Framing the Frame Problem

Bertrand Meyer
ETH Zurich, Innopolis University & Eiffel Software
http://se.inf.ethz.ch — http://www.eiffel.com

Abstract. Some parts of the software verification process require human annotation, but as much as possible of the rest should be automatic. An excellent candidate for full automation is change analysis, also known as the frame problem: how to determine which program properties remain unchanged under a given operation. The problem is particularly delicate in the case of programs using pointers or references, in particular object-oriented programs. The *double frame inference* strategy automates both frame specification and frame verification. On the specification side, it deduces the set of permissible changes of a routine (its "modifies clause") from a simple examination of its postcondition. On the implementation side, it applies the *change calculus*, itself based on the *alias calculus*, to determine the set of expressions whose values the routine can actually change. Frame verification then consists of ascertaining that the actual change set is a subset of the permissible change set.

Keywords. Software engineering, software verification, program specification, program verification, program proving, frame analysis, change analysis, alias analysis, alias calculus, frame calculus, frame inference, Design by Contract, Eiffel, object-oriented programming, pointers, references.

1 Overview of the problem and the solution

Software verification ascertains what programs do. It must also ascertain what they do not.

The "frame problem" covers the second of these obligations. McCarthy and Hayes explained it back in 1969 [3]:

> *In proving that one person could get into conversation with another, we were obliged to add the hypothesis that if a person has a telephone he still has it after looking up a number in the telephone book. If we had a number of actions to be performed in sequence we would have quite a number of conditions to write down that certain actions do not change the values of certain queries. In fact with n actions and m queries we might have to write down mn such conditions.*

With the exception of one word (I took the liberty of substituting "query" for McCarthy's "fluent", a term that has not stuck), this description is just as current today; so is the problem it describes. Software specifications and the corresponding verification techniques usually focus on how program operations change their environment: withdrawing a hundred francs decreases your balance by that amount. But we are also interested in what they leave unchanged: a withdrawal should not make someone else become your account's owner. Such a requirement that an action must leave a query's value unchanged is called a "frame property". Even if the specification of a program element restricts itself to non-frame properties, the verification process will often need to rely on frame properties.

A solution to the frame problem should cover two aspects:

- Specification: expressing the desired frame properties without falling into the mn trap described by McCarthy and Hayes. Some language conventions are necessary to ensure that the programmer or specifier can concentrate on the interesting parts — specifying how things change — and let the rest (frame properties) follow as a consequence.

- Verification: proving that actions do not change the queries specified in frame properties. In principle this task could use the same techniques as for non-frame properties; but the specific nature of frame properties, all of the form $query = \textbf{old}\ query$, may call for simpler solutions.

The rest of this discussion addresses these two issues in the context of an object-oriented programming language. Here is a summary of the results.

The solution is to use automation on both sides, hence the term **double frame inference**, then to compare the results.

The first component is a specification convention: avoid the mn problem by understanding every routine postcondition to be complemented by clauses of the form $query = \textbf{old}\ query$ for every $query$ not explicitly cited in the postcondition. In other words, if a command will change a query, it has to say something about that query in its specification. The specification does not have to say *how* the query will change; it simply names it. Unmentioned, unchanged. For a program element p, we may call **inferred frame specification**, written $\overline{p}$, the result of applying this convention: the set of queries which p, according to the specification, is permitted to change.

The second component is a technique for inferring from the program text which queries a command can *actually* change. The mechanism is the *change calculus*, a set of rules specifying, for instructions p of all possible kinds in a programming language, the set of queries that p may affect, called the **inferred frame implementation** and written $\underline{p}$. An example rule, slightly simplified, is $\underline{\textbf{if}\ c\ \textbf{then}\ p\ \textbf{else}\ q\ \textbf{end}} = \underline{p} \cup \underline{q}$: executing a conditional can change whatever either branch can change. (The full form of this rule is given in section 5 and involves an extra parameter, the alias relation.)

The combination of these two techniques yields a specification of what is permitted to change and of what may actually change. The third step, the easiest, is verification: ascertain that every routine p satisfies the **frame condition** $\underline{p} \subseteq \overline{p}$.

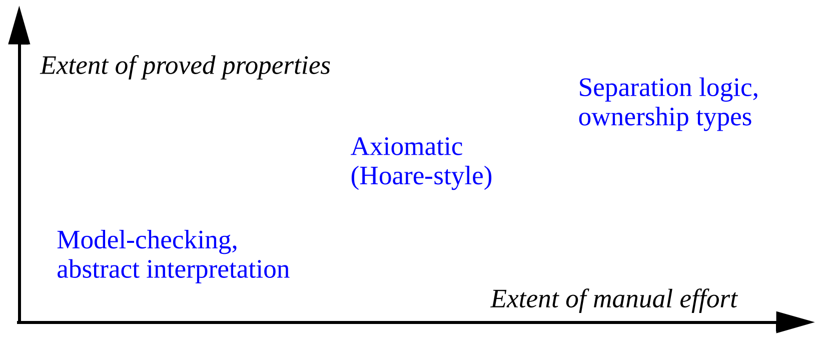
Both $\underline{p}$ and $\overline{p}$ may be over-approximations. On the implementation side, it is impossible to guarantee that $\underline{p}$ is the exact list of changed queries, both for theoretical reasons of undecidability and pragmatically, because we are relying on simple rules supporting efficient implementation. According to the change calculus, for example, $y := x\ ;\ x := y$ changes x (as well as y) even though x in fact regains its original value. On the specification side, it is not a bug to be over-cautious and write a postcondition that leaves open the possibility that the routine could change a query, even if verification does not need that property. In addition, we may have to extend $\overline{p}$ to ensure the frame condition in case of over-approximation in the implementation; in the example, we might have to list x in the postcondition.

The most difficult part of the process is the frame implementation inference (the computation of $\underline{p}$). More precisely it is difficult in the presence of pointer (reference) operations, as in object-oriented programming languages but also, for example, in Pascal or C. Pointers imply in particular that one can no longer assume the seemingly obvious property that an assignment $x := y$ can only change x and expressions involving x: if this assignment occurs after $z := x$, then it also affects any expression of the form $z \bullet a$. The reason is that before an assignment some expressions, such as z in this example, may be aliased to the current object. Frame analysis for an object-oriented language must as a consequence include *alias* analysis.

Usual approaches to the frame problem — involving, for example, separation logic or ownership types — require a manual specification effort, and annotations on the implementation as well. The contribution of this article is a comprehensive method of frame analysis, automatic on both the specification and implementation sides thanks to the technique of double frame inference. The benefit is to free software developers from the tedious task of specifying and verifying which properties should not change, allowing them instead to concentrate on the interesting part: which properties do change, and how.

2 The automation tradeoff

To understand the rationale for the present approach it is useful to take a global look at what various approaches to verification demand and yield: work required (from the specifier, programmer and proving engineer) versus extent of verification results. The following figure suggests the tradeoffs involved.


Extent of proved properties
Separation logic, ownership types
Axiomatic (Hoare-style)
Model-checking, abstract interpretation
Extent of manual effort

Although the classification is rough and the distances subjective, the general idea should be non-controversial. The most successful approaches either:

- Like axiomatic semantics, require a fair amount of manual effort (writing contracts and other verification conditions), but can yield verification of full functional correctness.
- Like model-checking, require only modest annotation effort, and hence are almost entirely automated, but only yield the specification of particular (although often critical) safety or liveness properties.

It is unrealistic to hope for the top-left area, and approaches at the bottom-right would be of no interest. Any verification technique, and any improvement to existing technique, should consider the tradeoff between manual effort and verification benefit. To make verification an integral part of the software development process, we must carefully weigh the amount of annotations that we expect programmers to contribute:

- It is legitimate to require that programmers express properties — such as contracts specifying routine preconditions, routine postconditions and class invariants — that describe the functional behavior expected of programs elements. Without such a specification there can be no functional verification, since one would not know what to verify.
- Everything else, in particular intermediate properties that are necessary for the proof but may seem to the programmer too detailed and internal, should as much as possible be inferred automatically.

Frame properties (like loop invariants, which however are still hard to synthesize automatically) belong to the second category. In spite of their intellectual elegance, approaches to framing relying on separation logic or type ownership have the disadvantage of requiring extensive annotations (heap assertions or type annotations), moving them to the right of the figure, while serving internal needs of Hoare-style verification and hence bringing little visible upward lift. Frame properties do not deserve imposing such a manual effort; having to specify them explicitly is a nuisance. The present work strives for an implicit and automatic approach to frame verification.

3 Frame specification inference: a convention

Three conventions are possible for specifying frame properties: manual, exclusive and implicit.

The **manual** convention does not rely on any specific technique for frame properties, but treats them like any other specification element. For example:

```
class ACCOUNT feature
    holder: PERSON
    number: STRING
    branch: BRANCH
    balance: INTEGER
    deposit (amount: INTEGER)
                -- Add amount to the account's balance.
        require
            amount > 0
        do
            …Implementation …
        ensure
            balance = old balance + amount
            -- The following clauses are pure frame properties:
            holder = old holder
            number = old holder
            branch = old branch
        end
    …Other features …
end
```

The annotations illustrate the concepts of the preceding discussions: only the first postcondition clause (the one on *balance*) is interesting for the programmer, since it expresses the routine's functional goal; but the others are boring frame properties. With m commands such as *deposit* and n queries such as *holder* the manual convention raises the mn problem. The problem is not just, however, the sheer number of boring properties to specify. Software engineering must account for changes; the class author or maintainer may add new queries, such as *iban* (international account number). If, as is usually the case, existing commands do not affect the new queries, they must still be updated, one by one, with postcondition clauses of the form *iban* = **old** *iban*. Not only is the manual convention tedious, it leads to instability in the software process.

The **exclusive** convention relies on the observation that the *mn* matrix is usually sparse: most commands affect only a few queries. It directs the programmer or specifier to give for each command an exhaustive list of the queries that the command may affect. With the exclusive convention the postcondition of *deposit* will read, in full:

```
ensure
    balance = old balance + amount
    only balance
```

The syntax here uses the keyword **only**, which was at one point proposed for Eiffel (until the implicit convention, studied next, displaced it). Other common keywords are **modifies** and (in JML) **assignable**; rather than appearing in the postcondition, as with **only**, such clauses can be separate elements of the specification. Regardless of the details, "modifies clause" is the accepted name for clauses of this kind. The semantics is that a modifies clause "**only** *a, b, c*" denotes a set of clauses *q* = **old** *q* for all queries *q* other than *a*, *b* and *c*. Note that in spite of its name a modifies clause does not *require* a command to modify the queries *listed*: instead, it *forbids* it from modifying the queries *not listed*. This is the reason for preferring the keyword **only**.

The exclusive convention is so named because it defines the unaffected queries by exclusion. As a side benefit, it also makes it easy to specify that a command is pure (has no side effects): just write an empty modifies clause. With appropriate precautions [2], the exclusive convention can be made to work with inheritance.

The **implicit** convention derives from the exclusive convention, but removes explicit frame specifications. It was initially suggested by an informal review performed on publicly available JML code, which revealed that in practice queries mentioned in a modifies ("assignable") clause for a command also appear in the postcondition of that command. In other words, it seems that whenever JML programmers state that something can be modified they also say *how* it will be modified. They do no necessarily say it in exact terms, as in *q* = *some_value*, but may just state *some_property* (*q*). Either way, however, the postcondition names *q*. It then seems a waste of effort to require writing a special clause listing such queries. If the empirical observation has exceptions, it is easy to correct a frame property omission by inserting into the postcondition a clause *relevant* (*q*) where *relevant* is a boolean function that always returns True but brings its argument, here *q*, into the picture.

The analysis of the postcondition should only consider query occurrences outside of an **old** clause: a clause *q1* = *some_function* (**old** *q2*) indicates that *q1* can be modified, but says nothing about *q2*.

The implicit convention yields the notion of frame specification inference: instead of requiring programmers to write modifies clauses, the verification tools infer them from the postcondition. This is only one kind of "frame inference", not to be confused with the other one, *frame implementation inference*, discussed in section 5.

The implicit convention reflects the observation that when you realize that something changes you will generally want to say something about how it changes. Of the three possible conventions, this is the simplest one for programmers. The present work relies on it, as the first part of its double frame inference method.

The rest of the discussion is, however, not dependent on this choice of convention. It only needs to assume the existence of a frame specification. With the manual convention, that specification follows from explicit *q* = **old** *q* clauses; in the exclusive approach it follows from modifies clauses; and in the implicit convention it obtains a list of unmodified queries from a straightforward parsing of the postconditions.

4 The nature of frame specifications

Defining frame properties precisely requires raises three delicate questions, requiring clear choices.

The first question is how to handle inheritance and polymorphism. Although the framework developed here can be extended to inheritance, the present article does not cover that aspect. For a discussion of the problem, see Leino *et al*. [2].

The second question is what exactly we should consider a "change" to a query. In particular, if a routine initially changes the value of a query, but restores it later during its execution, should we consider that it changes it? In the most trivial case, we may ask whether *x* := *x* changes *x*. There are arguments for either answer, but in the present work we will answer yes: as soon as a routine can set the value of a query, it changes it. (We may think in the same way as, in hardware, the designer of a solid-state drive memory: since each cell in an SSD can only be written a finite number of times before it goes bad, every write operation is significant even if it rewrites the cell with its previous value.) This convention can be surprising since the following scheme is common, for example in cursor-based data structures of the EiffelBase library:

- Save a certain state property, such as the cursor position in a list.
- Perform operations that change that property, for example by moving the cursor to some other part of the list.
- Restore the value, here by bringing the cursor to its original position.

In such cases the header comment typically states "Do not change cursor position". What this means in practice is that the postcondition should include a clause *index* = **old** *index*; in line with the "implicit" convention adopted in the preceding section, *index* (the cursor position), being mentioned in the postcondition, does not fall under the implicit frame specifications.

The third and final question is what kind of properties (called "queries" up to now) we should list in modifies clauses, whether explicit or implicit. On this side too three possibilities are available; we may call them A for attributes, Q for queries and E for expressions.

Much of the existing work on frame inference only considers changes that commands can make to *attributes* (object fields). The reason for this approach (convention A) is that changes to attributes are easy to deduce from the code. But in object-oriented programming we should apply the Uniform Access principle [4], which states that a query should be freely implementable as an attribute or a function, and that this implementation choice does not affect the essential semantics of a class and its use by clients. For example, in *a*.*balance* for an account *a*, *balance* could be a function (computing the balance) or an attribute (looking up the balance in the account object).

Convention Q, more in line with object-oriented principles, considers that a class is an implementation of an abstract data type, with some commands and some queries; a command changes an object, a query returns information on the object. Then frame properties are the effect of commands on queries, regardless of each query's implementation as an attribute or a function. Queries, however, can have arguments; just filling in the *mn* matrix, which indicates which command affects which query, lumps together (for example) all array elements, since command *put*, which changes an array element, modifies query *item*, which returns an array element, but a specification at that level implies that any element change potentially affects all elements. In such an example there may be no better way to specify frame properties, but for other cases the specification will be too coarse-grained.

Convention E more closely reflects how verification can use frame properties. In practice, programs work with expressions, including "path expressions" such as *first_element*.*right*.*right* (giving the third element of a list). This approach evaluates the effect of commands on expressions actually appearing in the program and its contracts, plus any others that might be needed in the frame computation. It may be called a "mercenary" approach in the sense that it does not try to compute the whole set of frame properties but restricts itself to properties of interest for the verification. An argument for this restriction is that for object-oriented programs the frame set is often infinite, as illustrated by this example:

```
from a := l until a = Void or else some_condition loop a := l.right end
if a /= Void then a.set (some_value) end
```

If *l* is attached to the first element of a list, *right* to the successor of every element, and *set* changes the contents of its target object, then depending on *some_condition* this code may change no object at all (if the list is not empty and its first element satisfies the condition) or any of the objects attached to *l*, *l*.*right*, *l*.*right*.*right*, and so on. If, on the other hand, we limit ourselves to expressions appearing in the program, we do not have to deal with infinity.

The rest of the discussion uses choice E.

5 Frame implementation inference

Frame implementation inference relies on the frame calculus, first presented in an earlier paper [1] but appearing here with corrections and improvements.

A general observation is that when a change set includes an expression *e* it also includes *e*.*f* for any expression *f*. For example any instruction that changes *l* also changes *l*.*right*, *l*.*right*.*right* and so on. The notation *e*.$*$ will be used (in informal discussions only) to denote all possible successors of *e*, including *e* itself.

From the discussion of the previous two sections we may assume that for every instruction *p* we have a specified change set $\overline{p}$. Since we have chosen the implicit convention, $\overline{p}$ will be obtained through analysis of the postcondition, but it could derive from another convention, such as "modifies" clauses, without affecting the rest of the discussion.

Let p_C be the actual change set of *p*. Frame verification consists of assessing that the following property holds:

$p_C \subseteq \overline{p}$ -- Theoretical frame condition

It is not possible, however, to define a calculus that would yield p_C exactly, since the frame set, like with many interesting program properties, is undecidable. Instead we can compute $\underline{p}$, an approximation of p_C. Soundness means that it should be an over-approximation; in other words $p_C \subseteq \underline{p}$. As a consequence, we may settle for the following variant of the frame condition:

```
p ⊆ p̄          -- Practical frame condition
```

The approximated frame set $\underline{p}$ is a property of the program state s. In general, a property w of the state can be of any of three possible kinds. Let $w\ (ST)$ the application of w to a set of states ST, and $p\ (ST)$ the set of states resulting from applying instruction p to any state in ST. (In many cases we are interested in applying w and p to a single state, yielding a single state, but programs can be non-deterministic, or undefined, so it is better to generalize p and w to relations by using their images.) Then, depending on how one can determine $w' = w\ (p\ (ST))$, w can be:

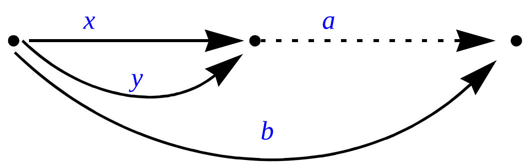
- "Syntactic", meaning that w' can be computed on the sole basis of p. This case only applies to simple properties such as the set of variables used by a program element.
- "Compositional", if w' can be computed from p and $w\ (ST)$.
- Non-compositional otherwise, meaning that w' requires more information about the state than given by $w\ (ST)$.

The change set $\underline{p}$ is syntactic only for a very simple language without pointers (where it suffices to list the targets of assignments), but it is also not compositional, as illustrated by the example of

```
x := y ; x.set_a (b)     -- set_a sets the a field
```

whose change set includes not only $x \bullet a \bullet *$ but also $y \bullet a \bullet *$, a property that cannot be deduced from the change sets of the two constituent expressions since neither them includes y.

The issue here is the close connection between framing and aliasing. The reason why y must appear in the change set after $x \bullet set\_a\ (b)$ is that before this instruction y is aliased to x, so any operation affecting successors of x also affects successors of y. The instruction $x \bullet set\_a\ (b)$, is such an operation, reattaching a as shown by the dotted line:



The combination of aliasing and framing is, on the other hand, compositional. In other words, we may work with pairs containing each a set c of expressions changed so far (possibly over-approximated) and an alias relation r. Then $[c, r]\ p$ will denote the corresponding pair (or an over-approximation) holding after the execution of p.

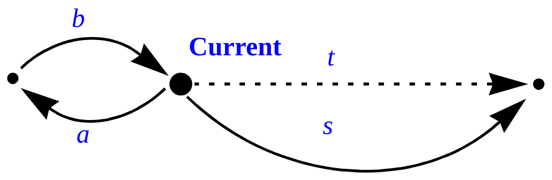
Compositionality means that it is possible to compute this resulting pair from c, r and p. The notation $\underline{p}$ used so far for the change set in the inferred frame implementation simply denotes the first projection of $[c, r]\ p$. At the program's starting point, c and r are empty.

An "alias relation" [5] is, at an arbitrary program position, the set of pairs of reference expressions that might (with possible over-approximation) become aliased to each other, that is to say, be attached to the same object. Aliasing is compositional just by itself, and reference [5] has introduced the alias calculus, which computes the alias relation. The rules will not be repeated here; the frame calculus only specifies the change set, as given below, followed by an explanation of the notations and of the practical meaning of each rule. Only the first rule (assignment) actually makes use of the relation r, but in a critical way.

$[c, r]\ (t := s)$	$= r\ (\textbf{Current})\bullet t$
$[c, r]\ (p\ ;\ q)$	$= ([c, r]\ p)\ q$
$[c, r]$ **then** p **else** q **end**	$= [c, r]\ p \cup [c, r]\ q$
$[c, r]$ **loop** p **else** q **end**	$= [c, r]\ p \cup [c, r]\ p^2 \cup \ldots$
$[c, r]$ **call** $r\ (l\,)$	$= ([c, r]\ \lvert r\rvert)\ [r^\bullet : l\,]$
$[c, r]$ **call** $x\bullet r\ (l\,)$	$= x\bullet([x'\bullet c, x'\bullet r]$ **call** $r\ (x'\bullet l\,))$

In addition, a meta-rule, **dot completeness**, states that whenever the change set as determined by these rules includes e, it also includes $e\bullet f$ for any f.

In the assignment rule, **Current** is the current object ("this" in some object-oriented languages), and $r\ (x)$ is the set of aliases of x, including x itself. A naïve version of the rule, correct in the absence of pointers, would list only t (and its successors per dot completeness) as being changed. But if the value of e is a reference to the current object, then the assignment will, in addition to t, affect $e\bullet t$, as in the figure below for $e = a\bullet b$.



In this rule and the last two (for calls), the dot symbol of object-oriented programming, "$\bullet$", is used as a distributive operator when applied to sets, lists and relations; for example $x\bullet[a, b]$ is $[x\bullet a, x\bullet b]$.

In programming languages, a conditional includes a condition, with a syntax such as **if** c **then** p **else** q **end**; the conditional rule ignores the **if** c part since the condition (assumed to have no side effect) does not affect the change set. In essence, it treats the conditional as non-deterministic choice. The loop rule similarly ignores the loop exit condition. The same conventions are used by the rules of the alias calculus [5]. They may introduce imprecision, for example in the case of a conditional whose condition always evaluates to True, but does not imperil soundness since the approach accepts possible over-approximations of p_c.

In the loop rule, p^n denotes n successive executions of p.

In the unqualified call rule (for **call** $x \bullet r\ (l)$), $r^\bullet$ is the list of formal arguments of r and the notation $[r^\bullet : l]$ denotes substitution of actuals arguments, given by the list l, for the corresponding formals.

The last rule, for qualified calls **call** $x \bullet r\ (l)$, relies on the preceding rule, transposed to the context of the target object x. To transpose the result back, it uses the "negative variable" technique [6] [5] for reasoning about properties of object-oriented programs: x', the "negation" of x, is a (fictitious) back-pointer to the calling object, with the property that $x \bullet x'$ = **Current**.

The calculus gives, for any particular program, a set of fixpoint equations that can be solved iteratively. Although there is no guarantee of finite termination, it is possible to stop after a certain number of iterations by adding some imprecision for expressions beyond a certain length.

In practice, an experiment reported in [1] showed that for a data structure library that includes explicit modifies clauses, and hence does not require frame inference on the specification side, the frame implementation inference yielded a high-quality set of clauses. It succeeded in uncovering a few critical frame conditions that the frame specification had missed, even though the library, explicitly designed for verification, had been written very carefully. The experiment also illustrated some of the limitations discussed next.

6 Limitations

The present work suffers from a number of limitations.

The frame calculus has not been proved sound. For the underlying alias calculus, a Coq-based proof has been performed [1] for the basic rules, in particular assignment, but not for the entire calculus.

As noted, the present discussion ignores inheritance. Handling inheritance is possible in the given framework, but requires extensions that we have not yet added.

Another set of limitations comes from the reliance on the alias calculus, which requires some over-approximations to avoid infinite relations.

As the experiment reported in [1] showed, a full-scale deployment of frame implementation inference still faces a number of obstacles. They are engineering issues rather than fundamental conceptual questions, but will require significant effort. They include in particular the need to build automatically the correspondence between abstract theory-based queries, as used in postconditions to obtain full specifications in modern Design by Contract [7], and attributes, as used in the code.

A first implementation is available, covering frame implementation inference, but it requires considerable extra work to become applicable to ordinary programs on a routine basis.

Acknowledgments

Alexander Kogtenkov wrote the first implementation of the change calculus [1] and provided extensive feedback in many discussions. The work on the alias calculus was performed with him and with Sergey Velder. At a presentation at the IFIP working group WG2.15 I received important feedback from Jean-Christophe Filiâtre, Andrei Paskevitch (who spotted a significant error in an initial version of one of the rules) and Andreas Podelski. I am also grateful to the speakers of the 2015 Marktoberdorf Summer School for important comments on the concepts underlying the alias and frame calculi, and to the organizers (Manfred Broy, Alexander Pretschner, Maximilian Irlbeck and Katharina Spies) for giving me the opportunity to present this work in the unique setting of the Marktoberdorf school.

References

[1] Alexander Kogtenkov, Bertrand Meyer and Sergey Velder: *Alias Calculus, Frame Calculus and Frame Inference*, in *Science of Computer Programming*, vol. 97, part 1, January 2015, pages 163-172.

[2] K. Rustan M. Leino, Arnd Poetzsch-Heffter and Yunhong Zhou: *Using data groups to specify and check side effects*, in PLDI (Programming Language Design and Implementation), 2002.

[3] John McCarthy and Patrick J. Hayes: Some philosophical problems from the standpoint of artificial intelligence, Machine Intelligence 4, eds. Melzer and Michie, 1969, pages 463–502.

[4] Bertrand Meyer: *Object-Oriented Software Construction*, 2nd edition, Prentice Hall, 1997.

[5] Bertrand Meyer: *Steps Towards a Theory and Calculus of Aliasing*, in *International Journal of Software and Informatics*, 2011, pages 77-116.

[6] Bertrand Meyer and Alexander Kogtenkov: *Negative Variables and the Essence of Object-Oriented Programming*, in *Specification, Algebra, and Software*, Kanazawa (Japan), 14-16 April 2014, eds. Shusaku Iida, Jose Meseguer and Kazuhiro Ogata, Springer Lecture Notes in Computer Science 8313, pages 171-187, 2014.

[7] Nadia Polikarpova, Carlo A. Furia, Yi Wei, Yu Pei and Bertrand Meyer: *What Good are Strong Specifications?*, in proceedings of ICSE 2013 (35th Int. Conf. on Software Engineering), San Francisco, May 2013.