Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Functional Dependencies of C Functions *via* Weakest Pre-Conditions

Pascal Cuoq[1], **Benjamin Monate**[1], **Anne Pacalet**[2], **Virgile Prevosto**[1]

[1] CEA, LIST, Laboratoire de Sûreté des Logiciels
Boîte courrier 94
91191 Gif-sur-Yvette cedex
e-mail: *firstname.lastname*@cea.fr
[2] INRIA Sophia Antipolis
2004 route des Lucioles - BP93
06902 Sophia Antipolis Cedex
e-mail: anne.pacalet@sophia.inria.fr

The date of receipt and acceptance will be inserted by the editor

Abstract. We present *functional dependencies*, a convenient, formal, but high-level, specification format for a piece of procedural software (function). Functional dependencies specify the set of memory locations which may be modified by the function (*i.e* the frame condition), and for each modified location, the set of memory locations that influence its final value. Verifying that a function respects pre-defined functional dependencies can be tricky: the embedded world uses C and Ada, which have arrays and pointers. Existing systems we know of that manipulate functional dependencies, Caveat and SPARK, are restricted to pointer-free subsets of these languages. This article deals with functional dependencies in a programming language with full aliasing.

We show how to use a weakest precondition calculus to generate a verification condition for pre-existing functional dependencies requirements. This verification condition can then be checked using automated theorem provers or proof assistants. With our approach, it is possible to verify the specification as it was written beforehand. We assume little about the verification condition generator and its internals. Our work takes place in the C analysis framework Frama-C, where an experimental implementation of the technique described here has been implemented on top of the WP plugin in the development version of the tool.

1 Introduction

Critical embedded software often needs to be assessed against a certain number of criteria, depending on criticality level and application domain. These criteria are described in norms, such as DO-178B [29] for the avionics industry, or the definition of Safety Integrity Levels (SIL) [7] for the railway industry.

In this context, properties such as the absence of runtime errors, absence of dead-code, or functional specifications, are verified. Different techniques have been invented to help check these properties: tests with more or less stringent coverage criteria, model-checking [8], static analysis and abstract interpretation [9], and deductive verification [14].

The DO-178 standard, for instance, lists in its "Reviews and Analyses of the Source Code" 6.3.4 section:
"*Compliance with the software architecture:* The objective is to ensure that the Source Code matches the data flow and control flow defined in the software architecture."

This article focuses on a particular class of lightweight specifications, that we call *functional dependencies*, that are exactly designed to answer the "data flow" part of this requirement in a context with aliasing (pointers can be found in code subject to the less critical levels of DO-178). Functional dependencies of a function `f` express the following information.

- the set $\overrightarrow{x}$ of locations that may be written to during `f`'s execution. This is also known as the *frame condition* of the function.
- for each $x \in \overrightarrow{x}$, the set $\overrightarrow{y_x}$ of locations that influence the final value of x.

In other words, the code of `f` must be such that there exists mathematical functions ϕ_x such that the value of each x at the end of the function is $\phi_x(\overrightarrow{y_x})$. In addition, any location which is not present in $\overrightarrow{x}$ must be unchanged by `f`. The set $\overrightarrow{x}$ of locations that may be written to during an execution of `f`, is similar to the information found in `assignable` clauses of JML or in the notion of memory footprint [5]. A technique to verify this first part of functional dependencies has been proposed in [24], and the present article only deals with the additional information on the way the new values are computed carried out by $\overrightarrow{y_x}$. Functional dependencies are a popular feature of critical code verification systems such as Caveat [27] (where they are called "From" clauses) and SPARK [26] (where they are called "derives" clauses). Both systems have been proven through industrial use.

Although functional dependencies are much less expressive than function contracts as popularized by Eiffel [22],

they are useful because they match the way engineers think about their systems. Namely, they express both the expected insulation between system components, and the data-flow inside a component. As such, writing functional dependencies specifications is easier than writing full function contracts. In addition, functional dependencies are also easier to verify. And in practice, accurate dependencies for a function `f` constitute a useful lemma to have in the verification of a caller `g` of `f`. That is, the functional dependencies of `f` play an important role at the proof level, to abstract the side-effects of a call to `f` inside `g`. The guarantee, provided by `f`'s functional dependencies, that all locations outside of $\overrightarrow{x}$ are left unchanged, can be sufficient to establish a property about `g`, without having to provide a complete specification of auxiliary function `f`. When the property to verify for the caller `g` does require more information about `f`, the functional dependencies of `f` ensure that the ϕ_x functions from the above definition are a correct abstraction of what the function does. The necessary functional specification for `f` can be provided in terms of axiomatization of the ϕ_x functions.

In some cases, although ϕ_x appears in the proof obligation for a property of `g`, the property may be provable without any axiomatization of ϕ_x. An example whose significance will appear later is if the predicate has the shape $C[\phi_x(t)] = C[\phi_x(t)]$ for some context C.

Let us now introduce an example with aliasing. As the approach described in this paper is currently being implemented in the Frama-C toolset for analysis of C programs [11], we use a small C function as illustration:

```
int x, y, z, *p;
void f(int c) {
  if (c) p = &y; else p = &z;
  if (!c) *p = x + 1; else x = *p + 1;
}
```

In the above code, the globals `x`, `z` and `p` may be written to during an execution of `f`. Variable `x`'s final value depends on `c`, `y`, and `x` (the latter because `x` may also keep its initial value). Variable `z`'s final value depends on `c`, `x` and `z`.

In presence of aliasing, the manual verification of dependencies is rarely immediate, because they are so synthetic. For instance, one has to remark that `*p` is written to only when `p` contains the address of `z`, meaning that only `z` can be modified by this assignment. Computing or verifying the optimal dependencies of this simple example requires to be able to take into account alias relations between locations, and to reason on the truth value of each test. A naive data-flow analysis would conclude that since `p` can point to either `y` or `z`, variable `x` depends on both, and also that both `y` and `z` may be modified by `f` (with their new values depending on `x` and `c`). In larger, real-life examples, dependencies computed by data-flow analysis are in general over-approximated, requiring manual reviews to decide if the computed additional dependencies are spurious or indicate an actual failure to satisfy the specification.

Caveat [27], one of Frama-C's ancestors, limits over-approximations by using Hoare logic [16]. The precision is only theoretical if the user does not find the courage to write optimal loop invariants and relies instead on the inferred ones; but at least the possibility is there to refine the results until they are satisfactory—a compromise shared by our approach. The main difference between Caveat and our proposal is that Caveat only infers functional dependencies from scratch. It does not verify provided dependencies: differences between specified dependencies and computed ones have to be reviewed by the human verifier. The verification of provided dependencies that have been written in advance is the use case we are advocating in this article. Our approach uses Hoare logic to compute a verification condition that corresponds to user-specified dependencies. The verification condition can then be established in an automatic or interactive prover. Even if there is human intervention in the verification of the proof obligation, the process is formal, in contrast to the reliance on informal reviews of differences between specified and computed dependencies.

The SPARK [26] system uses a data-flow analysis to infer dependencies that can be compared to the dependencies that have been written as specifications. Since aliasing is limited[1], and as long as the specifications do not try to be too subtle, the probability is high that the inferred dependencies are equal modulo reordering to the specified dependencies, allowing the verifier to conclude that the function behaves as specified. However, false negatives can be caused not only by the imprecision of the data-flow analysis, but by differences between equivalent ways to specify dependencies caused by aliasing. In order to expand the usefulness of functional dependencies to wider languages subsets where aliasing is more common, we think it is necessary to tackle this problem.

2 ACSL

2.1 Presentation

ACSL [2] is a specification language for C, quite similar to what JML [4] provides for Java. ACSL is a first-order logic whose basic terms are the pure expressions of C, and has built-in predicates and logic functions to deal with C pointers. Examples of built-in predicates are **`\valid`**, that expresses that a pointer is valid, **`\base_addr`**, that returns the base address of a valid pointer, that is the address at which the memory block enclosing the pointer starts, or **`\separated`** which expresses that two sets of locations are disjoint. The latter is comparable in intent to the $*$ operator found in separation logic [28].

Functional specifications in ACSL are based on the notion of function contract. A contract expresses the *pre-conditions* the function **`requires`** from its callers together with the *post-conditions* it **`ensures`** when it returns. In an **`ensures`** clause, it is possible to use the **`\old`**`(...)` operator to refer to the value of an expression as evaluated in the pre-state of the function. For instance, a `swap` function which exchange the

[1] SPARK does not have pointers, but has call-by-reference

content of the two (valid) pointers given as argument could be specified like this:

```
/*@ requires \valid(p) && \valid(q);
    ensures *p == \old(*q) && *q == \old(*p);
 */
void swap(int* p, int* q);
```

2.2 Functional dependencies in ACSL

ACSL allows to describe, in a contract, the set of locations that might be modified by a function and, for each such location, to give its functional dependencies. This is done through **assigns ... \from ...;** clauses. **assigns** clauses can be conservative: the presence of x in an assigns clause does not imply that x is necessarily modified. Conversely, the fact that x does not appear in any **assigns** clause does not necessarily mean that x stays unchanged during function execution, because of possible aliases. More precisely, for a contract of the following form:

```
/*@ assigns loc_1;
    ...
    assigns loc_n; */
```

and a location *loc*, the following implication holds:

```
\separated(loc,\union(&loc_1,...,&loc_n))
    ==> *loc == \old(*loc);
```

Similarly, a clause of the form

```
assigns loc \from loc_1,...,loc_n;
```

implies that the final value of *loc* does not depend on any location which is **\separated** from loc_1,...,loc_n.

Looking back at the previous examples, admissible functional dependencies for f and swap would be:

```
int x,y,z,*p;
/*@  assigns x \from c, y, x ;
     assigns z \from c, x, z ;
     assigns p \from c ;        */
void f(int c);

/*@  assigns *p \from *q ;
     assigns *q \from *p ;      */
void swap(int* p, int* q);
```

Note the clause concerning f's modifications of p. Variable p receives the address of either x or y. Addresses of variables are not themselves locations and are fixed for the whole execution of the program. In other words, p is assigned some abstract constant, and its final value depends only on c[2].

[2] In order to make functional dependencies better suited to exploitation by forward-propagation analyzers, a format for specifying that p may receive the address of globals x and y is an addition we are considering to ACSL contracts, but the fact remains that p does not depend on &x in the same sense that it depends on c.

2.3 Specifying Effects of Loops

Loops play a special role in deductive verification. They are annotated with *loop invariants*, properties which inductively stay valid from one loop step to the next one. More specifically, a loop invariant must hold when execution reaches the loop and be preserved by the execution of each loop step. This allows to abstract the effects of the loop: the only fact known about the program state at the exit of the loop is that the invariant holds (and that the condition is false). ACSL provides **loop invariant** annotations for invariant properties, but also **loop assigns** for the functional dependencies of a loop. This clause is a summary of all the assignments performed since entering the loop. Hence, if we consider the following function which stores partial sums of the elements of array a into s:

```
void sum() {
  int i = 0;
  while (i < length) {
    s[i] = a[i];
    if (i>0) s[i]+=s[i-1];
    i++;
  }
}
```

A corresponding **loop assigns** clause can be:

```
/*@
 loop assigns s[0..i] \from s[0..i],a[0..i];
 loop assigns i \from i;
 */
```

Even though s[i] appears to only depends from a[i] and s[i-1] in the current loop step, the **\from** clause must take into account all of the first n iterations.

2.4 Aliasing

As a last example demonstrating how tricky aliases make formal specifications in general (and functional dependencies are no exception), consider the function:

```
void h(void) {
  G = A;
  *p = B;
}
```

This function does not satisfy the dependencies

```
/*@ assigns G \from A ;
    assigns *p \from B ; */
```

because if p points to G, the value of G after calling h cannot be said to depend only on A. The dependencies in each **assigns** clause in a contract such as the above has to be satisfied on its own, even when the destination locations in some of them are aliased. The function h satisfies either of the following contracts:

```
/*@ requires \separated(G, *p) ;
    assigns G \from A ;
    assigns *p \from B ;  */
```

or

```
/*@ assigns G \from A, p, B ;
    assigns *p \from B ;  */
```

3 Frama-C

3.1 Abstract Interpretation

Frama-C provides several analysis tools for C programs in the form of plug-ins. ACSL is used as annotation language and for inter-plug-in communication. One plug-in, called *Value Analysis*, computes among other things functional dependencies *via* abstract interpretation, but as previously emphasized, they may be overapproximated, and, even when they are exact, may be expressed in a different form than expected. For instance, the plug-in computes functional dependencies of function `f` in section 1 as follows:

```
$ frama-c -deps -main f -lib-entry test.c
[from] Function f:
         x FROM y; z; c; (and default:true)
         y FROM x; c; (and default:true)
         z FROM x; c; (and default:true)
         p FROM c; (and default:false)
```

`default``:true` expresses the fact that the corresponding variable might stay unchanged during execution, while **`default`**`:false` means that `p` is always overwritten. As the results show, the plugin fails to detect that `y` never can be assigned and that `x` can only depend on `y`.

The value analysis can take advantage of hints [10]. For this example, allowing several states to be propagated separately (without joins) in a fashion comparable to trace partitioning [21] almost suffices, because the conditions that need separate study are exactly the conditions of the **`if`**s in function `f`. Unfortunately, the value analysis cannot represent the set of values `c` such that `c!=0`, so it is unable to take advantage of the information in the **`if`** `(c)` statement.

If we add the following annotation

```
//@ assert c < 0 || c==0 || c>0;
```

the assertion is proved by the value analysis (it is easy to see that the disjunction always hold), and then used as a hint that the three cases benefit from being propagated separately. The result becomes:

```
$ frama-c -deps -main f -lib-entry \
          test.c -slevel 10
[from] Function f:
         x FROM y; c; (and default:true)
         z FROM x; c; (and default:true)
         p FROM c; (and default:false)
```

This is equivalent to the ACSL specification of section 2.2, modulo the order in which the locations appear on the right-hand side of the `FROM` and the distinction made by this analysis between a variable that keeps its value or is recomputed using its old value. For more complicated functions, it may be impractical to force the value analysis to compute the functional dependencies one expects. We only succeeded in this case because the code was a toy example. Slightly more complex expressions such as conditionals or array indices are already enough to make dependencies inferred this way hopelessly over-approximated.

3.2 Deductive Verification

Other Frama-C plug-ins are based on weakest pre-condition computations and generate verification conditions. The Jessie Frama-C plug-in is currently incorporated in the distribution of the Why tool [31], while the another verification conditions generator, Wp, is currently under development and should be released in the next version of Frama-C. One of the main differences between these plug-ins lies in their memory model. In its original formulation, Hoare logic [14, 16] uses programming languages that do not explicitly manipulate memory. Thus, in order to use deductive verification for the C language, one has to encode all pointers operations into the logic. A very low-level memory model would consider pointers as offset in one big array of bytes. However, this leads to intractable proof obligations, as any write operation might have an impact on all allocated cells. More abstract memory models guarantee that a given assignment can only impact a certain class of cells. A well-known example of such models is Burstall-Bornat [3]. This model use static type information to separate cells that cannot be aliases of each other: writing in a pointer to **`char`** won't modify the content of a pointer to **`int`**, and writing to `x->f1` preserves the content of `y->f2`. These abstractions allow for more manageable proof obligations, but they prevent use of some C constructions. For instance, the Burstall-Bornat model can not handle programs containing heterogeneous pointer casts. It is therefore important to be able to use various memory models depending on the program under verification.

Jessie [24] uses a refined version of the Burstall-Bornat model allowing for more C constructions than the original presentation, while Wp allows several memory models to be used with a generic precondition engine. Since we want all weakest precondition plug-ins to be usable for the verification of functional dependencies, we design a method that is independent from a particular memory model and builds on basic operations that any such plug-in must have. Section 4 presents these operations, while section 5 explains how they can be used to generate verification conditions for functional dependencies.

4 Weakest Pre-Conditions

4.1 Hoare Logic

Hoare logic manipulates triples of the form $\{P\}s\{Q\}$, where P and Q are first-order formulas and s is a program. Intuitively, if execution of s starts in a state where P holds and the

execution terminates normally, then Q holds in the end-state. The weakest pre-condition for a formula Q and a program s is the formula P_0 such that for any P verifying $P \Rightarrow P_0$, $\{P\}s\{Q\}$ holds. Provided loops are annotated with invariants (see sections 2.3 and 4.4), it is possible to compute weakest pre-condition of a whole function. The *wp* function takes as input a formula and a program and computes their weakest pre-condition.

Hoare's original work [16] did not include the convenience \old construct. This construct, on which our presentation relies, can be implemented by treating any $\backslash\text{old}(e)$ logic expression as an atom during the computations (in particular, substitutions during the weakest precondition computations do not descend inside \old). The $\backslash\text{old}(e)$ is transformed into e at the very end of the computations, when the predicate is at the level of the entry of the function.

4.2 An API for Generating Verification Conditions of Functional Dependencies

wp is not the only function that we need for generating our verification conditions. Since we aim at being independent of the memory model, we only assume that there is a function $[\![.]\!]$ that translates an lvalue in C [30] syntax (`x`, `t[i]`, `*p`, `s->f`, ...) into the appropriate logic term. The way this translation work is dependent on the memory model. In Hoare's original presentation [16], variables were translated as themselves (variable `x` in the program was translated as logic variable `x` in program predicates) and neither arrays nor pointers were supported. A memory model supporting arrays and pointers may introduce an explicit map from addresses to values, typically called a store. In this latter case, the translation from C lvalue to logic term is less immediate, but an existing verification condition generator must have such a translation internally: it uses it each time it handles an assignment `lvalue = expr;`. This is typically done by using on the logical side *functional arrays*. Functional arrays are objects that can be applied to two functions: $select(a, i)$ to retrieve the i-th element of array a and $update(a, i, v)$ which returns a copy of array a except that the value at index i is v. More formally, $update$ and $select$ are defined by the following axioms:

$$select(update(a, i, v), i) = v$$
$$select(update(a, i, v), j) = select(a, j) \text{ if } i \neq j$$

Typical memory models separates the store in one or more arrays, and pointers are seen as indices to these arrays. Two distinct arrays represent separated region of the memory. Thus, in order for the model to be correct, two pointers must be translated as indices to two distinct arrays only if they are known to be pointing to separated locations. A safe representation would use only one array of **`char`**, but this means that any assignment (translated as an $update$) potentially invalidates all the previous values present the array, leading to proof obligations that become quickly intractable. More abstract memory models uses several arrays, for instance according to the static type of the pointers. This basic information can be further refined to separate more pointers in distinct arrays, for instance using a region analysis [17], which separates the memory in region known to be separated, or on the contrary to accommodate for C constructions that break static type safety (union and casts) [23], but the general idea is the same. In a basic model, the translation of lvalues mentioned above would look like this (assuming all of them have type `int`):

$$\begin{aligned}
[\![\text{x}]\!] &= x \text{ if the address of x is not taken} \\
[\![\text{x}]\!] &= select(int_array, x) \\
[\![*\text{p}]\!] &= select(int_array, p) \\
[\![\text{t[i]}]\!] &= select(int_array, t + i) \\
[\![\text{s->f}]\!] &= select(int_array, select(f_array, s))
\end{aligned}$$

A scalar variable x can only be represented in the logic by a scalar variable if its address is not taken. Otherwise, it must be treated like `*(&x)` to account for possible aliasing with a pointer `p`. Dereferencing a pointer of a certain type amounts to $select$ its index on the corresponding array, possibly with a shift for the bracket notation `t[i]`. In the end, accesses to the field of a structure are translated in two operations. Each field has its own array (as distinct fields are always separated from each other, this is known as the "component as array trick" in [3]), which can be used to retrieve an index to the array corresponding to the appropriate type.

In addition, the formulas produced by *wp* can have free variables. In [16], the free variables correspond to variables of the program; in memory models with an explicit store, the array(s) representing the store typically appear(s) as free variable(s). We assume a function *close* which, given a formula, performs universal quantification over its free variables.

We also require of the verification condition generator a function α for converting a formula describing a memory state to a formula describing a different, fresh memory state. In [16], α would simply rename all free variables of the formula to fresh variables. In more sophisticated memory models, the precise behavior of function α would very much depend on modelization details. For instance, offsets inside C arrays must be the same in P and $\alpha(P)$ since they are independent from the program state itself.

4.3 Function Calls in Hoare Logic

A function call in Hoare logic acts like a cut in natural deduction. If `f` has pre-condition *Pre*, post-condition *Post*, and frame condition *Assign* then it must proved at the call point that *Pre* holds, and the only thing known about the locations that have been written by `f` is what *Post* says. More specifically, with $\overrightarrow{x}$ being the formal arguments of `f`, we have

$$\begin{aligned}
wp(\texttt{f}(\overrightarrow{v}), Q) = \; & Pre[\overrightarrow{x} \leftarrow \overrightarrow{v}] \\
& \wedge \forall store'.\, havoc(store, store', Assign) \\
& \qquad \wedge Post[\overrightarrow{x} \leftarrow \overrightarrow{v}; store \leftarrow store'] \\
& \qquad \Rightarrow Q[store \leftarrow store']
\end{aligned}$$

This is known as the adaptation rule [15]. Section 5.3 will show how this rule applies when `f`'s specification itself takes

the form of functional dependencies. In presence of aliasing, specifying the effects of the call can be complicated and is highly dependent on the memory model chosen. At our level, we just assume the existence of a predicate *havoc* where $havoc(store_1, store_2, loc)$ says that $store_1$ and $store_2$ are equal except for the locations contained in the set loc, which are mapped to arbitrary values in $store_2$. This predicate is equivalent to the **havoc** command of Boogie [1] and every verification condition generator for C should have it in one form or another.

As in section 4.2, depending on the memory model, quantification over $store'$ may in fact involve quantification over several arrays, and similarly the *havoc* predicate may be a conjunction of *havoc* over these arrays. For instance, in our basic model, the axiomatic definition of havoc would be:

$$\forall a_1, a_2, i, S.\ havoc(a_1, a_2, S) \wedge i \notin S \Rightarrow select(a_2, i) = select(a_2, i)$$

and given a function f specified like this (assuming that `struct S` has an `int` field `f`):

```
/*@ assigns *p, s->f; */
void f(int *p, struct S* s);
```

we would have:

$$\begin{array}{l} wp(\mathrm{f(q,t)}, Q) = \\ \quad \forall int\_arr'. \\ \quad\quad havoc(int\_arr, int\_arr', \{q\} \cup \{select(f\_arr, t)\}) \\ \quad\quad\quad \Rightarrow Q[int\_arr \leftarrow int\_arr'] \end{array}$$

Since the *havoc* predicate on f_arr (which maps pointers to `S` to pointers to `int`) would be over an empty set of location (we modify the content of `t->f`, not its address), we can use the same array before and after the call. On the contrary, two indices of int_arr might have been modified, and we have to use a new array, related to the old one through the *havoc* predicate.

4.4 Loops in Hoare Logic

Weakest pre-condition computation of loops bears some similarities with a call to some function whose pre- and post-conditions would be the invariant of the loop, with the additional proof obligation that the invariant must be preserved at each step. Again, we suppose that the *Assign* of the loop are present. Given a loop of the form `while(c) { b; }` where `c` is a side-effect free expression, and an associated invariant I, its weakest pre-condition is the following formula:

$$\begin{array}{l} wp(\mathrm{while(c)\ \{\ b;\ \}}, Q) = \\ \quad I \\ \quad \wedge \forall store'.havoc(store, store', Assign) \wedge I' \wedge \neg c' \Rightarrow Q' \\ \quad \wedge \forall store'.havoc(store, store', Assign) \wedge I' \wedge c' \\ \quad\quad \Rightarrow wp(b, I)[store \leftarrow store'] \end{array}$$

where $I' = I[store \leftarrow store']$ and $c' = c[store \leftarrow store']$.

5 A Formula for Verifying Functional Dependencies

5.1 Presentation

We seek verification conditions for functional dependencies. This task is twofold: on the one hand, we have to establish that only the locations $(loc_1, ..., loc_n)$ mentioned on the left of a `\from` might be modified, and on the other hand that each of these locations depends only on the locations present in the right-hand side of the corresponding clause. As said above, the first part amounts to verifying the following post-condition:

```
\forall char *loc.
  \separated(loc,\union(&loc_1,...,&loc_n))
     ==> *loc == \old(*loc)
```

Computing the weakest pre-condition of this clause in the Jessie memory model has been studied in [24], based on similar work for JML-annotated Java programs [20]. In this paper, we focus on the second part. For that, we start from clause `assigns` e `\from` $\overrightarrow{deps}$, program s, and precondition *Pre*. A first idea would be to compute the following weakest pre-condition:

Definition 1.

$$\begin{array}{l} deps\_ho(e, \overrightarrow{deps}, s, Pre) = \\ \exists \phi, \\ \quad close\left(Pre \Rightarrow wp\left(s, [\![oldloc(e)]\!] = \phi(\overrightarrow{[\![\backslash\mathrm{old}(deps)]\!]})\right)\right) \end{array}$$

`assigns` clauses are evaluated in the pre-state, but since we perform a weakest pre-condition computation over it

$$[\![oldloc(e)]\!] = \phi(\overrightarrow{[\![\backslash\mathrm{old}(deps)]\!]})$$

represents a property which is supposed to hold in the post-state. *oldloc* is derived from \old and indicates that if e is an access to a memory location, such as `*p`, the memory location itself is evaluated in the pre-state, while the access is evaluated in the post-state.

5.2 Proof Obligation for Functional Dependencies

A major issue is that the formula above is higher-order (there is a quantification over a function), making it unsuitable for first order automated provers that are usually used by Frama-C. Higher-order theorem proving is much more difficult than first order, and there exist only very few and experimental automated tools for this task. In addition, proving the formula above amounts in many cases to provide a witness for ϕ, that is to give the exact functional expression of x in terms of the inputs of the function. This is a stronger demand than simply collecting functional dependencies.

We now devise a semantically equivalent first-order formula. Informally, we simulate two distinct executions of s, for which we only assume that they share the same values for $\overrightarrow{deps}$ and e. First, we introduce fresh logic function symbols $\overrightarrow{f}$ and define the predicate Pre_{deps} as $\overrightarrow{[\![deps]\!]} = \overrightarrow{f()}$. Second,

we use a fresh predicate symbol A standing as a predicate observer of e in $A([\![\mathit{oldloc}(e)]\!])$. It is used as an argument of the *wp* to extract the semantics of the computation of e. The first-order verification condition we propose is:

Definition 2.

$$\mathit{deps\_fo}(e, \overrightarrow{\mathit{deps}}, s, \mathit{Pre}) = \mathit{close}\left(\begin{pmatrix}\mathit{Pre} \wedge \mathit{Pre}_{\mathit{deps}} \wedge \alpha(\mathit{Pre}) \wedge \alpha(\mathit{Pre}_{\mathit{deps}})\end{pmatrix} \Rightarrow \begin{pmatrix}\mathit{wp}(s, A([\![\mathit{oldloc}(e)]\!])) \\ \Leftrightarrow \alpha(\mathit{wp}(s, A([\![\mathit{oldloc}(e)]\!])))\end{pmatrix}\right)$$

The only way to prove the above equivalence is to show that both expressions (which represent the value of e at the end of two distinct executions) are equal. On the other hand, the $\mathit{Pre}_{\mathit{deps}}$ assumption provides the hypothesis that the locations $\overrightarrow{\mathit{deps}}$ share the same values. Note that α is a variable-renaming function (extended to terms): it always returns the same image for a given variable.

If we close the term defined as $\mathit{deps\_fo}(e, \overrightarrow{\mathit{deps}}, s, \mathit{Pre})$ in definition 2 by quantifying (universally) over A and $\overrightarrow{f}$, we can prove that both formulations are equivalent:

Theorem 1.

$$\mathit{deps\_ho}(e, \overrightarrow{\mathit{deps}}, s, \mathit{Pre}) \Leftrightarrow \mathit{deps\_fo}(e, \overrightarrow{\mathit{deps}}, s, \mathit{Pre})$$

Proof. We give only a sketch of the proof here. One of the key ingredients is the following property:

$$(A \Rightarrow B) \Rightarrow \mathit{wp}(s, A) \Rightarrow \mathit{wp}(s, B)$$

which is easily derived by induction on s and the rules for computing *wp*.

Suppose that 1 holds. Since we have both $\mathit{Pre}_{\mathit{deps}}$ and $\alpha(\mathit{Pre}_{\mathit{deps}})$, we know that $\overrightarrow{[\![\mathit{deps}]\!]} = \overrightarrow{\alpha([\![\mathit{deps}]\!])}$. Then, we can substitute $[\![\mathit{oldloc}(e)]\!]$ by $\phi(\overrightarrow{[\![\mathit{deps}]\!]})$ in 2, and the equivalence follows from the equality above.

Conversely, let us suppose now that (the universal closure of) 2 holds. There exists a function ψ such that $[\![\mathit{oldloc}(e)]\!] = \psi(\overrightarrow{[\![\mathit{reads}]\!]})$, where *reads* is the set of locations (expressed in the pre-state) which may be read during the execution of `f`. We split $\overrightarrow{[\![\mathit{reads}]\!]}$ in two parts: $\overrightarrow{[\![\mathit{deps}]\!]}$ and $\overrightarrow{\mathit{others}}$. Take arbitrary values $\alpha(\overrightarrow{\mathit{others}})$ of appropriate type[3] and define

$$\phi(\overrightarrow{[\![\mathit{deps}]\!]}) = \psi(\overrightarrow{[\![\mathit{deps}]\!]}, \alpha(\overrightarrow{\mathit{others}}))$$

We must now prove that

$$\forall \overrightarrow{[\![\mathit{deps}]\!]}, \forall \overrightarrow{\mathit{others}}, \mathit{wp}(s, \psi(\overrightarrow{[\![\mathit{deps}]\!]}, \overrightarrow{\mathit{others}}) = \phi(\overrightarrow{[\![\mathit{deps}]\!]}))$$

For that, we introduce two sets of fresh variables $\beta(\overrightarrow{[\![\mathit{deps}]\!]})$ and $\beta(\overrightarrow{\mathit{others}})$ in place of the quantifiers, and instantiate A in 2 by the following predicate:

$$\lambda x.x = \psi(\beta(\overrightarrow{[\![\mathit{deps}]\!]}), \beta(\overrightarrow{\mathit{others}}))$$

[3] we suppose that all types used by the model to represent C values are inhabited

We obtain thus

$$\begin{array}{l}\forall \overrightarrow{[\![\mathit{deps}]\!]}, \forall \overrightarrow{\mathit{others}}, \forall \alpha(\overrightarrow{[\![\mathit{deps}]\!]}), \forall \alpha(\overrightarrow{\mathit{others}}), \\ \quad \overrightarrow{[\![\mathit{deps}]\!]} = \beta(\overrightarrow{[\![\mathit{deps}]\!]}) = \alpha(\overrightarrow{[\![\mathit{deps}]\!]}) \Rightarrow \\ \quad\quad \mathit{wp}(s, [\![\mathit{oldloc}(e)]\!] = \psi(\beta(\overrightarrow{[\![\mathit{deps}]\!]}), \beta(\overrightarrow{\mathit{others}}))) \\ \quad\quad\quad \Leftrightarrow \alpha(\mathit{wp}(s, [\![\mathit{oldloc}(e)]\!] = \psi(\beta(\overrightarrow{[\![\mathit{deps}]\!]}), \beta(\overrightarrow{\mathit{others}}))))\end{array}$$

With the equality hypotheses and by instantiating $\overrightarrow{\mathit{others}}$ with $\beta(\overrightarrow{\mathit{others}})$, the left-hand side of the equivalence is true, so that the right-hand side holds. Hence, we obtain

$$\mathit{wp}(s, \psi(\alpha(\overrightarrow{[\![\mathit{deps}]\!]}), \alpha(\overrightarrow{\mathit{others}})) = \psi(\beta(\overrightarrow{[\![\mathit{deps}]\!]}), \beta(\overrightarrow{\mathit{others}}))$$

Then, with the equalities $\beta(\overrightarrow{[\![\mathit{deps}]\!]}) = \alpha(\overrightarrow{[\![\mathit{deps}]\!]})$ and the definition of ϕ, the following equality holds:

$$\mathit{wp}(s, \phi(\beta(\overrightarrow{[\![\mathit{deps}]\!]})) = \psi(\beta(\overrightarrow{[\![\mathit{deps}]\!]}), \beta(\overrightarrow{\mathit{others}})))$$

□

As an example, the verification condition generated for the dependencies of variable z in the specification of section 2.2 using a classical representation of the memory as an array of **`int`** is the following. Since we have a pointer to **`int`** and take the address of `y` and `z`, we have in the formula an array of **`int`** with two distinct offsets `y` and `z`. Note that α will preserve these offsets (on the other hand, we find of course two versions of the array mem itself).

$$\begin{array}{l}\forall \mathit{mem}, c, x, y, z, \mathit{mem}_1, c_1, x_1. \\ \quad (y \neq z \wedge c = C() \wedge x = X() \wedge \mathit{access}(\mathit{mem}, z) = Z() \\ \quad \wedge c_1 = C() \wedge x_1 = X() \wedge \mathit{access}(\mathit{mem}_1, z) = Z()) \\ \Rightarrow \\ \quad (((c \neq 0 \Rightarrow \\ \quad\quad ((c = 0 \Rightarrow A(\mathit{access}(\mathit{store}(\mathit{mem}, y, x+1), z))) \\ \quad\quad \wedge (c \neq 0 \Rightarrow A(\mathit{access}(\mathit{mem}, z))))) \\ \quad\quad \wedge (c = 0 \Rightarrow \\ \quad\quad ((c = 0 \Rightarrow A(\mathit{access}(\mathit{store}(\mathit{mem}, z, x+1), z))) \wedge \\ \quad\quad (c \neq 0 \Rightarrow A(\mathit{access}(\mathit{mem}, z)))))) \\ \quad \Leftrightarrow \\ \quad ((c_1 \neq 0 \Rightarrow \\ \quad\quad ((c_1 = 0 \Rightarrow A(\mathit{access}(\mathit{store}(\mathit{mem}_1, y, x_1+1), z))) \\ \quad\quad \wedge (c_1 \neq 0 \Rightarrow A(\mathit{access}(\mathit{mem}_1, z))))) \\ \quad\quad \wedge (c_1 = 0 \Rightarrow \\ \quad\quad ((c_1 = 0 \Rightarrow A(\mathit{access}(\mathit{store}(\mathit{mem}_1, z, x_1+1), z))) \wedge \\ \quad\quad (c_1 \neq 0 \Rightarrow A(\mathit{access}(\mathit{mem}_1, z)))))))\end{array}$$

Automatic prover Alt-Ergo easily proves this formula.

In the remainder of this section, we deal with functional dependencies in presence of function calls and loops. For a call to `f`, the adaptation rule of section 4.3 puts the post-conditions of `f`, which include functional dependencies, as hypotheses of the current goal. Similarly, in the case of a loop, the invariant, and by extension the **`loop assigns`** clauses, appear both as goals and hypotheses of proof obligations. When functional dependencies occur as goals that have to be proved, we use the slightly contrived, but first-order, formulation. When functional dependencies occur as hypotheses, we use a skolemized version of the higher-order initial formulation (definition 1). As long as care is taken always to use

the same function symbols ϕ_x for the same output variable of a same C function, this version is both powerful and easily digested by first-order automatic provers.

5.3 Call to a Function with Provided Dependencies

Adding treatment for functional dependencies to a verification conditions generator that does not already have it requires one modification that cannot be expressed in terms of the building blocks the generator can be expected already to have. The generator must have a primitive to compute the weakest precondition for a block of code for which only functional dependencies are provided.

The simplest dependencies can be translated in term of assignment. For instance, if we take the following dependencies:

```
assigns x \from y, z ; assigns z \from t ;
```

we could handle them roughly the same way the following assignments would have been:

```
old_x = x;
old_z = z;
x = Phi_x(y, old_z);
z = Phi_z(t);
```

The sequence of assignments `old_x = x;` at the beginning are useful because each of the dependencies in a function contract expresses dependencies with respect to the values of the variables before the call, even if some of these variables are also modified by the call. Indeed, in the case of the dependencies of a function that swaps the contents of two locations, like the earlier example `swap`, these preliminary assignments cannot be done without .

Some care must be taken, if someone has for some reason specified redundant functional dependencies. The dependencies component of each of the clauses must be true separately, so that a contract that contains the redundant clauses

```
assigns x \from y, z ;
assigns x \from z, t ;
```

is effectively equivalent to the contract:

```
assigns x \from z ;
```

Variable `x`'s new value cannot depend on `y` because the second clause says it doesn't, and it cannot depend on `t` because the first clause says it doesn't.

In presence of pointers and aliasing, the main difficulty is that sometimes, the latter case of redundant clauses occurs in less obvious ways. For instance, if `p` is not prevented to point to `x` by a precondition, the dependencies below mean that when it does, `x`'s contents depend only from `z` after the call.

```
assigns *p \from y, z ;
assigns x \from z, t ;
```

The above dependencies can easily wrongly be proposed for a function that modifies `x` and `*p`, but at least attention will be drawn to the aliasing corner cases when this function fails verification.

Because different verification conditions generators may use different memory models, there is no universal formula for handling a function call with dependencies. We assume that the generator provides a predicate transformer $\mathcal{F}_{n,d}$ that transforms a predicate that has to hold after some unknown code C into the predicate that has to hold before C, with only the information that C has functional dependencies d, using the skolemized version of the high-order expression of d. The name n should be used as a prefix for the names of the skolemized functions, in a way that two calls to the same C function use identical skolem functions, but two calls to different C functions that happen to have the same dependencies don't. The function $\mathcal{F}_{n,d}$ is an additional requirement for the API of section 4.2.

Often, verifying the functional dependencies of a caller `g` is possible with only functional dependencies as specifications for the callee `f`. This gives rise to proof obligations of the form $C[\phi_x(z)] \Leftrightarrow C[\phi_x(z)]$ alluded to earlier, where it is vital that two calls to the same function introduce the same logic symbols.

5.4 Functional Dependencies and Loops

The functional dependencies d of a loop `while(c) b;` are verified in two steps, establishment and invariance. Verifying the establishment means verifying that 0 iterations of the loop body satisfy the dependencies. Assuming that an invariant I has already been established for the loop, one clause **`loop assigns`** e **`\from`** $\overrightarrow{deps}$ among those of d is translated as proof obligation $\mathit{deps\_fo}(e, \overrightarrow{deps}, \texttt{skip}, I)$. In general, it simply means that e should always occur in its corresponding *deps*, because loop functional dependencies are supposed to hold for any number of iterations, including zero.

Loop dependencies are part of the loop invariant. Thus, they are proved in the same induction. We assume that an arbitrary number of iterations have modified the variables, according to the provided loop dependencies and to I (induction hypothesis). We reduce the establishment of one of the clauses of d, **`loop assigns`** e **`\from`** $\overrightarrow{deps}$, to the formula below, expressed just before the loop:

$$\mathcal{F}_{induction,d}(\mathit{deps\_fo}(e, \overrightarrow{deps}, \texttt{if(c) b;}, I)))$$

In this formula, $\mathit{deps\_fo}(e, \overrightarrow{deps}, \texttt{if(c) b;}, I)$ is the goal, the first-order formula that expresses the proper dependencies, expressed at the point at the beginning of the n+1-th iteration. Applying $\mathcal{F}_{induction,d}$ to that term means applying the induction hypothesis to the first n loop iterations to obtain a formula to be verified before the loop.

Note that in practice, the loop dependencies often constitute an inductive loop invariant, so that *true* can be used for I.

6 Related Work and Conclusion

6.1 Adaptation Rule

Various proof rules have been proposed for function or procedure calls since Hoare's foundational paper [15], with different restrictions on the kind of calls which are supported (reference passing, aliasing between out parameters and global variables,...). In particular, issues raised by aliasing (see section 5.3) have been tackled by the notion of simultaneous assignment of Cartwright and Oppen [6] or the multiple assignments of Gries and Levin [12]. These rules are however meant for traditional post-conditions, not functional dependencies as they are described here.

6.2 Secure Information Flow

Establishing that a program has the *secure information flow* property is a classical problem in security analysis. Informally speaking, the variables of the program are split in "low-security" and "high-security", and the desired property is that knowing the value of the low-security variables does not give any information on the values of the high-security ones (while of course high-security variables can depend on both low and high security variables). This property bears some similarity with functional dependencies. Roughly speaking, saying that function `f` has secure information flow property amounts to give `f` the following specification:

```
assigns low_security \from low_security;
assigns high_security
      \from high_security, low_security;
```

Joshi and Leino have presented in [18] a characterization of secure information flow property of program `S` in terms of semantical equivalence of the two programs `S; HH` and `HH; S; HH`, where `HH` is a program that puts arbitrary values in high-security variables. Informally, the first program executes `S` and discards the values of high-security variables, while the second does the same thing but starts in a state where high-security variable have arbitrary values. For both to be equivalent the low-security variables must not depend on the high-security ones or in other words, `S` has secure information flow. The context set up by Joshi and Leino is a bit different to the one of this paper, as they consider non-deterministic programs, and do not appear to deal with aliases, but it appears that most of their work could be adapted to functional dependencies, by splitting `HH` in two: `HNF`, which puts arbitrary values in all locations that are **`\separated`** from the **`\from`** and `HNA` which puts arbitrary values in locations that are **`\separated`** from the **`assigns`** of the program. The characterization of functional dependencies would then become the equivalence between `HNF; S; HNA` and `S; HNA`. In the case of deterministic programs, Joshi and Leino show that secure information flow amounts to prove that there exists a function f mapping the initial values of low security variables to their final values. This result is analogous to the higher-order presentation of definition 1. However, they do not provide a first-order counterpart similar to definition 2.

6.3 Comparison to JML `assignable` *and* `accessible` *clauses*

JML has two kind of clauses related to functional dependencies. First, `assignable` clauses specifies locations that may be modified by a method. Second, `accessible` clauses (which the JML reference manual [19] calls "a seldom-used feature") for specifies the locations that the method may read from. In the process of converting a C ACSL-annotated program into a JML-annotated equivalent Java program, it would therefore be possible, with some loss of information, to convert

```
/*@ assigns loc1 \from dep1a, dep1b ;
    assigns loc2 \from dep2a, dep2b ;
 */
```

into:

```
/*@ assignable loc1, loc2
    accessible
      dep1a, dep1b, dep2a, dep2b
 */
```

Some information is lost in this translation: the individual dependencies of each output variable do not appear in the JML contract. Our contribution is not only to suggest the finer-grained dependencies, though, which are an obvious refinement once one is convinced of the utility of these dependencies.

We argue that the dependencies are as useful as the list of modified locations, because they can be used for a more precise treatment of a function call. To quote again from the JML reference manual, "an assignable clause gives a frame axiom for a specification. It says that, from the client's point of view, only the locations named, and locations in the data groups associated with these locations, can be assigned to during the execution of the method". The intent of the ACSL assigns clause is similar, but when the specifier goes to the length of providing dependencies, the call rule becomes even more precise. Some properties may be proved without need for further specification of the called function. Like the properties that become provable when introducing `assignable` clauses, they are not the properties that are strongly related to what the function does, but the properties that are independent from what the function does, and for which an underspecified function call only seems to interfere when it does in fact not.

6.4 SAL lightweight annotations

Functional dependencies have similarities with the SAL annotation language [13]. There are, however, differences in intent and in detail. A common idea of both approaches is that weaker formal properties that describe only a well-identified

aspect of a computation, while less interesting than complete formal specifications, are also easier to obtain from specifiers and to verify automatically.

SAL annotations have been used for finding bugs in existing code. Annotations were inferred from the code of pre-existing functions. In the SAL context the annotations do not need to be in a specific concrete representation among the set of all correct representations stemming from pointer aliasing relations. In our context the development process is a strict V development cycle and specifications are written before the code, during the descending phase of the V cycle. In the ascending verification phase, neither the code nor the specification are supposed to be adapted to suit the verification tool. For one exceptional case, this development process may allow to document that some inferred dependencies for a function are different from the pre-existing dependencies that had been specified because the inferred and the pre-existing dependencies are both correct characterization of the function, differing only because of aliasing. However, a method that would require this argumentation too often would simply not be suitable to this development process. To be suitable for these development process constraints, the proposed method has to allow verifying pre-existing specifications against pre-existing code without requiring to adapt either.

In detail, SAL annotations are syntactically lightweight because they are written near the formal argument they apply to: annotations reviews are therefore more straightforward. The drawback of this approach is that global variables accessed by the function are not annotated. This compromise makes special sense for library functions that are not supposed to have an internal state. The elementary annotations characterize pointer arguments as pointing to input or/and output buffers, specify if each of them can be NULL, and distinguish pointers to fixed-length buffers from pointers to zero-terminated strings.

The information contained in SAL annotations is also typically part of the description written in advance for a function near the end of the conception of critical systems. There is some overlap with the information in functional dependencies: inputs buffers can be recognized in that they appear on the right-hand side of a dependency, and output buffers on the left-hand side. With functional dependencies, nullable pointers cannot be expressed. Dependencies of functions manipulating zero-terminated strings can be specified following the work of [25]. On the other hand, SAL annotations do not contain dependency information. Besides, contrary to SAL annotations, functional dependencies allow and require to list *all* memory locations modified by the function. This property is useful for embedded systems where functions modify a global state, and the knowledge that inputs and outputs are listed exhaustively is important for reasoning about the function in the context of a call. Therefore SAL annotations do not apply on embedded systems requiring such a property.

7 Conclusion

Our contribution is a first-order method for verifying that a function satisfies provided functional dependencies (or at least the part of functional dependencies that was not already addressed within Frama-C). We have shown that in a language with widespread aliasing, the problem was not simply one of computing dependencies by a data-flow analysis and comparing the computed dependencies to the provided ones. We think that the verification of functional dependencies in presence of significant aliasing requires weakest precondition techniques. We gave a first-order formulation of a post-condition that, if verified, guarantees that the function satisfies the functional dependencies. Being first-order, existing automatic provers specialized in program verification (Simplify, Z3, Alt-Ergo,...) can handle this formulation. Because it corresponds to an abstract, approximative view of what the function does, it can often be proved automatically. Functional dependencies are a compact and very accessible way to express high-level formal specifications. Providing tools for helping their verification is therefore a key path for widening the use of formal methods.

8 Acknowledgments

The authors wish to thank Philippe Herrmann for pointing them at the Joshi and Leino paper and other suggestions on an draft version of this paper, as well as the anonymous referees for their helpful comments. This work has been partially supported by the ANR U3CAT project.

References

1. Barnett, M., Chang, B.Y., DeLine, R., Jacobs, B., Leino, K.: Boogie: A modular reusable verifier for object-oriented programs. In: Formal Methods for Components and Objects, LNCS, vol. 4111, pp. 364–387. Springer (2005)
2. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language Preliminary design (2008), `http://frama-c.com/downloads/acsl_1.4.pdf`
3. Bornat, R.: Proving Pointer Programs in Hoare Logic. In: Mathematics of Program Constructions. Lecture Notes in Computer Science, vol. 1837, pp. 102–126. Springer (2000)
4. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfers 7(3), 212–232 (2005)
5. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Footprint Analysis: A Shape Analysis that Discovers Preconditions. In: Static Analysis Symposium (SAS). LNCS, vol. 4634 (2007)
6. Cartwright, R., Oppen, D.: Unrestricted Procedure Calls in Hoare's Logic. In: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL). pp. 131–140 (1978)

7. CENELEC: CENELEC 50128 - Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems (2001)
8. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (1999)
9. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL). pp. 269–282 (1979)
10. Cuoq, P., Prevosto, V.: Frama-C's Value Analysis Manual, `http://frama-c.com/download/frama-c-value-analysis.pdf`
11. Frama-C home page, `http://frama-c.com/`
12. Gries, D., Levin, G.: Assignment and Procedure Call Proof Rules. ACM TOPLAS 2(4), 564–579 (1980)
13. Hackett, B., Das, M., Wang, D., Yang, Z.: Modular checking for buffer overflows in the large. In: ICSE '06: Proceedings of the 28th international conference on Software engineering. pp. 232–241. ACM, New York, NY, USA (2006)
14. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580 and 583 (1969)
15. Hoare, C.A.R.: Procedures and Parameters: an Axiomatic Approach. In: Symposium on Semantics of Algorithmic Languages. Lecture Notes in Mathematics, vol. 188, pp. 102–116. Springer Berlin (1971)
16. Hoare, C.A.R.: Proof of correctness of data representations. Acta Informatica 1(4), 271–281 (1972)
17. Hubert, T., Marché, C.: Separation analysis for deductive verification. In: Heap Analysis and Verification (HAV'07). pp. 81–93. Braga, Portugal (Mar 2007)
18. Joshi, R., Leino, K.R.M.: A semantic approach to secure information flow. Science of Computer Programming 37(1-3), 113–138 (May 2000)
19. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M.: JML Reference Manual (draft) (2009)
20. Marché, C., Paulin-Mohring, C.: Reasoning about Java programs with aliasing and frame conditions. In: 18th International Conference on Theorem Proving in Higher Order Logics. Lecture Notes in Computer Science, vol. 3603, pp. 179–194 (2005)
21. Mauborgne, L., Rival, X.: Trace Partitioning in Abstract Interpretation Based Static Analyzers. In: Sagiv, M. (ed.) European Symposium on Programming (ESOP'05). Lecture Notes in Computer Science, vol. 3444, pp. 5–20. Springer-Verlag (2005)
22. Meyer, B.: Object-oriented Software Construction. Prentice Hall (1997)
23. Moy, Y.: Union and cast in deductive verification. In: Proceedings of the C/C++ Verification Workshop. vol. Technical Report ICIS-R07015, pp. 1–16. Radboud University Nijmegen (Jul 2007)
24. Moy, Y.: Automatic Modular Static Safety Checking for C Programs. Ph.D. thesis, Université Paris Sud (2009)
25. Moy, Y., Marché, C.: Inferring local (non-)aliasing and strings for memory safety. In: Heap Analysis and Verification (HAV'07). pp. 35–51. Braga, Portugal (Mar 2007)
26. Praxis High Integrity Systems: SPARK95 - The SPADE Ada 95 Kernel (Including RavenSPARK), 4.8 edn. (2008), `http://www.altran-praxis.com/downloads/SPARK/technicalReferences/SPARK95_RavenSPARK.pdf`
27. Randimbivololona, F., Souyris, J., Baudin, P., Pacalet, A., Raguideau, J., Schoen, D.: Applying formal proof techniques to avionics software: a pragmatic approach. In: World Congress on Formal Methods. Lecture Notes in Computer Science, vol. 1709, pp. 1798–1815. Springer (1999)
28. Reynolds, J.C.: Separation Logic: A Logic for Shared Mutable Data Structures. In: LICS. pp. 55–74. IEEE Computer Society (2002)
29. RTCA and EUROCAE: DO-178B - Software Considerations in Airborne Systems and Equipment Certification (1992)
30. WG14: ISO C Standard 1999. Tech. rep., ISO (1999), `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf`, ISO/IEC 9899:1999 draft
31. Why home page, `http://why.lri.fr/`