polymorphic regions are regarded as extra function parameters of the corresponding functions, the callee functions accept extra arguments, the other polymorphic regions of the caller functions. This, however, leads to additional requirements expressed in the form of additional function preconditions, because a caller function can supply the same region as an argument for two distinct parameter regions at the same call site violating the separation assumptions made during the separation analysis of the callee function's strongly connected component. In the above example, assuming the variables d, e and c to be parameters of a function, those additional preconditions can be formulated as follows:

$(e_0 \le d_0 - 16 \vee e_0 \ge d_0 + 16) \wedge$
$(e_0 \le c_0 - 16 \vee a_0 \ge c_0 + 4) \wedge$
$(d_0 \le c_0 - 16 \vee d_0 \ge c_0 + 4)$

The obtained optimized VCs can be simplified even further if certain restrictions on the source code of the analyzed programs are introduced. If it's guaranteed that in the source program aliasing between pointer variables of different types never occurs, i.e. two pointers of different types can never address the same variable, than the separation between regions corresponding to pointers of different types can be assumed globally for the whole program thus further reducing some function preconditions. In such assumptions it seems natural to abandon the low-level byte-wise representation of memory regions' logical arrays elements and opt for a more efficient higher-level representation employing logical (unbounded) integral types and modular arithmetic:

$a_1 = e_0 \wedge$
$b_1 = a_1 \wedge$
$M_{a,b,e,1} = M_{a,b,e,0}[(b_1 + n_0) \bmod 4294967296 \leftarrow 1] \wedge$
$M_{d,1} = M_{d,0}[(d_0 + m_0) \bmod 4294967296 \leftarrow 2] \wedge$
$M_{c,1} = M_{c,0}[c_0 \bmod 4294967296 \leftarrow 97] \wedge$
$e_0 \le d_0 - 4 \vee e_0 \ge d_0 + 4$ (@@@@@@@).

As a result we end up with a high-level memory model applicable to verification of programs involving not only the simple-typed variables, but also arrays

and structures including support for prefix (hierarchical) type casts[2] [15] and discriminated unions[3]. The corresponding approaches are described in [13, 14] and [15].

All the optimizations considered above are implemented in the Jessie [16, 8] deductive verification tool. As in practice the vast majority of C programs exploit only a form of memory aliasing limited in a way that all pointers are aligned (there is no pointers of any type addressing any word inside a variable of that type other than the first one) and the pointed variables are disjoint (they can be nested, but otherwise they don't intersect), and also, as claimed by some researchers [17], prefix typecasts (including casts to/from the void *) along with discriminated unions together constitute up to 99% of all pointer type casts in most production C codebases, the memory model implemented in Jessie should predictably turn out to be appropriate and efficient in practice.

Issues of the Existing Memory Model

Meanwhile some important fragments in operating system kernels source code that serves as one of the most promising fields of application for existing deductive verification techniques remain largely disregarded by the model. Unlike the model used in VCC [18] the Jessie memory model lacks support for operations on bit-fields and for pointer type reinterpretations. But at the same time 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, integers and structures in various combinations.

The original thesis [8] describing the methods underlining the current implementation of the Jessie tool suggests exploiting the region inference (and effect computation) techniques [19, 9, 11, 12] mentioned above 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 (especially, those SMT-solvers based on DPLLL [2, 1]). They usually implement each of the supported logical theories separately and establish the interaction between the theories by propagating disjunctions of predicates among which only equalities are properly interpreted by all the theories involved [2]. 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 particular solver's set of natively supported features or their support is very limited and inefficient. Within such restrictions all the interaction between the involved low-level and high-level memory models is expressed through either quite sophisticated inter-theory predicates or uninterpreted predicates with the appropriate sets of axioms. In this case all the additional tasks of specifying such predicates and possibly manually proving some of the resulting VCs (due to the complexity of the predicates) lay upon the verification engineer. 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 (especially in case of long bit vectors possibly resulting from some memory reinterpretations between structure or union arrays). 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 (or its may-aliases) in all other places are preferably encoded as bit-vectors as well. In addition to that, bit-vectors were still unsupported by the latest version of the Why3 [20] verification platform targeted by the Jessie tool at the time of writing this paper, and the implementation of the bit-vector regions support in Jessie itself were still mostly raw and incomplete.

Another shortcoming of the original Jessie memory model was that its treatment of pointer shift operation turned out to be incompatible with the corresponding C counterpart in presence of prefix structure pointer cast support. Initially the Jessie memory model aimed supporting several input languages (at least Java with JML, C with ACSL and OCaml with some analogous specification language). So apparently in order to simplify handling of Java (and OCaml) arrays, the semantics of Jessie arrays was chosen to be correspondent to that of Java arrays (or OCaml arrays of boxed values). As a result, while in C shifting a pointer into an array of derived structures (with longer list of fields) becomes wrong after casting the pointer to the type of a parent structure (with shorter list of fields), the original Jessie intermediate language semantics lacked this restriction (in compliance with Java arrays covariance) and thus significantly limited the amount of C programs correctly translatable into the intermediate lan-

[2] *Up-* and *downcasts* (also *prefix* or *hierarchical casts*) are casts between two structure types, one of the structures (*base* structure) having the list of fields that forms a prefix of the list of fields comprising the other (*derived*) structure.

[3] *Discriminated unions* are unions whose field addresses are not taken (directly or indirectly through the corresponding pointer casts) and in which only the last field written should be subsequently read.

guage (without a significant amount of additional explicit checks).

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 thesis [8] and paper [22]. 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 (SIL)* we present in the paper still allows us to emphasize the basic aspects of the full language and show how it was extended with partial support for low-level memory access (pointer type reinterpretations), covering its most important use cases such as encoding/decoding operations and conversions between different byte orderings.

The paper first introduces this simplified intermediate language (SIL), a small toy analogue to the real Jessie, along with the corresponding high-level memory model. Then the *extended simplified intermediate language (ESIL)* is introduced, which extends the SIL (and its memory model) with intent to provide enough support for some low-level pointer casts, while maintaining all previous advantages of the model. The suggested approach is extensible to structures with bitfields and interacts well with improvements made in the existing implementation of the Jessie plugin, i.e. bounded pointers, multiple inheritance hierarchies, and region inference. So the extension part of the presented language and its memory model corresponds to our contribution with regard to the full Jessie intermediate language and its implementation. The current results of this implementation are mentioned in the conclusion.

SIMPLIFIED INTERMEDIATE LANGUAGE

Abstract Syntax

The Jessie translator is the tool that stands in the middle of the verification toolchain working sequentially in the following order: FRAMAC 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 of the original program merged together along with the specifications 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 (or *normalizations*) primarily concerning simplification of the program's memory layout (e.g. eliminating nested structures and address taking operations). So in the paper we aren't introducing the Jessie intermediate language itself, but rather using its dramatically simplified counterpart capturing the basic capabilities of the full language that refer to its memory model. The most important simplifications distinguishing SIL in relation to Jessie are the absence of typing, region inference and reduced discriminated union support. In practice, the existing region inference techniques implemented in Jessie are directly applicable to the SIL as well. SIL only supports discriminated unions whose fields have simple (i.e. non-composite) types. This limitation can be relatively easily overcome by refining (and complicating) the semantics of the corresponding *strong update* operations, i.e. the assignments to the union fields, by adding the necessary subsidiary non-deterministic assignments. Another option, suggested by this presentation, is using the explicitly specified memory reinterpretation operations presented in the following sections. The abstract syntax of this *simplified intermediate language* (SIL) is presented in Fig. 3.

In this figure the following notation is assumed:

—v stands for an integer variable;

—$n \in \mathbb{Z}$ 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 naturally acquires precise semantics when the language is analyzed using a deductive reasoning technique such as weakest precondition calculus [21];

—$\star$ stands for any binary operation on integer values;

—p stands for a pointer variable;

—$f \in \mathbb{F}$ is a unique structure field or union field label from a finite set $\mathbb{F}$ of such labels (the filed labels are unique throughout the whole program rather than within the containing structure or union only);

—$t \in \mathbb{T}$ is a unique tag label[4] of a structure, a union or a field of a union from a finite set of tags $\mathbb{T}$. The finite sets of labels and tags share no common elements (so the union fields are provided with two distinct labels each, one label from $\mathbb{F}$ and one label from $\mathbb{T}$;

—$\bowtie$ stands for any binary relation on integer values;

—$\diamond$ stands for any binary logical connective;

—The predicates $\mathbf{omin}(t_p) \bowtie t_n$, $\mathbf{omax}(t_p) \bowtie t_n$, $\mathbf{tag}(t_p) = t$, $\mathbf{tag}(t_p) \preceq t$ and operators **assume** $\bar{p}$ and **assert** $\bar{p}$ (they are marked with a star) only apply for the analyzable part of the language, so they have no

[4] In case of explicit (non-anonymous) and unique structure and union tags, the tag labels can be identified with the tags themselves.

integer term:

$$
\begin{array}{rll}
t_n ::= & v & \text{int. variable} \\
| & n & \text{int. value} \\
| & * & \text{non-det. int. value} \\
| & t_n \star t_n & \text{bin. int. operation} \\
| & \mathbf{acc}(t_p, f) & \textit{dereference} \\
| & \mathbf{psub}(t_p, t_p) & \textit{pointer difference}
\end{array}
$$

pointer term:

$$
\begin{array}{rll}
t_p ::= & p & \text{pointer variable} \\
| & \mathbf{null} & \textit{null pointer} \\
| & \mathbf{alloc}(t, t_n) & \textit{memory allocation} \\
| & \mathbf{acc}(t_p, f) & \textit{memory access} \\
| & \mathbf{shift}(t_p, t, t_n) & \textit{pointer shift}
\end{array}
$$

term:

$$
\begin{array}{rll}
t_{np} ::= & t_n \ | \ t_p & \text{pointer or int. term}
\end{array}
$$

predicate:

$$
\begin{array}{rll}
\bar{p} ::= & t_n \bowtie t_n & \text{int. bin. relation} \\
| & \mathbf{peq}(t_p, t_p) & \textit{pointer equality} \\
{}^*| & \mathbf{omin}(t_p) \bowtie t_n & \textit{min. offset} \\
{}^*| & \mathbf{omax}(t_p) \bowtie t_n & \textit{max. offset} \\
{}^*| & \mathbf{tag}(t_p) = t & \textit{tag equals} \\
{}^*| & \mathbf{tag}(t_p) \preceq t & \textit{tag is upper bounded by} \\
| & \bar{p} \Diamond \bar{p} & \textit{logical connective}
\end{array}
$$

operators:

$$
\begin{array}{rll}
o := & v \leftarrow t_n & \text{var. assignment} \\
| & p \leftarrow t_p & \text{pointer assignment} \\
| & \mathbf{upd}(t_p, f, t_{np}) & \text{field update} \\
| & \mathbf{free}(t_p) & \textit{deallocation} \\
{}^*| & \mathbf{assume}\ \bar{p} & \text{assumption} \\
{}^*| & \mathbf{assert}\ \bar{p} & \text{assertion}
\end{array}
$$

operator sequence:

$$
\begin{array}{rll}
s ::= & o & \text{operator} \\
| & o; s & \text{sequence}
\end{array}
$$

Fig. 3. Abstract syntax of the simplified intermediate language (SIL).

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 Fig. 3 with the corresponding short explanations on the right (given in italics).

We assume the set $\mathbb{T}$ to be partially ordered with respect to the partial order relation $\preceq \subseteq \mathbb{T} \times \mathbb{T}$. So the partially ordered set $\mathbb{T}$ constitutes a bounded join-semilattice with top element $\top$. The relation $\preceq$ corresponds to inheritance (prefix) relation between structures which extends also to unions and union fields so that (1) if the list of fields of any structure with tag t_1 forms a prefix of the list of fields of any structure with tag t_2 we have $t_2 \preceq t_1$, and (2) for the fields of unions we have $t_f \preceq t_u$, where t_f is the label of a field of the union with tag label t_u.

SIL doesn't support array, structure or union embedding (nesting). For that reason along with the source C program normalization mentioned above (and involving elimination of addressing operations (&) and reducing all C memory access operations to the only operation -> Jessie plugin also applies transformations eliminating embedded structures, unions and arrays. Generally, embedding is eliminated by transforming embedded structures, arrays and unions into their indirectly accessed, explicitly allocated and deallocated counterparts. Thus every embedded variable is automatically moved into dynamic memory (heap). Also, as SIL (so as Jessie) supports composite types (i.e. arrays, structures and unions) only through heap pointers, all automatic (stack) composite variable allocations are transformed into heap allocations and deallocations as well. The embedding of a base structure into a corresponding derived structure as the first field is treated as a special case. In this case the list of fields of the embedded (inner, base) structure is prepended to the list of fields of the outer (derived) structure, and the inheritance (prefix) relation $\preceq$ is established between the structures so that the corresponding hierarchical type casts become possible. In SIL, due to its dynamically-typed semantics, the prefix type casts can be omitted, but they are necessary in Jessie intermediate language as in C.

The meaning of **omin**, **omax** and **tag** predicates is explained in the next section presenting the pointer and memory models of the SIL.

Memory Model

According to its original description [8], 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 Fig. 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 $0 \leq o < \mathrm{A}[l]$. Only valid pointers can be dereferenced so that a program can only read from or write into the memory accessible through valid pointers, which corresponds to memory allocated by the OS (or memory management subsystem of the kernel core) through the appropriate interfaces. No distinction, however, 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 one of C).

However, as was mentioned in the introduction, this memory model is not originally compatible with both its support for hierarchical pointer casts and the semantics of pointer shifts in the C programming language. This incompatibility arises from the possibility of dealing with unaligned pointers resulting from nonzero shifts of pointers to derived structure array elements casted to one of the corresponding base structure types.

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 in SIL. 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 in SIL.

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

The absence of misaligned pointers allows high-level memory representation with a separately updatable logical array ("memory") per each structure/union field [15, 22] as long as we refuse to maintain the original C language semantics in presence of non-hierarchical pointer casts.

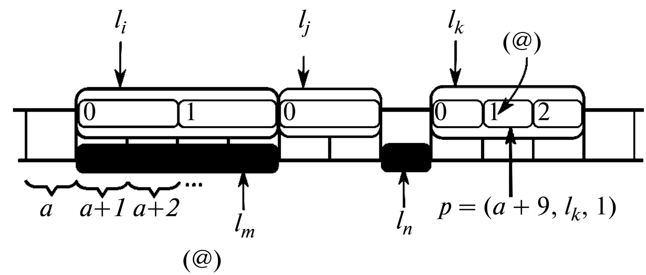

Fig. 4. Byte-level block memory model.

The abstract syntax of SIL 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).

Semantics

SIL (just as 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 Vcs 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 [15, 22].

The dynamic memory (heap) of the SIL program is represented by the map $\mathbf{M} : \mathbb{P} \to \mathbb{M}_n \cup \mathbb{M}_p$, where $\mathbb{M}_n = \{\mathbf{M} : \mathbb{F} \to \mathbb{Z}\}$, and $\mathbb{M}_p = \{\mathbf{M} : \mathbb{F} \to \mathbb{P}\}$, $\mathbb{P} = \{(l, o) | l \in \mathbb{L}, o \in \mathbb{Z}\}$ is the set of pointers, and $L \simeq \mathbb{N} \cup \{0\}$ is a countable set of distinct (unique) memory block labels including a special label for the null pointer.

However, the initial representation of mappings A and T 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.

We consider two most important of these optimizations: local encoding of memory block lengths making the representation of the map A more efficient, and

partially axiomatic encoding of the map T for structure tags.

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 $A : \mathbb{L} \to \mathbb{Z}$ with the map $A' : \mathbb{P} \to \mathbb{Z}$ such that $\forall (l, o) \in \mathbb{P} .A'[(l, o)] = A[l]$. Next we can notice that now all the three mappings A, T and M are defined on the set $\mathbb{P}$. Thus we can hide the internal structure of pointers as pairs behind an abstract type by introducing a function $o : \mathbb{P} \to \mathbb{Z} : \forall (l, o) \in P.o((l, o)) = o$. Then by substituting for any pointer p the function $o(p)$ and the map $A'[p]$ with two functions $\text{omin}(\mathcal{A}, p) = -o(p)$ and $\text{omax}(\mathcal{A}, p) = A'[p] - o[p] - 1$ 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. **shift**, **psub**, **acc**, **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. $\text{omax}(\mathcal{A}, p) \geq 0$ vs. $A[p] - o(p) - 1 \geq 0$). This encoding first appeared in the C@aduceus deductive verification tool [22, 23] and is also used in Jessie, where the functions omin and omax are encoded as uninterpreted functions of two arguments with the appropriate set of axioms.

Second, we notice that as we only allow discriminated unions as a special case of moderated unions[5], pointers to structures are never aliased with pointers to unions. But for pointers to structures the following invariant is always maintained: $\forall p \in \mathbb{P}.\forall i \in \mathbb{Z}.T[p] = T[\text{shift}(p, T[p], i)]$. This suggests replacing the map T with a function $\text{tag}(\mathcal{T}, p) = T[p]$ for pointers to structures and a map T_u similar to the original T for unions. Then the function $\text{tag}(\mathcal{T}, p)$ is encoded as an uninterpreted function of two arguments with several necessary axioms (including the mentioned invariant). This significantly reduces function preconditions since instead of requiring appropriate 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.

EXTENDED SIMPLIFIED INTERMEDIATE LANGUAGE

Motivation and Abstract Syntax

As was mentioned in the introduction, many real-world C programs involving low-level reinterpretations of some memory regions, cannot be directly expressed both in SIL described in previous sections and in the original Jessie. Typical examples of such reinterpretations are casts of structures to byte arrays preceding their transferring through network interfaces and byte reorderings in integer variables performed after receiving them from files or USB interfaces.

[5] A *moderated union* in C is a union whose field addresses are not taken, directly or indirectly through pointer type casts, or, in other words, whose filed accesses are always mediated by (i.e. done by involving) the union itself.

So in order to add the support for such code fragments to the deductive verification tool we suggest extending SIL with two new capabilities—memory *reinterpretation* and memory block *ripping*.

The intuitive meaning of the former capability is transforming 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 (*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 (*joining* inside a memory block).

The reinterpretation operation alone, however, is still insufficient to represent the reinterpretation of a byte array filled with two consecutive structures of mutually non-divisible sizes into the two corresponding structure pointers (due to the indivisibility of the byte array size by the size of either structure).

Here we can apply the latter capability, whose intuitive meaning is splitting the original memory block just before the reinterpretation into two continuous parts—an accessible and a ghost one and then joining these parts back together after the necessary memory 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 *ripping*" rather than "memory block splitting"' because the latter term is rather associated with the corresponding kind of memory 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.

The extension of the abstract syntax of SIL is shown in Fig. 5. The extension introduces a new function *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 can not be enforced in ESIL due to its untyped semantics, but the explicit cast is mandatory both in Jessie intermediate language and in C. In ESIL the use of the function is needed for the resulting program not to get stuck at

some of the subsequent pointer access operations (*acc, psub, shift, upd* or *free*). Here the typing rules of C considerably help by disallowing the corresponding implicit low-level pointer casts in the original code. In absence of the necessary typecasts or reinterpretation operations (*rmem*), the program in ESIL gets stuck on one of the further operations which is manifested by invalidity of the VC corresponding to the operation precondition.

The function *rip* and the operations *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 the function. This pointer then can be used in the following *rmem* 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 to 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.

So we extended the SIL presented in Fig. 3 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 (reinterpretation between types of equal sizes can be regarded in both ways), while the function *rip* and the operation *mend* correspond to memory ripping and mending respectively.

Extended Memory Model

To clarify and formalize the semantics of the functions *cast* and *rip* and the operations *rmem* and *mend* we consider the appropriately extended memory model. The memory model of SIL presented above is extended with model functions and predicates allowing to express the corresponding constraints.

$$
\begin{array}{rll}
t_p ::= & \dots & \\
| & \mathbf{cast}(t_p, t, t) & \textit{reinterpret cast} \\
^*| & \mathbf{rip}(t_p, t_p) & \textit{ripping} \\
 & & \\
o := & \dots & \\
^*| & \mathbf{rmem}(t_p, t, t) & \textit{reinterpretation} \\
^*| & \mathbf{mend}(t_p, t_p) & \textit{mending}
\end{array}
$$

Fig. 5. Extension of the SIL abstract syntax.

To support memory reinterpretation we introduce a new model function φ: $\mathbb{T} \times \mathbb{T} \to \mathbb{Z}$. The function is defined on ordered pairs of structure, union and union field tags in the following way:

—if the tags t_1 and t_2 satisfy the following requirements: (1) the size s_{t_1} of the first composite type, union, or union field (including alignment padding) with tag t_1 is a multiple of the size of the second one (s_{t_2} tagged with t_2), (2) both t_1 and t_2 correspond to either union fields or structures/unions without pointer fields (this requirement is not satisfied, in particular, by the label $\top$), and (3) there is a logical predicate $rmem_{t_1, t_2}$(**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 or a union field with tag t_1 and the fields of a structure or a union field with tag t_2 (if either t_1 or t_2 is a union tag, the corresponding predicate $rmem_{t_1, t_2}$ can be safely turned to identical truth as none of the union fields can be accessed or updated before (correspondingly after) such reinterpretation), then $\varphi(t_1, t_2) = s_{t_1}/s_{t_2}$;

—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_{t_1}) and the two remaining conditions are met, then $\varphi(t_1, t_2) = -(s_{t_2}/s_{t_1})$;

—$\varphi(t_1, t_2) = 0$, otherwise.

This predicate $rmem_{t_1, t_2}$ 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.

The next model function π: $\mathbb{L} \times \mathbb{T} \to \mathbb{L}$, is defined on pairs of block labels and destination tags as illustrated by Fig. 6.

Here we use the isomorphism between the sets $\mathbb{L}$ and $\mathbb{N} \cup \{0\}$. We supplement unique block labels with tags from $\mathbb{T}$. For allocation (**alloc**) operations that assign unique labels to their corresponding memory blocks these label tags must correspond to the tags

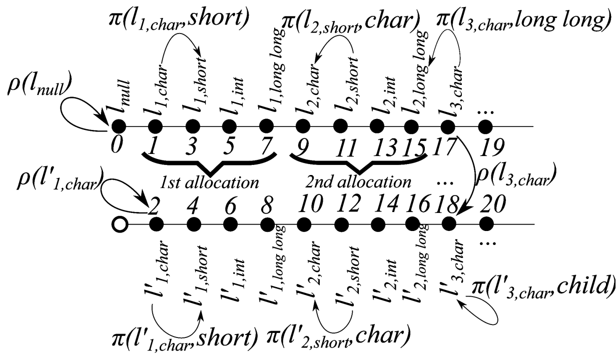

Fig. 6. Unique memory block labels in the extended byte-level block memory model.

explicitly specified as arguments of the corresponding operations. So a unique label l_{it} (or $l_{i,\,t}$) 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 $l_{i,\,t}$ there is a right-adjacent ghost label $l'_{i,\,t}$ and a collection of reinterpretation labels $\{l_{i,\,t} | t \in \mathbb{T}\}$, where for any two labels $l_{i,\,t} \neq l_{j,\,t'}$ holds given $i \neq j \wedge t \neq t'$. The function π maps a label $l_{i,\,t}$ to its corresponding reinterpretation label $l_{i,\,t'}$ so that $\pi(l_{i,\,t}, t') = l_{i,\,t'}$. Generally, there is no difference between a memory bock obtained initially by actual allocation and a block reinterpreted from another one with a different tag by using the **rmem** operation.

The last function we introduce, ρ: $\mathbb{L} \to \mathbb{L}$ maps a label $l_{i,\,t}$ to its corresponding right-adjacent ghost label $l'_{i,\,t}$ and is intended to be used in the **rip** function and the **mend** operation.

With the three new functions φ, π and ρ it is possible to formalize the semantics of the new functions **cast** and **rip** and the operations **rmem** and **mend**.

The semantics should preserve the invariant that exactly the memory addressed by valid pointers in the original program can be accessed in the corresponding intermediate language program. Besides the evaluation semantics, the operation **rmem** also has additional implicit analyzable semantics. The operation has a postcondition $rmem_{t_1,\,t_2}(\mathbf{M}, \mathrm{T}, l, l', \mathrm{A}[l])$ that is treated as an **assume** operator inserted just after the **rmem** operation.

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 three or more structures of arbitrary size in one memory block is quite a common practice, which cannot be expressed in ESIL 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 express both cases of ripping, i.e. either the rightmost or the leftmost part of a memory block once at a time. With local pointer encoding, the ESIL semantics enables one 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 systems and network device drivers.

CONCLUSION AND FUTURE WORK

In the paper we presented the simplified intermediate language (SIL), an analogue of the Jessie intermediate language, supplied with simultaneous support for hierarchical pointer casts and discriminated unions. The semantics of SIL is compatible with the semantics of the C programming language, in particular with regard to modeling of 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 [8] 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 **rmem** operation in mostly finished, so that we can already prove fragments involving byte reorderings, but not yet some cases of encoding/decoding operations and buffering.

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.

ACKNOWLEDGMENTS

The research was supported by the Ministry of Education and Science of the Russian Federation (unique project identifier RFMEFI60414X0051).

REFERENCES

1. Gomes Carla, P., Kautz, H., Sabharwal Ashish, and Selman, B., *Satisfiability Solvers*, 2008.
2. Kroening D. and Strichman O., *Decision Procedures: An Algorithmic Point of View*, 1st edition, Springer Publishing Company, Incorporated, 2008.
3. Reynolds, J.C., Separation logic: A logic for shared mutable data structures, *Proc. 17th Annual IEEE Symposium on Logic in Computer Science. LICS '02*, Washington, DC, USA: IEEE Computer Society, 2002, pp. 55–74. URL: http://dl.acm.org/citation.cfm id= 645683.664578http://dl.acm.org/citation.cfm id= 645683.664578http://dl.acm.org/citation.cfm id= 645683.664578.
4. Cormac Flanagan, Rustan, K., Leino, M., Lillibridge, M., et al., Extended static checking for Java, *Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02. New York, NY, USA: ACM, 2002, pp. 234–245. URL: http://doi.acm.org/10.1145/512529.512558http://doi.acm.org/10.1145/512529.512558http://doi.acm.org/10.1145/512529.512558.
5. Barnett, M., Leino, K., Rustan, M., and Schulte Wolfram, The spec# programming system: An overview, *Proc. 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices,* CASSIS'04. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 49–69. URL: http://dx.doi.org/10.1007/978-3-540-30569-9_3http://dx.doi.org/10.1007/978-3-540-30569-9_3http://dx.doi.org/10.1007/978-3-540-30569-9_3.
6. Cohen, E., Dahlweid, M., Hillebrand, M., et al., Vcc: A practical system for verifying concurrent C, *Proc. 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 23–42. URL: http://dx.doi.org/10.1007/978-3-642-03359-9_2http://dx.doi.org/10.1007/978-3-642-03359-9_2http://dx.doi.org/10.1007/978-3-642-03359-9_2.
7. Cuoq, P., Kirchner, F., Kosmatov, N., et al., Frama-C: A software analysis perspective, *Proc. 10th International Conference on Software Engineering and Formal Methods*, SEFM'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 233–247. URL: http://dx.doi.org/10.1007/978-3-642-33826-7_16http://dx.doi.org/10.1007/978-3-642-33826-7_16http://dx.doi.org/10.1007/978-3-642-33826-7_16.
8. Moy Yannick, Automatic modular static safety checking for C programs, *PhD Thesis*, Université Paris-Sud., 2009. URL: http://www.lri.fr/~marche/moy09phd.pdfhttp://www.lri.fr/~marche/moy09phd.pdfhttp://www.lri.fr/~marche/moy09phd.pdf.
9. Talpin, J.-P. and Jouvelot, P., Polymorphic type, region and effect inference, *Journal of Functional Programming*, 1992, vol. 2, pp. 245–271.
10. Tofte, M. and Talpin, J.-P., Region-based memory management, *Information and Computation*, 1997, vol. 132, no. 2, pp. 109–176. URL: http://www.sciencedirect.com/science/article/pii/S0890540196926139http://www.sciencedirect.com/science/article/pii/S0890540196926139http://www.sciencedirect.com/science/article/pii/S0890540196926139.
11. Hubert Thierry and Marchée Claude, Separation analysis for deductive verification, *Heap Analysis and Verification (HAV'07)*, Braga, Portugal: 2007, pp. 81–93. URL: http://www.lri.fr/~marche/hubert07hav.pdfhttp://www.lri.fr/~marche/hubert07hav.pdfhttp://www.lri.fr/~marche/hubert07hav.pdf.
12. Moy Yannick and Marché Claude, Modular inference of subprogram contracts for safety checking, *Journal of Symbolic Computation*, 2010, vol. 45, pp. 1184–1211.
13. Burstall Rodney, M., Some techniques for proving correctness of programs which alter data structures, *Machine Intelligence*, 1972, vol. 7, nos. 23–50, p. 3.
14. Bornat, R., Proving pointer programs in hoare logic, *Proc. 5th International Conference on Mathematics of Program Construction*, MPC '00. London, UK, UK: Springer-Verlag, 2000, pp. 102–126. URL: http://dl.acm.org/citation.cfm id=648085.747307http://dl.acm.org/citation.cfm id=648085.747307http://dl.acm.org/citation.cfm id=648085.747307.
15. Moy Yannick, Union and cast in deductive verification, *Proc. C/C++ Verification Workshop. Technical Report ICIS-R07015,* Radboud University Nijmegen, 2007, pp. 1–16. URL: http://www.lri.fr/~moy/Publis/moy07ccpp.pdf.
16. Marché Claude, Jessie: An intermediate language for Java and C verification, *Proc. 2007 Workshop on Programming Languages Meets Program Verification*, PLPV '07. New York, NY, USA: ACM, 2007, pp. 1–2. URL: http://doi.acm.org/10.1145/1292597.1292598http://doi.acm.org/10.1145/1292597.1292598http://doi.acm.org/10.1145/1292597.1292598.
17. Condit, J., Harren, M., McPeak, S., et al., Ccured in the real world, *Proc. ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03. New York, NY, USA: ACM, 2003, pp. 232–244. URL: http://doi.acm.org/10.1145/781131.781157 http://doi.acm.org/10.1145/781131.781157http://doi.acm.org/10.1145/781131.781157.
18. Cohen, E., Moskal, M., Tobies, S., and Wolfram Schulte, A precise yet efficient memory model for C, *Electron. Notes Theor. Comput. Sci.*, 2009, vol. 254, pp. 85–103. URL: http://dx.doi.org/10.1016/j.entcs.2009.09.061http://dx.doi.org/10.1016/j.entcs.2009.09.061http://dx.doi.org/10.1016/j.entcs.2009.09.061.
19. Steensgaard Bjarne, Points-to analysis in almost linear time, *Proc. 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96. New York, NY, USA: ACM, 1996, pp. 32–41. URL: http://doi.acm.org/10.1145/237721.237727http://doi.acm.org/10.1145/237721.237727http://doi.acm.org/10.1145/237721.237727.
20. Filliâtre, J.-C. and Paskevich, A., Why3: Where programs meet provers, *Proc. 22Nd European Conference on Programming Languages and Systems*, ESOP'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 125–128. URL: http://dx.doi.org/10.1007/978-3-642-37036-6_8http://dx.doi.org/10.1007/978-3-642-37036-6_8http://dx.doi.org/10.1007/978-3-642-37036-6_8.
21. Dijkstra Edsger, W., Guarded commands, nondeterminacy, and formal derivation of programs, *Communications of the ACM*, 1975, vol. 18, no. 8, pp. 453–457.

22. Filliâtre, J.-C. and Marché Claude, The why/krakatoa/caduceus platform for deductive program verification, in *CAV '07*, 2007, pp. 173–177.

23. Filliâtre, J.-C. and Marché Claude, Multi-prover verification of C programs, *Proc. 6th International Conference on Formal Engineering Methods*, Davies, J., Wolfram Schulte, and Barnett, M., Eds., vol. 3308 of Lecture Notes in Computer Science, Seattle, WA, USA: Springer, 2004, pp. 15–29. URL: http://www.lri.fr/~filliatr/ftp/publis/caduceus.ps.gzhttp://www.lri.fr/~filliatr/ftp/publis/caduceus.ps.gzhttp://www.lri.fr/~filliatr/ftp/publis/caduceus.ps.gz .

24. Khoroshilov, A., Mutilin, V., Novikov, E., Shved, P., and Strakh, A., Towards an open framework for C verification tools Benchmarking, *Proc. 8th International Andrei Ershov Memorial Conference*, PSI 2011, Novosibirsk, Russia, June 27–July 1, 2011, Revised Selected Papers, pp. 179–192. URL: http://dx.doi.org/10.1007/978-3-642-29709-0_17.

25. Khoroshilov, A.V., Mutilin, V.S., Novikov, E.M., Shved, P.E., and Strakh. A.V., Linux driver verification architecture, *Proc. Institute for System Programming*, 2011, vol. 20, pp. 63–187. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print).

SPELL: OK