A Value-based Memory Model for Deductive Verification

Quentin Bouillaguet, François Bobot, Mihaela Sighireanu, Boris Yakobowski

A Value-based Memory Model for Deductive Verification*

Quentin Bouillaguet[1,2], François Bobot[1], Mihaela Sighireanu[2], and Boris Yakobowski[1]

[1] CEA, LIST, Software Reliability Laboratory, France
firstname.lastname@cea.fr
[2] IRIF, University Paris Diderot and CNRS, France
firstname.lastname@irif.fr

Abstract

Collaboration of verification methods is crucial to tackle the challenging problem of software verification. This paper formalizes the collaboration between Eva, a static analyzer, and WP, a deductive verification tool, both provided by the Frama-C platform, and concerned with the verification of C programs. The collaboration focuses on verification of programs using pointers, where most deductive verification tools are limited to C programs that do not contain union types, pointer arithmetics, or type casts. We remove some of these limitations from WP by transferring information computed by Eva, which soundly supports these features. We formalize this collaboration by defining a memory model that captures the information on memory inferred by the points-to analysis of Eva, and complies with the abstract memory model used by WP to generate verification conditions. The memory model defined combines a raw memory model with a typed memory model. It captures the low-level operations on pointers allowed by C and provides information about the partition of the memory in disjoint memory regions. This expressivity increases the realm of programs dealt by WP and its efficiency in generation of verification conditions.

1 Introduction

Software verification is a challenging problem for which different solutions have been proposed. Two of these solutions are deductive verification (DV) and static analysis (SA). While deductive verification is interested in checking precise and expressive properties of the input code, static analysis targets checking a fixed class of properties. This loss in expressivity of properties in static analysis is counterbalanced by its high degree of automation. Conversely, deductive verification requires efforts from the user that has to specify the properties to be checked and other annotations, e.g., loop invariants. Using these specifications, DV tools build verification conditions which are formulas in various logic theories and send them to specialized solvers. SA tools may also need to solve constraints, but they generally employ ad-hoc algorithms that soundly decide the satisfiability of constraints.

The complementarity of the two methods have motivated to search new methods that combine deductive verification and static analysis. One of the methods proposed, e.g., [1], consists in first applying static analysis to check some fixed properties of the input code and to infer invariants about the states of the program at each program point. The kind of invariants that may be inferred are interval constraints on the integer or float variables, pointer aliasing, and the shape of the memory. Then, these inferred invariants are injected in the specifications used by the deductive verification tool in order to ease the specification task or to strengthen the existing specifications.

*This work was partially supported by ANR project VECOLIB, grant ANR-14-CE28-0018-03.

Such a collaboration is possible when two conditions are met. The first condition is the availability of a channel that eases the dialogue between such tools. The second condition is the soundness of this dialogue, i.e., the translation of invariants inferred by static analysis into the specification logic used by the deductive verification tool preserves the meaning of properties. The Frama-C platform [19] satisfies the first constraint because it includes both static analysis and deductive verification tools, Eva resp. WP plugins, for the verification of C programs. These tools communicate by associating validity statuses (True, False, Unknown) to the *alarms* present in the program. Alarms are logical assertions, written in the ACSL [5] specification language, that eliminate the runs in which a statement leads to an undefined behavior (e.g., `assert(d != 0);` removes undefined behavior for the statement `r = 1/d;`). Collaboration by alarms is sound due to the common semantics of program expressions, but it is not extensible to more complex properties, e.g., separation of memory regions.

In this paper, we propose a solution that enables a more intimate collaboration of these methods. The idea is to transfer information from SA to DV in the form of a logic formula that encodes the state invariant inferred by the static analysis. This idea has been already applied in the context of test generation in [1], but only for C programs without pointer manipulation. Our solution may infer properties about pointers like aliasing, pointer restricted to a fixed memory region, etc. These properties are valid in a model of the program memory that captures pointer manipulation, e.g., pointer arithmetics, updates at arbitrary locations, type casts. One issue is that this specific memory model is not currently available in WP.

Our work provides this missing link between Eva and WP. We propose a new memory model that captures precisely the one used in Eva and satisfies the constraints of the abstract memory model used by WP to generate verification conditions. A consequence of this connection is that the points-to information computed by Eva is used by WP, like in [28], to know which memory regions are separated. This information is important for WP to generate smaller verification conditions. Indeed, WP may use different memory models for these regions, thus generating smaller constraints, or avoiding their generation altogether.

The memory model defined in this work is actually independent of the static analyzer used. Indeed, we define an interface that contains which information the memory model requires from the abstract values computed by the static analyzer. We implemented this interface for the abstract domain used by Eva, but it could be implemented for other abstract domains that compute abstractions of points-to relations.

To motivate such a collaboration, let us consider the C code from Figure 1. The function `copy` copies the first two elements of the array pointed to by `b` into the one pointed to by `a`. The ACSL specification of the function requires the validity of memory locations that are accessed, while the postcondition expresses that `a` ends up containing the initial contents of `b`. Notice that this specification is incomplete: if the cells `a[0]` and `b[1]` are not separated, the post-conditions is *not* verified. However, in all the calls of `copy` from `main`, this separation hypothesis *is* verified, and Eva is able to infer automatically the separation property for this program. Hence, by injecting this information in WP, we could prove the post-condition.[1]

```
/*@ requires \valid(&a[0..1]) && \valid(&b[0..1]);
    ensures a[0] == \old(b[0]) && a[1] == \old(b[1]);
*/
void copy(int* a, int* b) {
  a[0] = b[0]; a[1] = b[1];
}
extern int t[4], u[5];
void main() {
  copy(&t[0], &u[2]);
}
```

Figure 1: Verification in presence of aliasing

[1] Of course, we are weakening the overall verification, as the specification of the function `copy` is no longer valid in all possible calling contexts. In essence, we assume the *closed world* hypothesis, where `copy` is supposed

Related work Several memory models have been proposed to capture the semantics of programs manipulating pointers. All these models view the memory as a collection of disjoint regions. Two main classes may be distinguished: (i) the regions are typed by the value stored, which is a good abstraction for type-safe languages, (e.g., Java-like [2, 4], HOL [23]) and (ii) the regions are seen as raw arrays of bytes to capture low-level manipulations of memory in C and it is used in static analyzers (e.g., HAVOC [10], VCC [6], MemCAD [9], Infer [8], SMACK [26], Eva [7]) or in deductive verification tools, (e.g., Caduceus [15] or VeriFast [18]). Most of tools based on the second class of memory models allow also typing of memory regions. The abstract memory model of WP supports both classes of models and WP provides concrete implementations for both classes [13].

The use of abstract memory model to capture refinements of these models has been proposed in the CompCert project [22]. The abstract memory model of WP is inspired by the second version of this project [21], but the concrete model we propose has not been considered in CompCert. In [27] is proposed a method to design static analyzers based on an abstract memory model. Eva is not built following these principles for efficiency reasons.

In [17] is used a static analysis based on region inference for the partitioning of a memory model. The analysis employed is less precise than the points-to analysis in Eva because the loss of precision for one location could force many precise locations to be collapsed in the same region. Recent work [28] proposes a precise points-to analysis to infer information about the separation of memory regions in order to decrease the size of verification conditions generated by deductive verification tools. Although Eva is doing a less precise analysis, it is still able to infer such separation properties, and we define in addition a formalized channel to transfer such information to WP. The authors of [6] explore different memory models to generate with VCC a benchmark of problems for SMT solvers. By implementing an additional concrete memory model for WP, we can use it to provide such benchmarks.

Separation Logic [24] is used in many verification tools for C (e.g., GRASSHoper [25], HIP/Sleek [11], Infer, VeriFast) due to the simpler specification of disjointness between memory regions. ACSL includes a separating conjunction operator (understood by WP and Eva), but it is far weaker than the standard separating conjunction operator.

Paper organization Section 2 introduces a simple programming language that exhibits the pointer features we are interested in. On this language, we illustrate the notions of abstract memory model and verification condition generation based on this model. Section 3 provides two memory model employed by WP for the verification of C programs and discusses their advantages and limitations. Section 4 presents the Eva static analyzer and its memory model. The memory model we formalized for Eva in WP is presented in Section 5. We conclude by presenting the experimental results obtained in Section 6.

2 Deductive Verification for a Toy Language

To illustrate our contribution, we introduce in this section a toy programming language with support for record types, pointer arithmetics, type casts, and updates at arbitrary locations in the memory. We define its syntax and its semantics with respect to an abstract memory model. Then, we sketch the generation of verification conditions for this language in a first order logic based on a generic interface with the underlying memory model.

to be called only from the `main` function.

$\mathtt{n} \in \mathbb{N}, \mathtt{i} \in \mathbb{Z}$	integer constants	$\mathtt{arith}$	arithmetic type in $\{\mathtt{i8}, \mathtt{u8}, \mathtt{i16}, \ldots, \mathtt{u64}\}$
$\mathtt{rt} \in \mathtt{crec}$	record type names	$\mathtt{f} \in \mathtt{cfld}$	field names
$\mathtt{v} \in \mathtt{cvar}$	program variables	$\mathtt{op} \in \mathtt{O}$	unary and binary arithmetic operators

program types	$\mathtt{ctyp} \ni \mathtt{t}$	$::= \mathtt{u} \mid \mathtt{rt} \mid \mathtt{t}[\mathtt{n}]$
scalar types	$\mathtt{styp} \ni \mathtt{u}$	$::= \mathtt{arith} \mid \mathtt{t}\ \mathtt{ptr}$
expressions	$\mathtt{expr} \ni \mathtt{e}$	$::= \mathtt{i} \mid \mathtt{lv} \mid \mathtt{a} \mid (\mathtt{arith})\mathtt{e} \mid \mathtt{op}\ \mathtt{e} \mid \mathtt{e}\ \mathtt{op}\ \mathtt{e}'$
addresses	$\mathtt{addr} \ni \mathtt{a}$	$::= \&\mathtt{v} \mid \mathtt{e} \mid \mathtt{a}.\mathtt{f} \mid \mathtt{a}[\mathtt{e}]$
left values	$\mathtt{lval} \ni \mathtt{lv}$	$::= *_{\mathtt{u}}\mathtt{a}$
statements	$\mathtt{stmt} \ni \mathtt{s}$	$::= \mathtt{lv} = \mathtt{e} \mid \mathtt{assert}\ \mathtt{e}$

Figure 2: Syntax of our C-like toy language

2.1 A Toy Language

Figure 2 lists the constructs of this language. For simplicity, we consider only integer (signed or unsigned) arithmetic types. User defined types are pointer types, static size array types, and record types. A record type declares a list of typed fields with names from a set `cfld`; for simplicity, we suppose that each field has a unique name. Expressions combine integer constants and address expressions using operators in `O` (that includes arithmetic operations, equality and relational comparisons, left and right shifts and bitwise operations), and casts into an arithmetic type. Address expressions contains constant addresses (i.e., locations of program variables), expressions in `expr` of pointer type, an address shifted by a field in `a.f`, and an address shifted by a natural value obtained from the valuation of an integer expression in `a[e]`. A left value of some type `t` is obtained by dereferencing an address expression `a` of type `t ptr` in $*_{\mathtt{t}}\mathtt{a}$. We consider only simple statements for assignment and assertion testing. Classic control statements can be dealt using standard techniques in deductive verification.

We consider only well typed programs. In the following, we model the results of the type checker on a program by a set of semantic functions as follows. For program types, $|.| : \mathtt{ctyp} \to \mathbb{N}$ maps each type to its size in bytes (like `sizeof`). Also, we consider the conversion function $\mathtt{convert}(v, \mathtt{arith})$ for integer values v to some arithmetic type as defined, e.g., in [21]. For a field `f`, $\mathtt{offset}(\mathtt{f})$ gives the offset of this field in its definition record. For any field, variable, expression, or address, $\mathtt{cty}(.)$ returns its type in `ctyp`.

2.2 Abstract Dynamic Semantics

We define the small-step semantics of our language using an abstract memory model that is reminiscent of the first abstract memory model defined in [21, 22] for CompCert, enriched with some notations to increase readability of our presentation. Figure 3 summarizes the elements of this abstract memory model. The states of the memory are represented by an abstract data type *mem*. A memory state stores several *memory blocks*, each block being uniquely identified by a value in *block*. The empty memory is denoted by the constant *emp*. Pointer values, called locations, are represented by pairs (b, o) of block identifier b and a byte offset o inside the block. We denote by *loc* the set of such pairs and provide two operations to build them: $\mathit{base}(\mathtt{v})$ gives the location of a program variable `v`, and $\mathit{shift}(\ell, n)$ computes the location obtained by shifting the offset of location ℓ by n bytes. The shift operation abstracts pointer arithmetics. Memory blocks store values of type *val*, which may be integer or location values. The typing function $\mathtt{cty}(.)$ is extended to locations based on the typing of the program variable used as base of the

Types	mem,	$block$,
	$loc \triangleq$	$block \times \mathbb{N}$,
	$val \triangleq$	$Vint(\mathbb{Z}) \mid Vptr(loc)$
Constants	$emp:$	mem
Operations	$base:$	$\texttt{cvar} \to loc$
	$shift:$	$loc \to \mathbb{N} \to loc$
	$load:$	$mem \to \texttt{styp} \to loc \rightharpoonup val$
	$store:$	$mem \to \texttt{styp} \to loc \to val \rightharpoonup mem$

Figure 3: Abstract memory model

$$
\begin{array}{lll}
[\![\,\texttt{i}\,]\!](m) & \triangleq Vint(\texttt{i}) & \\
[\![\,*_{\texttt{u}}\texttt{a}\,]\!](m) & \triangleq load(m, \texttt{u}, [\![\,\texttt{a}\,]\!](m)) & \\
[\![\,\texttt{a}\,]\!](m) & \triangleq Vptr([\![\,\texttt{a}\,]\!](m)) & [\![\,*_{\texttt{u}}\texttt{a} = \texttt{e}\,]\!](m) \quad \triangleq \mathsf{store}(m, \texttt{u}, [\![\,\texttt{a}\,]\!](m), [\![\,\texttt{e}\,]\!](m)) \\
[\![\,(\texttt{arith})\texttt{e}\,]\!](m) & \triangleq \texttt{convert}([\![\,\texttt{e}\,]\!](m), \texttt{arith}) & [\![\,\texttt{assert e}\,]\!](m) \quad \triangleq \text{if } [\![\,\texttt{e}\,]\!](m) \not\sim 0 \text{ then } m \text{ else } \bot \\
[\![\,\texttt{op e}\,]\!](m) & \triangleq \widehat{\texttt{op}}([\![\,\texttt{e}\,]\!](m)) & \\
[\![\,\texttt{e op e}'\,]\!](m) & \triangleq \widehat{\texttt{op}}([\![\,\texttt{e}\,]\!](m), [\![\,\texttt{e}'\,]\!](m)) &
\end{array}
$$

$$
\begin{array}{ll}
[\![\,\&\texttt{v}\,]\!](m) & \triangleq base(\texttt{v}) \\
[\![\,\texttt{e}\,]\!](m) & \triangleq \ell \text{ if } [\![\,\texttt{e}\,]\!] = Vptr(\ell) \\
[\![\,\texttt{a.f}\,]\!](m) & \triangleq shift([\![\,\texttt{a}\,]\!](m), \texttt{offset}(\texttt{f})) \\
[\![\,\texttt{a[e]}\,]\!](m) & \triangleq shift([\![\,\texttt{a}\,]\!](m), |\texttt{cty}(\texttt{a}[0])| \times \texttt{convert}([\![\,\texttt{e}\,]\!](m), \texttt{u32}))
\end{array}
$$

Figure 4: Generic semantics of expressions ($[\![\,\cdot\,]\!] : mem \to \texttt{expr} \rightharpoonup val$), addresses ($[\![\,\cdot\,]\!] : mem \to \texttt{addr} \rightharpoonup loc$), and statements ($[\![\,\cdot\,]\!] : mem \to \texttt{stmt} \rightharpoonup mem$); $\widehat{\texttt{op}}$ denotes type dependent operations, e.g., addition with pointer operand is done using *shift*.

location. The axiomatization of reading and storing operations is similar to the one in [21, 22]. We use partial functions for them in order to denote potential failures; we denote by $\bot$ the undefined value.

Figure 4 defines the semantics of expressions, addresses, and statements with respect to a memory state m, via the overloaded functions $[\![\,\cdot\,]\!]$. The semantic functions are partial: the undefined case $\bot$ cuts the evaluation.

2.3 Generating Verification Conditions

To fix ideas, we recall here the principle used to generate verification conditions (VC) and we apply it to the above toy language. This general principle is adapted in WP as explained in [3].

Verification conditions are generated from Hoare's triple $\{P\}\ s\ \{Q\}$ with P and Q formulas in some logic theory $\mathcal{L}$ used for program annotations and $\texttt{s}$ a program statement. For this, WP (and most deductive verification tools) employs the efficient weakest precondition computation method proposed in [16, 20]. It computes a formula $R(\vec{v}_b, \vec{v}_e)$ in $\mathcal{L}$ that specifies the relation between the states of the program before and after the execution of $\texttt{s}$, which are represented by the set of logic variables $\vec{v}_b$ (resp. $\vec{v}_e$). The VC built is $\forall \vec{v}_b, \vec{v}_e.\ \big(P(\vec{v}_b) \wedge R(\vec{v}_b, \vec{v}_e)\big) \implies Q(\vec{v}_e)$ and it is given to solvers for $\mathcal{L}$ to check its validity.

To compile a program statement $\texttt{s}$ into a formula $R(\cdot, \cdot)$, the tool uses the dynamic semantics of the language, given in Figure 4 for our toy language. The abstract memory model *mem* used in this semantics is represented by a *memory model environment* (called simply environment)

Types	mem,	loc
	$\mathsf{val} \triangleq$	$\mathsf{Vint}(\mathcal{E}_{\mathbb{I}}) \mid \mathsf{Vptr}(\mathsf{loc})$
Constants	$\mathsf{emp}:$	$2^{\mathtt{cvar}} \to \mathsf{mem}$
Operations	$\mathsf{base}:$	$\mathtt{cvar} \to \mathsf{loc}$
	$\mathsf{shift}:$	$\mathsf{loc} \to \mathcal{E}_{\mathbb{I}} \to \mathsf{loc}_\perp$
	$\mathsf{load}:$	$\mathsf{mem} \to \mathtt{styp} \to \mathsf{loc} \to \mathsf{val}_\perp$
	$\mathsf{store}:$	$\mathsf{mem} \to \mathtt{styp} \to \mathsf{loc} \to \mathsf{val} \to (\mathsf{mem} \times \mathcal{E}_{\mathbb{B}})_\perp$

Figure 5: Interface for memory model environments

that keeps the information required by the VC generation, for example the current set of variables used for modeling the state at the current point. Figure 5 defines a *signature for memory model environments* that captures the essential features required by the compilation of statements in our toy language into VC formulas. In the next sections, we supply several memory model environments that demonstrate the abstraction capabilities of this signature. In particular, we provide in Section 5 an environment that is parameterized by the abstract domain of Eva. Notice that, in WP, the VC generation follows a pass over the program syntax which collects, for each program statement, the safety conditions, (called alarms in the introduction), that eliminate the runs leading to undefined behaviors. For this reason, the VC generation does not collect such constraints in $R(\cdot,\cdot)$ and focuses only on the encoding of the semantics of statements.

The signature in Figure 5 should be compared with the abstract memory model from Figure 3. The environment shall define a type mem providing information about the state of the memory and a type loc encoding information on memory locations. By omitting memory blocks from this signature, we permit a more abstract relation between memory locations and program variables, based only on the operation base. This abstraction allows to define an efficient environment for programs without pointer manipulation (see Section 3). The environment shall provide a type for basic values stored in the memory: integers and locations. The integer values are specified by integer terms in $\mathcal{L}$, denoted by $\mathcal{E}_{\mathbb{I}}$. The empty environment for a set V of active program variables is provided by emp(V). The arithmetic operation on locations is encoded in operation shift, if this operation is supported by the memory model. Indeed, we use an error monad with error value $\perp$ for the result type of some operations to point out that these operations may be defined only under some conditions (e.g., no dereference, no alias, variable used only by reference, ...). We provide in Section 3.1 an example of environment where the operation shift is undefined. The updating of the memory environment store produces a new environment and a formula in $\mathcal{L}$, in the set $\mathcal{E}_{\mathbb{B}}$.[2]

The logic theory used to compile VCs, $\mathcal{L}$, shall satisfy the following constraints. It must be a multi-sorted first order logic that embeds the logic theory used to annotate programs in our toy language. (In Frama-C, the logic theory for annotations is defined using ACSL [5].) It includes the boolean theory (sort $\mathbb{B}$), the bit vector theory (sort $\mathbb{V}$) for bit operations, the integer arithmetic theory (sort $\mathbb{I}$), the array theory (with polymorphic type $array(\alpha,\beta)$ and classic operations for selection $a[k]$ and update $a[k \longleftarrow v]$), abstract data types (or at least polymorphic pairs with component selection by *fst* and *snd*), and uninterpreted functions. We suppose that $\mathcal{L}$ includes a conditional operator if-then-else for term building denoted by

[2] In languages with conditionals, environments should also provide a function $\mathsf{join} : \mathsf{mem} \to \mathsf{mem} \to \mathsf{mem} \times \mathcal{E}_{\mathbb{B}}$ which is used to join execution paths. It returns the new environment to use and a set of equalities which make the environments equal.

$ite(\cdot,\cdot,\cdot)$. We denote by $\mathcal{E}$ the set of logic terms built in $\mathcal{L}$ using the constants, operations, and variables in a set $\mathcal{X}$. We also suppose that an infinite number of fresh variable can be generated. For a logic sort τ, we denote by $\mathcal{E}_\tau$ the terms of type τ. The expressivity of $\mathcal{L}$ determines the precision of the VC generated, and the subset of the programming language constructs that may be dealt precisely by the VC generator. We illustrate this problem in the next section, where we present two memory model environments used by WP.

To ease the reading of environment definitions, we distinguish the logic terms by using the mathematical style and by underlining the terms of $\mathcal{L}$, e.g., $\underline{x+x}$. For example, the logic term $\mathsf{m[l]}\underline{+x}$ is built from a VC generator term $\mathsf{m[l]}$ that computes a logic term of integer type and the logic sub-term $\cdot + x$. For example, the VC generated for the Hoare's triple $\{P\}\ *_{\mathtt{i8}}(\&\mathtt{r.f}) = 5\ \{Q\}$ is $\underline{P \wedge e_1 \implies Q}$ where:

m_0	$\triangleq$	$\mathsf{emp}(\{\mathtt{r}\})$
l_0	$\triangleq$	$\mathsf{shift}(\mathsf{base}(\mathtt{r}), \mathtt{offset(f)})$
m_1, e_1	$\triangleq$	$\mathsf{store}(\mathsf{m}_0, \mathtt{i8}, \mathsf{l}_0, \mathsf{Vint}(5))$
$\underline{P}$		generated from P using environment m_0
$\underline{Q}$		generated from Q using environment m_1

3 Memory Models in WP

This section presents two memory model environments employed by WP for the generation of VCs. Notice that the logic theory $\mathcal{L}$ in which the VCs are encoded is the one provided by the Qed [12] module. Qed includes a programmatic interface to $\mathcal{L}$, several rewriters used to simplify the encoded formulas and translators to the input language of different solvers.

The memory model environments presented here could be combined to obtain an environment that provides the best memory model for each location. For simplicity, we present them separately in this section.

3.1 Simple Memory Model

$\mathsf{mem} \triangleq \mathsf{array}\ (\mathtt{cvar}, \mathcal{X})$ $\qquad$ $\mathsf{loc} \triangleq \mathtt{cvar}$

$$\mathsf{emp(V)} \triangleq \begin{cases} \bot & \text{if any v of V has pointer type} \\ [\mathtt{v}_1 \longleftarrow \alpha_1, \cdots, \mathtt{v}_n \longleftarrow \alpha_n] & \text{with } \{\mathtt{v}_1, \cdots, \mathtt{v}_n\} = \mathtt{V} \text{ and } \alpha_1, \cdots, \alpha_n \text{ fresh in } \mathcal{X} \end{cases}$$

$\mathsf{base}(\mathtt{v}) \triangleq \mathtt{v}$ $\qquad$ $\mathsf{shift}(\mathsf{l}, \underline{e}) \triangleq \bot$

$\mathsf{load}(\mathsf{m}, \mathtt{arith}, \mathsf{l}) \triangleq \mathsf{m[l]}$ $\qquad$ $\mathsf{load}(\mathsf{m}, \mathtt{t\ ptr}, \mathsf{l}) \triangleq \bot$

$\mathsf{store}(\mathsf{m}, \mathtt{t}, \mathsf{l}, \mathtt{Vint}\ (\underline{e})) \triangleq (\mathsf{m}[\mathsf{l} \longleftarrow \alpha], \underline{\alpha = e})$ $\quad$ with $\alpha \in \mathcal{X}$ a fresh variable

Figure 6: Simple memory model environment

The simplest memory model environment provided by WP is limited to programs of our toy language (and of C) that does not employ pointers. Only arithmetic type variables `x` are representable and dereferences are present only in left values $*(\&\mathtt{x})$. For this reason, the environment type mem associates each program variable `v` to one logic variable from $\mathcal{X}$. The locations (in loc) are defined by the set of program variables `cvar`. The

```
int x = 10;
int y = 11;
/*@ assert x == 10; @*/
```

Figure 7: Simple program

implementation of the interface defined in Figure 5 is given in Figure 6. Notice that some operations are not implemented. The advantage of such a memory model is that read-over-writes are statically separated for the prover. Therefore, the program in Figure 7 is compiled into $\underline{x_0 = 10 \wedge y_0 = 11 \implies x_0 = 10}$ where x_0 and y_0 are logic variables (in $\mathcal{X}$) to which the memory model environment maps the variables `x` and `y`. The solver does not need to check that the assignment of `y` does not have any effect on the value of `x`.

3.2 Typed Memory Model

The next environment is restricted to the well-typed subset of the C programming language, which includes our toy language. The separation of memory regions is done per type, which is similar to "components as array" model of Burstall-Bornat. The definition of this environment is given in Figure 8. A location in this memory environment is a pair of integers (b_v, o) where b_v models the base of program variable `v` and o models the offset. Therefore, the environment type loc represents pairs of integer terms computing the base and the offset. The memory type mem maps each scalar type `u` in `styp` to an array logic variable $\alpha_{\mathtt{u}}$ representing the memory region storing values of type `u`. These arrays are indexed by a pair of integers representing locations, i.e., type of $\alpha_{\mathtt{u}}$ is $\mathit{array}(\mathbb{I} \times \mathbb{I}, \mathtt{u})$.

$$\mathsf{mem} \triangleq \mathsf{array}\ (\mathtt{styp}, \mathcal{X}) \qquad \mathsf{loc} \triangleq \mathbb{I} \times \mathcal{E}_{\mathbb{I}}$$

$$\mathsf{emp}(\mathtt{V}) \triangleq [\mathtt{u}_1 \longleftarrow \alpha_1, \cdots, \mathtt{u}_n \longleftarrow \alpha_n] \quad \begin{array}{l}\text{with } \{\mathtt{u}_1, \cdots, \mathtt{u}_n\} \text{ scalar types in } \mathtt{V} \\ \text{and } \alpha_1, \cdots, \alpha_n \text{ fresh array variables}\end{array}$$

$$\mathsf{base}(\mathtt{v}) \triangleq (\underline{b_{\mathtt{v}}}, \underline{0}) \qquad \text{with } b_{\mathtt{v}} \in \mathbb{I} \text{ fixed for } \mathtt{v}$$

$$\mathsf{shift}(\mathsf{l}, \underline{e_{\mathbb{I}}}) \triangleq (\underline{\mathit{fst}(\mathsf{l})}, \underline{\mathit{snd}(\mathsf{l}) + e_{\mathbb{I}}})$$

$$\mathsf{load}(\mathsf{m}, \mathtt{u}, \mathsf{l}) \triangleq \begin{cases} \mathsf{Vint}(\mathsf{m}[\mathtt{u}][\underline{\mathsf{l}}]) & \text{if } \mathtt{u} \in \mathtt{arith} \\ \mathsf{Vptr}(\mathsf{m}[\mathtt{u}][\underline{\mathsf{l}}]) & \text{otherwise} \end{cases}$$

$$\mathsf{store}(\mathsf{m}, \mathtt{u}, \mathsf{l}, \mathsf{Vint}(\underline{e})) \triangleq (\mathsf{m}[\mathtt{u} \longleftarrow \alpha], \underline{\alpha =}(\mathsf{m}[\mathtt{u}])[\underline{\mathsf{l} \longleftarrow e}]) \quad \text{with } \alpha \in \mathcal{X} \text{ a fresh array variable}$$

$$\mathsf{store}(\mathsf{m}, \mathtt{u}, \mathsf{l}, \mathsf{Vptr}(\underline{e})) \triangleq (\mathsf{m}[\mathtt{u} \longleftarrow \alpha], \underline{\alpha =}(\mathsf{m}[\mathtt{u}])[\underline{\mathsf{l} \longleftarrow e}]) \quad \text{with } \alpha \in \mathcal{X} \text{ a fresh array variable}$$

Figure 8: Typed memory model environment

Such a memory model is relevant for a larger class of programs, but the generated VCs are more complex. For example, the formula obtained in this memory model for the program in Figure 7 is $\underline{\left(\alpha_1 = \alpha_0[(b_{\mathtt{x}}, 0) \longleftarrow 10] \wedge \alpha_2 = \alpha_1[(b_{\mathtt{y}}, 0) \longleftarrow 11]\right) \implies \alpha_2[(b_{\mathtt{x}}, 0)] = 10}$ where $\alpha_0, \alpha_1, \alpha_2 \in \mathcal{X}$ are logic arrays variables and $b_{\mathtt{x}}$ and $b_{\mathtt{y}}$ are distinct constants fixed for program variables `x` and `y`.

4 Value Analysis in Eva

The Eva plug-in implements a static *value analysis* based on the principles of abstract interpretation [14]. Abstract interpretation links a *concrete* semantics, typically the set of all possible executions of a program, to a more coarse-grained, *abstract* semantics. Any program transformation must have an abstract encoding that captures all possible outcomes defined by the concrete semantics. This ensures that the abstract semantics is a sound approximation of the runtime behavior of the program. For each instruction of the program, the information inferred by Eva includes two components:

1. the status True/False/Unknown of each alarm generated for the instruction (recall that these alarms prevent any undefined behavior from taking place) and
2. for each memory location involved in the instruction, an over-approximation of the values it may contain.

In this work, both kinds of information above are useful, for different reasons. Firstly, the alarms already proved (status True or False) by Eva are removed from the set of properties to be proven by WP. Secondly, we use the over-approximation inferred for pointer values (which is equivalent to a points-to information) to build the memory model environment (mainly the set of memory blocks) used by WP uses to build VCs (Section 5).

4.1 Abstractions for values

memory abstraction	$\mathtt{mem}^\# \ni \mathtt{m}^\#$
extended integers	$\mathtt{i}^\#_\infty ::= i \mid -\infty \mid +\infty$
abstract integers	$\mathtt{I}^\# \ni \mathtt{i}^\# ::= \{i, \dots\} \mid [\mathtt{i}^\#_\infty \,..\mathtt{i}^\#_\infty], n\%n$
abstract bases	$\mathtt{b}^\# ::= \mathtt{v} \mid \mathbf{0}$
abstract locations	$\mathtt{loc}^\# \ni \mathtt{l}^\# ::= \{(\mathtt{b}^\#, \mathtt{i}^\#), \dots\}$
abstract values	$\mathtt{val}^\# \ni \mathtt{v}^\# ::= \mathtt{l}^\#$

Figure 9: Abstractions used for memory and scalars in Eva

The abstractions used by Eva to represent integers and pointers are given in Figure 9 and detailed below. We omit the details on the abstraction used for memory states (domain $\mathsf{mem}^\#$) which are highly technical and not relevant for this work.

Abstract integers values: If the over-approximation computed for an integer value is a small set of constant values, the abstract value used is exactly this set. If the computed set is large (i.e., exceeds some threshold), it is represented as a potentially unbounded interval with congruence information. For instance, $x \in [3..255], 3\%4$ means that x is such that $3 \leq x \leq 255 \wedge x \equiv 3 \bmod 4$. The congruence information is very important to precisely encode alignment constraints for pointer arithmetics.

Abstract location values: Eva assumes (and verifies) that the program performs no invalid (e.g., out-of-bounds) array/pointer accesses. For this, pointers are seen as offsets with respect to a symbolic block location called *base* $\mathtt{b}^\#$ and have no relation with the locations in the virtual memory space used during the concrete execution. Blocks of different bases are implicitly separated: it is impossible to move from the block of one base to another using pointer arithmetic. Base locations can be (1) the location assigned to a local or global variable, (2) the location of the formal parameter of a function and (3) the **0** location, that stands for the NULL pointer. Offsets are plain integers and abstracted by values in $\mathtt{I}^\#$.

The abstraction of pointers as a set of pairs built from a base location and an offset is slightly more precise than the traditional abstraction in which a pointer is mapped to the pair of (i) the set of possible bases and (ii) the set of possible offsets for all bases. Hence, Eva can represent precisely a pointer that could be either NULL, equal to the address of a variable `x`, or equal to the addresses of the cells 3 to 10 of an array `T` of 16 bits integers. Expressed as an element of $\mathtt{loc}^\#$, and assuming that offsets represent a number of bytes, this set of possible values for a pointer is abstracted by $\{(\mathbf{0}, \{0\}); (\mathtt{x}, \{0\}); (\mathtt{T}, ([4..18], 0\%2))\}$.

Abstract values: Abstract values $\mathtt{v}^\#$ are exactly abstract locations. The abstraction of integer variables is given by the offset part of an abstract location with base **0**.

4.2 Abstract operations

$$[\![\,\mathtt{i}\,]\!]^\#(\mathsf{m}^\#) \triangleq \{(\mathbf{0},\{i\})\} \qquad [\![\,\mathtt{op}(\mathtt{e}_1,\mathtt{e}_2)\,]\!]^\#(\mathsf{m}^\#) \triangleq \widehat{op}^\#([\![\,\mathtt{e}_1\,]\!]^\#(\mathsf{m}^\#),[\![\,\mathtt{e}_2\,]\!]^\#(\mathsf{m}^\#))$$
$$[\![\,\&\mathtt{v}\,]\!]^\#(\mathsf{m}^\#) \triangleq \{(\mathtt{v},\{0\})\} \qquad \mathsf{shift}^\#(\{(b_k^\#,i_k^\#),\dots\},i^\#) \triangleq \{(b_k^\#,i_k^\#\ +^\#\ i^\#),\dots\}$$

$$\begin{aligned}
\mathsf{ct}^\# &: \mathsf{mem}^\# \to \mathtt{styp} \to \mathtt{cvar}\times\mathbb{N}\to\mathsf{val}^\# \\
\mathtt{valid}^\# &: \mathsf{loc}^\#\to\mathbb{N}\to\mathsf{loc}^\# \\
\mathsf{load}^\#(\mathsf{m}^\#,\mathtt{u},\mathtt{l}^\#) &\triangleq \sqcup^\#_{l\in\gamma(\mathtt{l}^\#)}\mathsf{ct}^\#(\mathsf{m}^\#,\mathtt{u},l) \\
[\![\,*_{\mathtt{u}}\mathtt{a}\,]\!]^\#(\mathsf{m}^\#) &\triangleq \mathsf{load}^\#(\mathsf{m}^\#,\mathtt{u},\mathtt{valid}^\#([\![\,\mathtt{a}\,]\!]^\#(\mathsf{m}^\#),|\mathtt{u}|))
\end{aligned}$$

Figure 10: Selected abstract operations

We give in Figure 10 the definition of most interesting abstract operations for the operations of our language. Integers and addresses are injected into the abstractions described above in the obvious way. Binary operators are handled using corresponding abstract operators $\widehat{\mathsf{op}}^\#$. We omit arithmetic casts, that are handled by the integer abstraction. An abstract shift operation $\mathsf{shift}^\#$ is used to shift pointers by an integer: it preserves the base and shifts the offset. The semantics for $[\![\,\mathtt{a.f}\,]\!]^\#$ and $[\![\,\mathtt{a[e]}\,]\!]^\#$ are the same as in Figure 4, except they use $\mathsf{shift}^\#$.

We omit the full definition of abstract semantics for memory reads and updates, because of the complexity of the memory abstraction. We only sketch the semantics for a memory access $[\![\,*_{\mathtt{u}}\mathtt{a}\,]\!]^\#_{\mathsf{m}}$, based on three functions. The first function, $\mathsf{ct}^\#$, has as input an abstract memory $\mathsf{m}^\#$, a scalar type $\mathtt{u}$, and a concrete location (b,o). It reads the content of $\mathsf{m}^\#$ for the base address b, at offset o, over $|\mathtt{u}|$ bytes, and returns it as an abstract value. The second function, $\mathsf{load}^\#(\mathsf{m}^\#,\mathtt{u},\mathtt{l}^\#)$, joins (using the abstract join $\sqcup^\#$) the abstract contents computed by $\mathsf{ct}^\#$ at the concrete locations abstracted by $\mathtt{l}^\#$. This operation requires that $\gamma(\mathtt{l}^\#)$, the concretization of $\mathtt{l}^\#$, is a set of finitely many locations. The last function, $\mathtt{valid}^\#$, takes as argument an abstract location $\mathtt{l}^\#$ and a size s. It selects from $\mathtt{l}^\#$ the pairs $(b^\#,i^\#)$ that allow a *valid* access of s bytes. In particular, this function trims the pairs of $\mathtt{l}^\#$ that would lead to out-of-bound accesses, including those on the base $\mathbf{0}$. (These locations are ruled out by the alarms that guarantee that the instruction executes without an undefined behavior.) Since our abstract bases have a known C type, the number of concrete locations in $\mathtt{valid}^\#(\mathtt{l}^\#,|\mathtt{u}|)$ is always finite. Hence, $[\![\,*_{\mathtt{u}}\mathtt{a}\,]\!]^\#_{\mathsf{m}}$ can be written in terms of $\mathsf{load}^\#$ operations.

4.3 Handling function calls

Eva performs whole-program analyses. Function calls are treated by symbolic inlining. The arguments of the functions are evaluated, assigned to the formal parameters of the callee, and the body of the callee is analyzed. (Recursion is not supported.) In particular, this means that the address of a local variable of the caller can be used transparently in the callee, as shown in Figure 11. The abstraction for `p` in `f` is $\{(\mathtt{x},\{0\})\}$, where `x` is a variable which is *not* syntactically in scope in `f`. This is the main difficulty in translating the results of Eva to another, modular verification tool, e.g. WP, and the reason why we need a custom memory model.

```
void f(int *p) {
  *p = 1;
}
void main() {
  int x;
  f(&x);
}
```

Figure 11: Pointers to local variables

5 Deriving a Memory Model from an Abstract Domain

In this section, we introduce our memory model environment based on a value analysis performed by a static analyzer. The set of disjoint memory blocks building the memory is derived from the bounded set of memory blocks computed by this analysis. We first provide a signature defining what is required from the static analyzer, then define our generic environment on top of this signature, and finally instantiate the signature with Eva.

5.1 Signature required from the static analyzer

$$
\begin{array}{l}
\textbf{sig}\ \mathsf{Domain}\ : \\
\quad \textbf{type}\ \mathcal{V} \qquad\qquad\qquad \textbf{type}\ \mathcal{S} \\
\quad \mathsf{base} : \mathtt{cvar} \to \mathcal{V} \qquad \mathsf{load} : \mathcal{S} \to \mathtt{styp} \to \mathcal{V} \to \mathcal{V} \\
\quad \mathsf{shift} : \mathcal{V} \to \mathcal{E}_{\mathbb{I}} \to \mathcal{V} \\
\quad \mathsf{domain} : \mathcal{V} \to 2^{\mathtt{cvar}}
\end{array}
$$

Figure 12: Domain signature

Figure 12 defines the signature that any static analyzer must implement in order to build our memory model environment on top of its results. The analyzer shall provide a sort $\mathcal{V}$ for the abstraction of values and another for its abstraction of memory states $\mathcal{S}$. In addition, it shall provide basic operations over those sorts, including shifting on abstract values (function shift), querying the domain of a value (i.e., the blocks to which it refers given by function domain) and reading the abstract value at a given abstract location in some abstract memory state (function load). In order to ensure the soundness of our memory model, the static analyzer shall provide sound approximations for the functions in the above signature. We suppose for the rest of this section that this assumption is true.

5.2 Generic Model

Given a domain D satisfying the signature Domain, we define a memory model environment in Figure 5. The separation of memory is statically encoded by the finite number of disjoints blocks associated to bases. As in the typed model in Section 3.2, a pointer value is represented by a pair (b, o) where b is the representation of a base into our logic theory (encoded as an integer), and o is an offset in *bits* from this base. Locations are then encoded as a pair of a pointer of our model, associated with its abstract location $\mathtt{l}^{\#} \in \mathtt{loc}^{\#}$ counterpart. A possible solution is to encode the memory into a single array indexed by base identifiers. This solution is improved using the results provided by the static analysis, i.e., the finite set of bases a pointer may points-to. Then, we do a static separation of the memory in a finite set of arrays, one by base. Values stored at each base (block) are represented by an array $\mathit{array}(\mathbb{I}, \underline{\mathtt{t}})$ with $\underline{\mathtt{t}}$, as defined in Section 3.2.

$$
\begin{array}{ll}
\mathsf{mem} \triangleq \mathsf{array}(\mathtt{cvar}, \mathcal{X}) & \\
\mathsf{loc} \triangleq \mathcal{E}_{\mathbb{I}\times\mathbb{I}} \times \mathtt{loc}^{\#} & \\
\mathsf{emp}(\mathtt{V}) \triangleq [\mathtt{v}_1 \longleftarrow \alpha_1, \cdots, \mathtt{v}_n \longleftarrow \alpha_n] & \text{with } \{\mathtt{v}_1, \cdots, \mathtt{v}_n\} \text{ the bases computed by SA tool} \\
 & \text{and } \alpha_1, \cdots, \alpha_n \text{ fresh array variables in } \mathcal{X} \\
\mathsf{base}(\mathtt{v}) = (\underline{(b_{\mathtt{v}}, 0)},\ \mathsf{D.base}(\mathtt{v})) & \\
\mathsf{shift}((\underline{(b, o)}, \mathtt{l}^{\#}), e_{\mathbb{I}}) = (\underline{(b, o + e_{\mathbb{I}})},\ \mathsf{D.shift}(\mathtt{l}^{\#}, e_{\mathbb{I}})) &
\end{array}
$$

Our generic model could be designed to use the different over-approximations of the abstract state at *each* statement. For the simplicity of the presentation, we consider that the analysis using the domain D is terminated and we use the over-approximation $s^\#$ of the abstract state *for all* the statements simultaneously.

Reading a value from an abstract memory state m is obtained by statically dispatching the actual base value among the possible bases obtained from the domain.

$$\mathsf{load}(\mathsf{m},\mathtt{t},(\underline{(b,o)},\mathtt{l}^\#)) = \begin{cases} \mathtt{Vint}\,(e) & \textit{if } \mathtt{t} \in \mathtt{arith} \\ \mathtt{Vptr}\,\big(e, \mathsf{D.load}(s^\#,\mathtt{t},\mathtt{l}^\#)\big) & \textit{otherwise} \end{cases}$$
$$\textit{where} \begin{cases} e \triangleq \underline{\mathit{ite}(b = b_{\mathtt{v}_1}, \mathsf{m}[\mathtt{v}_1][o], \cdots, \mathit{ite}(b = b_{\mathtt{v}_{n-1}}, \mathsf{m}[\mathtt{v}_{n-1}][o], \mathsf{m}[\mathtt{v}_n][o]) \cdots)} \\ \text{when } \mathsf{D.domain}(\mathtt{l}^\#) = \{\mathtt{v}_1, \cdots, \mathtt{v}_n\} \end{cases}$$

The store operation is implemented with a parallel conditional write on each chunk that may be pointed-to. The generated formula is linear on the size of the set of chunks given by the domain.

$$\mathsf{store}(\mathsf{m},\mathtt{t},(\underline{(b,o)},\mathtt{l}^\#),v) = \mathsf{m}',\underline{e'}$$
$$\textit{where} \begin{cases} v = \mathsf{Vint}(\underline{e'}) \text{ or } v = \mathsf{Vptr}((\underline{e'},\mathtt{l}_v^\#)) \\ \mathsf{m}' = \mathsf{m}[\mathtt{v}_1 \longleftarrow \alpha_1, \cdots, \mathtt{v}_n \longleftarrow \alpha_n] \\ e' \triangleq \underline{\bigwedge_{i=1}^n \mathit{ite}(b = b_{\mathtt{v}_i}, \alpha_i = \mathsf{m}[\mathtt{v}_i][o \longleftarrow e'], \alpha_i = \mathsf{m}[\mathtt{v}_i])} \\ \text{when } \mathsf{D.domain}(\mathtt{l}^\#) = \{\mathtt{v}_1, \cdots, \mathtt{v}_n\} \end{cases}$$

This use of *ite* in load and store is reminiscent of the encoding with arrays. Indeed, if the points-to analysis always returns for $\mathsf{D.domain}$ the set of all variables, the VC generated does not win any concision. On the other hand, if the points-to analysis always returns a singleton for calls to $\mathsf{D.domain}$, the model produces a formula as simple as the one obtained using the simple memory model in Section 3.1, but for a larger class of programs that may use pointers.

5.3 An instance on top of Eva

We now show how to build an instance of the signature Domain of Section 5.1 using the value analysis provided by Eva. The functions required in the signature are given below. Values are simply abstract locations, making most operations immediate. When computing domain, we skip the $\mathbf{0}$ base, because this corresponds to out-of-bounds accesses, implicitly forbidden by the alarms generated for the instruction.

$$\begin{aligned}
&\textbf{module } \mathsf{Eva} : \mathsf{Domain} = \\
&\qquad\qquad \textbf{type } \mathcal{V} \triangleq \mathtt{loc}^\# \\
&\qquad\qquad \mathsf{base}(\mathtt{v}) \triangleq \{(\mathtt{v},\{0\})\} \\
&\qquad\qquad \mathsf{shift}(\mathtt{l}^\#, e_{\mathbb{I}}) \triangleq \begin{cases} \mathsf{shift}^\#(\mathtt{l}^\#,\{k\}) & \text{if } e_{\mathbb{I}} \text{ is a constant } k \\ \mathsf{shift}^\#(\mathtt{l}^\#,([-\infty..+\infty],0\%1)) & \text{otherwise} \end{cases} \\
&\mathsf{domain}(\{(b_k^\#,o_k^\#),\dots\}) \triangleq \{b_k^\#,\dots\} \setminus \{\mathbf{0}\} \\
&\qquad\qquad \textbf{type } \mathcal{S} \triangleq \mathtt{mem}^\# \\
&\qquad\qquad \mathsf{load}(\mathtt{m}^\#,\mathtt{t},\mathtt{l}^\#) \triangleq \mathsf{load}^\#(\mathtt{m}^\#,\mathtt{t},\mathtt{l}^\#)
\end{aligned}$$

The over-approximations computed by Eva on values of expressions can also be used as additional constraints to obtain more precise VCs, especially for integer variables or offsets. However, this is not required by the signature of Figure 12, and is thus optional.

6 Experiments

We have implemented in OCaml the proposed memory model. The implementation is available as a plug-in of Frama-C. It includes 560 LoC for the module that encodes the memory model and implements the interface (signature) required by the WP for the memory models. In addition, the plug-in includes 65 LoC for the glue between Eva and WP, which calls Eva plug-in, interprets its results using the memory model proposed, and launches the WP plug-in. The plug-in will be released within the Frama-C platform once it will be more mature.

The plug-in presents additional features. For example, it transfers more informations from Eva than the sound partitioning of the memory. Indeed, the static analysis of Eva returns also the size of each element of the partition computed and an over-approximation of the set of values for each variable, at each program point. Our plug-in transfers this information about the contents of variables to WP, as additional assertions.

The interface provided by WP allows us to compare our memory model with existing memory models. Unfortunately, most of examples in the test benchmark of WP use only features supported by the existing memory models (i.e., the ones presented in Section 3), and the points-to analysis does not improve the quality of VC generated. We are currently testing our plug-in on the benchmark of the SV-COMP competition. One challenge with this benchmark is to obtain code annotated for WP, because many interesting examples involve loops, for which a loop invariant is often required for WP. We are currently experimenting with exporting the results of Eva for the contents of variables as invariants in order to avoid manual annotations.

An example that is present in the benchmark of WP and which motivated this work is the one presented in the introduction (see Section 1). In this example, the default memory model of WP (see Section 3.2) cannot prove the post-condition of function `copy`. Since separation is done per type, the model is not able to specify that parameters `a` and `b`, which have the same type, cannot alias. Therefore, the following clause

$$(a + \mathsf{sizeof}(a) \leq b) \vee (b + \mathsf{sizeof}(b) \leq a)$$

is added to the generated VC, unless specified as a pre-condition of the function. The invariants inferred by the analysis with Eva for this example specify that `a` (resp. `b`) points-to a block of memory indexed by variable `t` (resp. by `u`). From the declaration in `main`, Eva infers that those blocks are disjoint. Our plug-in transfers this information to WP through our memory model, and this is sufficient to automatically prove the post-condition.

7 Conclusion

To summarize our contributions, we have defined a framework that allows to transfer the knowledge inferred by a static analyzer on the program memory regions to a deductive verification tool that supports generation of verification conditions parameterized by a memory model. One of the challenging point was to find a simple formalization of this framework, which is independent of the specificities of tools employed. We implemented this framework in Frama-C, and reported on preliminary experimental results.

We hope to be able to report on additional benchmark for which a collaboration of WP with Eva is fruitful in the near future. We also envision various directions as potential future works.

First, one of the current limitations of WP lies in its handling of pointer casts and unions (when used for type punning): most programs are statically rejected as impossible to be encoded precisely within the typed memory model. But the low-level memory model of Eva gives us precise memory information, which could be used to determine whether the casts endanger the

soundness of using the typed memory model. Alternatively, we could define an entirely new model for blocks whose contents are used in type-unsafe way.

Second, another interesting extension would be to support dynamic allocation and deallocation. Currently, the model used by Eva for calls to `malloc` and `free` uses a weak update semantics, while WP does not handle these functions. It is likely that some form of support on the WP side, especially on the interface of memory models, will be required.

Third, we mentioned in Section 6 that we export more than the memory partitioning information to WP. However, the information which is transferred remains limited to simple cases. We believe it is possible to improve this exchange of information, typically for arrays that may be represented very concisely by Eva.

Finally, we would also like to extend the information exchanged with WP w.r.t. pointers. Currently, we rely on the fact that pointers pointing to different blocks are separated. However, it would also be interesting to infer more fine-grained separation. Typically, two pointers pointing to the same block, but in separate parts of this block. This would permit proving that it is safe to call `copy(&t[0],&t[2]);` in the example of Figure 1. This information could be inferred from the abstract offsets of the pointer values. Another interesting information to exchange is the validity of pointers, so that the precondition of `copy` also becomes superfluous. This should be quite straightforward, due to the validity information inferred by Eva and used to check that pointer accesses are in bounds.

References

[1] S. Bardin, M. Delahaye, R. David, N. Kosmatov, M. Papadakis, Y. L. Traon, and J. Marion. Sound and quasi-complete detection of infeasible test requirements. In *ICST'15*, pages 1–10. IEEE Computer Society, 2015.

[2] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of FMCO*, number 4111 in LNCS, pages 364–387. Springer, 2005.

[3] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *Proceedings of PASTE'05*, pages 82–87. ACM, 2005.

[4] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Proceedings of CASSIS*, LNCS, pages 49–69. Springer, 2004.

[5] P. Baudin, J. C. Filliâtre, T. Hubert, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI C Specification Language (preliminary design V1.2)*, preliminary edition, May 2008.

[6] S. Böhme and M. Moskal. Heaps and data structures: A challenge for automated provers. In *Proceedigns of CADE-23*, volume 6803 of *LNCS*, pages 177–191. Springer, 2011.

[7] D. Bühler. *Structuring an Abstract Interpreter through Value and State Abstractions*. PhD thesis, University of Rennes, 2017.

[8] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *SAS*, volume 4134 of *LNCS*, pages 182–203. Springer, 2006.

[9] B.-Y. Chang, X. Rival, and G. Necula. Shape analysis with structural invariant checkers. In *Proceedings of SAS*, volume 4634 of *LNCS*, pages 384–401. Springer, 2007.

[10] S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamarić. A low-level memory model and an accompanying reachability predicate. *STTT*, 11(2):105–116, February 2009.

[11] W. Chin, C. David, H. H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.*, 77(9):1006–1036, 2012.

[12] L. Correnson. Qed. computing what remains to be proved. In *Proceedings of NFM*, volume 8430 of *LNCS*, pages 215–229. Springer, 2014.

[13] L. Correnson and F. Bobot. Exploring memory models with Frama-C/WP, 2017. Personal comunication.

[14] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[15] J. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *Proceedings of CAV*, volume 4590 of *LNCS*, pages 173–177. Springer, 2007.

[16] C. Flanagan and J. B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. *SIGPLAN Not.*, 36(3):193–205, Jan. 2001.

[17] T. Hubert and C. Marché. Separation analysis for deductive verification. In *Proceedings of HAV*, pages 81–93, Braga, Portugal, Mar. 2007.

[18] B. Jacobs and F. Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, 2008.

[19] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A software analysis perspective. *Formal Asp. Comput.*, 27(3):573–609, 2015.

[20] K. R. M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, 2005.

[21] X. Leroy, A. W. Appel, S. Blazy, and G. Stewart. The CompCert Memory Model, Version 2. Research Report RR-7987, INRIA, June 2012.

[22] X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *J. Autom. Reasoning*, 41(1):1–31, 2008.

[23] F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. In *Proceedings of CADE-19*, volume 2741 of *LNCS*, pages 121–135. Springer, 2003.

[24] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of CSL*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001.

[25] R. Piskac, T. Wies, and D. Zufferey. Automating separation logic with trees and data. In *Proceedings of CAV*, pages 711–728. Springer, 2014.

[26] Z. Rakamaric and A. J. Hu. A scalable memory model for low-level code. In *Proceedings of VMCAI*, volume 5403 of *LNCS*, pages 290–304. Springer, 2009.

[27] P. Sotin, B. Jeannet, and X. Rival. Concrete memory models for shape analysis. *Electr. Notes Theor. Comput. Sci.*, 267(1):139–150, 2010.

[28] W. Wang, C. Barrett, and T. Wies. Partitioned memory models for program analysis. In *Proceedings of VMCAI*, volume 10145 of *LNCS*, pages 539–558. Springer, 2017.