An Abstract Interpretation Framework for Refactoring with Application to Extract Methods with Contracts

Patrick Cousot Radhia Cousot
ENS, CNRS, INRIA & NYU CNRS, ENS, INRIA
pcousot@cims.nyu.edu {cousot, rcousot}@ens.fr

Francesco Logozzo Michael Barnett
Microsoft Research
{logozzo,mbarnett}@microsoft.com

Abstract

Method extraction is a common refactoring feature provided by most modern IDEs. It replaces a user-selected piece of code with a call to an automatically generated method. We address the problem of *automatically* inferring contracts (precondition, postcondition) for the extracted method. We require the inferred contract: (a) to be valid for the extracted method (validity); (b) to guard the language and programmer assertions in the body of the extracted method by an opportune precondition (safety); (c) to preserve the proof of correctness of the original code when analyzing the new method separately (completeness); and (d) to be the most general possible (generality). These requirements rule out trivial solutions (*e.g.*, inlining, projection, etc.)

We propose two theoretical solutions to the problem. The first one is simple and optimal. It is valid, safe, complete and general but unfortunately not effectively computable (except for unrealistic finiteness/decidability hypotheses). The second one is based on an iterative forward/backward method. We show it to be valid, safe, and, under reasonable assumptions, complete and general. We prove that the second solution subsumes the first. All justifications are provided with respect to a new, set-theoretic version of Hoare logic (hence without logic), and abstractions of Hoare logic, revisited to avoid surprisingly unsound inference rules.

We have implemented the new algorithms on the top of two industrial-strength tools (`CCCheck` and the Microsoft Roslyn CTP). Our experience shows that the analysis is both fast enough to be used in an interactive environment and precise enough to generate good annotations.

Categories and Subject Descriptors D. Software [*D.1 Programming Techniques*]: D.1.0 General, D.2.1 Requirements/Specifications, D.2.2 Design Tools and Technique, D.2.4 Software/Program Verification, D.2.5 Testing and Debugging

General Terms Design, Documentation, Experimentation, Human Factors, Languages, Reliability, Verification.

Keywords Abstract interpretation, Design by contract, Method extraction, Program transformation, Refactoring, Static analysis.

1. Introduction

In their everyday activity, professional programmers heavily rely on the use of refactoring tools to improve, simplify, clean up, document, and modularize their code. Modern Integrated Development Environments (IDE) such as Eclipse, IntelliJ IDEA, or Visual Studio offer simple user interfaces to automate very tedious and error-prones activities. Method extraction is used at design time to avoid code-bloat, to improve code readability, to emphasize reuse, and to simplify methods. Method extraction consists in selecting a piece of code and asking the IDE refactoring engine to produce a new version of the program where: (i) the selected code is replaced by a call to a newly generated method (the *extracted method*); and (ii) the extracted method's parameters are the variables used (read/written) in the selected code and its body is the selected code. The engine must guarantee that the new program is a *syntactically* legal program, *i.e.*, if the original program compiled with no errors, then the new one compiles successfully, too. Furthermore, the *concrete semantics* of the original program (up to the additional method call) should be preserved in the new version. The problem of generating a syntactically correct refactored program (*e.g.*, [21, 24, 31]) is now considered a solved problem. However, the interaction between refactoring and static program analysis and verification has received minimal, if any, attention.

We are interested in the interaction between method extraction and static analysis and verification in a Design by Contract context (DbC) [35]. We focus our attention on the inference of *good* contracts (preconditions and postconditions) for the extracted method. Contracts are useful for the automatic generation of documentation and for the separate modular analysis and verification.In DbC, contracts are used to reason across method boundaries. Our static analysis is based on an assume/guarantee reasoning where the correctness proof is split between the callee and the caller. During the analysis of a method body, its precondition is assumed and the postcondition should be proved. Dually, when the caller is analyzed, the precondition of the callee should be proven and its postcondition can be assumed.

Currently, refactoring tools bind the programmer to manually add the contracts in order to prove the modified method with its call to the extracted method (*e.g.*, by adding postconditions to the extracted method). Our goal is not only to infer the contracts automatically, but also to have good contracts. Intuitively, a good *inferred* contract should: (a) be valid for the extracted method (validity); (b) guard the language and programmer assertions in the body of the extracted method by an opportune precondition (safety); (c) preserve the proof of correctness of the original code when analyzing the new method separately (completeness); and (d) be the most general possible (generality). In particular, the generality requirement allows the new method to be called in contexts other than the original refactoring context, and it rules out trivial solutions such as, *e.g.*, projecting the abstract states at the beginning and the end of the selected text.

2. Informal introduction of the problem

Imprecision induced by refactoring We illustrate the problem with some C# examples. We use the CodeContracts API [4] to specifiy contracts[1].

Example 1. Let us consider the simple code snippet below. We assume the C# compiler is invoked with the `-checked+` switch, to generate overflow/underflow checks.

```
public int Decrement(int x) {
  Requires(x >= 5);
  Ensures(Result<int>() >= 0);

  while (x != 0) x--;

  return x;
}
```

Assuming the precondition holds, `CCCheck` proves that: (i) no arithmetic overflow/underflow happens; and (ii) that the method exit is reached with $\texttt{x} = 0$, validating the (weaker user-provided) postcondition.

Let us now select the loop and apply the extract method refactoring provided by Visual Studio. The new program

```
public int Decrement(int x) {
  Requires(x >= 5);
  Ensures(Result<int>() >= 0);

  x = NewMethod(x);

  return x;
}

private int NewMethod(int x) {
  while (x != 0) x--;

  return x;
}
```

can no longer be proved correct by `CCCheck`:

```
1  CodeContracts: Suggested requires: Contract.Requires((x == 0 || Int32.MinValue <= (x - 1)));
2  CodeContracts: ensures unproven: Contract.Result<int>() >= 0
3   + location related to previous warning
4  CodeContracts: Possible underflow in the arithmetic operation
5  CodeContracts: Checked 3 assertions: 1 correct 2 unknown
```

The analyzer suggests a necessary precondition for `NewMethod` (message $\#1$), *i.e.*, a precondition that should hold otherwise the execution will definitely fail later. The precondition is not sufficient to ensure that `NewMethod` is correct, though. It reports that it cannot prove that the postcondition of `Decrement` holds on exit (messages $\#2, \#3$) and that the decrement of `x` does not underflow (message $\#4$). The imprecision is caused by the modular reasoning perfomed by `CCCheck`: it analyzes each method in isolation, using method contracts as summaries for all called methods. So when analyzing `Decrement`, since `NewMethod` has no contracts, it assumes the worst case: the return value of `NewMethod` can be any integer. And when analyzing `NewMethod`, since it has no contracts, `x` is unconstrained: the decrement may underflow (*e.g.*, for an initial negative value of `x`). □

Our work is motivated by the weaknesses of the following state-of-the-art strawman solutions.

First solution: Method Inlining One way to solve the problem is to perform the inverse operation of method extraction: method inlining. In general, inlining makes the analysis more precise. Nevertheless, we reject this solution. We want the analysis to be modular, and to use only boundary annotations to reason on method calls. Boundary annotations have many advantages. First, they provide documentation for the method. Accurate documentation and early error-checking (*e.g.*, by means of defensive programming) are crucial aspects of robust programming. Second, they make the analysis more scalable: a method can be analyzed once and its results/specification can be used many times. Conversely, inlining may cause code bloat, with the same piece of code analyzed again and again. Third, boundary annotations provide check gates, which help in quickly understanding regressions and make the analysis results easier to understand for the end-user. For example, let us suppose that a method `m` returns a positive value, and that this fact is used by the callers to infer some complex property ϕ which eventually is used to discharge the assertion `a`. Now, let us suppose in the next version of the program the implementation of `m` is changed so to return a non-negative value. The value is propagated to the callers (*i.e.*, by inlining `m`), ϕ is no more inferred, and `a` cannot be proven anymore, so the analyzer issues a warning for `a`. For the user, it is in general very hard to trace back the cause of the problem to the change in `m`, in particular if she does not own `m`. However, with explicit postconditions, `m` would have the postcondition that it returns a positive value, so the static verifier can immediately spot the problem where it occurs, and provide better error messages to the user. Fourth, the extracted method may be later moved to another module, so that, *e.g.*, its body will no longer be available for inlining.

Second solution: Isolated analysis Analyzing only the extracteded method in isolation does not take into account the context of the refactored code. It would result in the trivial method precondition `true`, which is in general too

[1] In CodeContracts, contracts are specified as opportune method calls of static members of the .NET type `Contract`. In the examples, for the sake of readibility, we omit the explicit reference to the type `Contract`.

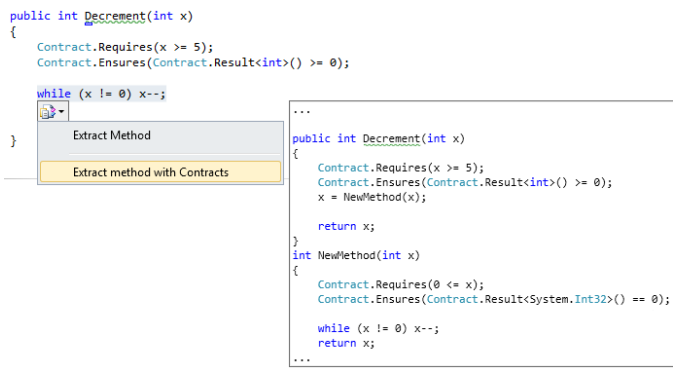


Figure 1. A screenshot of the extract method with contracts. The suggested contract for the extracted method is valid, safe, complete and the most general one.

imprecise. In particular, some information present in the original code (programmer assertions, runtime errors, etc) is lost when the refactored code is statically analyzed separately. This can be avoided by using the method *safety contract* suggested by `CCCheck`. When the safety precondition is violated, the execution of the extracted method will either not terminate or definitely yield a run-time error [18]. So the safety precondition is *necessary* for avoiding runtime errors. As shown by our example, the safety precondition is in general *not sufficient* to guarantee the absence of runtime errors: when the safety precondition is satisfied, the execution of the extracted method may or may not fail/terminate[2]. Once the necessary safety precondition is inferred [18], it can be used to get a safety post-condition by isolated reachability analysis of the method body [11, 13].

In general, an independent separate safety static analysis of the extracted method which does not take into account the pre-invariant and post-invariant of the selected code is too weak. It might not be strong enough to guarantee that the refactored code invariant is still provable separately. Our main motivation for this work was that the isolated analysis raised numerous (and self-evident) complaints from end-users of `CCCheck`.

Third solution: User assistance Another way of solving the problem is to require the user to provide the precondition and the postcondition for the extracted method. This is the actual state of the art: programmers using any form of DbC (CodeContracts, Spec#, JML, Eiffel, Separation Logic, etc.) need to manually insert the contracts for the extracted methods. We think that this is overkill and that this represents another barrier for a wider adoption of DbC methods. We think that method extraction should come with *automatic contract refactoring*, which automatically infers *good* contracts for the extracted method.

Forth solution: Abstract states projection An immediate idea to solve the problem consists in projecting the relevant variables from the original abstract proof so as to get the required modular proof. Such a solution is unsatisfactory for three main reasons. First, it does not work when refactoring unreachable code: the abstract state is empty, so the generated precondition is `false`. Second, too much information may be lost (*e.g.*, for relational analyzes) or too much information may be preserved (*e.g.*, not related to the method correctness). For instance, in Ex. 1, the projection of the abstract state produces the too strong precondition $5 \leq$ `x` for the extracted method. Ideally we'd love to infer the precondition $0 \leq$ `x`. Third, programs evolve over time so a refactoring might work when performed but no longer work with later program modifications.

Example 2. Suppose that we want to extract a method `MakeRoom` from `(*)...(**)` in the code below.

```
Insert(string[] list,
       ref int count, string newElement) {
  Requires(list != null &&
         // in bounds
    0 <= count &&  count <= list.Length &&
         // no overflows on resize
    list.Length < 33554432);

  if (list.Length == count)
  {
    (*) var tmp = new string[count*2 + 1];
    CopyArray(list, tmp);
    list = tmp; (**)
  }
 list[count++] = newElement;
 return list;
}
```

If we simply project the abstract states, the contract for the new method is

```
Requires(list!=null && list.Length==count   (1)
          && count <= 33554432);
Ensures(Result<string[]>() != null);
```

The precondition is too strong for the callers. The refactored method `MakeRoom` can only be invoked when `count` is exactly the length of the array *and* the array is not too large (less than 2^{25} elements). Furthermore, in the postcondition, because of the imprecision of the projection, we lost the relation between the length of the result array and `count`. With our technique, we instead infer the more general contract:

```
Requires(list != null && 0 <= count * 2 + 1);
Ensures(Result<string[]>() != null
 && 2 * count - Result<string[]>().Length == -1);
```

The precondition ensures that the internal allocation is safe (even with possible arithmetic overflows) and that we copy a valid list. The postcondition guarantees that the array returned

[2] The safety precondition is not a weakest liberal precondition that would be sufficient but maybe not necessary to guarantee the absence of runtime errors. The difference is that this sufficient precondition might exclude valid executions while the necessary safety precondition only excludes executions which are guaranteed to be definitely invalid or will not terminate.

by `MakeRoom` is large enough. Overall, many more callers are enabled and no (new) warning is raised in `Insert`. □

Our solution We want to suggest good contracts for extracted methods. The suggested contracts should enjoy some theoretical properties, to rule out the problems illustrated by the strawman solutions above.

(a) – validity First, the inferred contract should be valid for the extracted method. For instance, in Ex. 1, the following contract for `NewMethod`

```
Requires(5 <= x);
Ensures(Result<int>() == 12345);
```

would allow the proof of `Decrement` go through, but clearly is not satisfied by `NewMethod`'s implementation.

(b) – safety Second, the extracted method precondition should check the language and programmer assertions in the body of the extracted method [18]. For instance, the empty precondition for `NewMethod` in Ex. 1 does not meet this criterion (a negative input value causes an underflow). Without the safety precondition, the language and programmer assertions within the selected code would be hidden in the extracted method call. So the inference of a safety contract for the method enclosing the selected code, if any, would become impossible. The semantics is preserved, up to the fact that *definite errors* will be signaled earlier, at the method call, whereas in the original call they would have been signaled later, during execution of the selected code.

(c) – completeness Third, the verification of the original code should remain unchanged when analyzing the new method separately. For instance, the following contract for `NewMethod` in Ex. 1:

```
Requires(5 <= x);
Ensures(Result<int>() <= OldValue(x));
```

satisfies the (a) – validity and (b) – safety criteria, but it fails (c) – completeness.

(d) – generality Fourth, the automatically inferred contract should be the most general possible with the above properties. The generality criterion is important for reusability of the extracted method as well as to guarantee that other program modifications/transformations/refactorings are not influenced at all by this method refactoring.

For instance, the following contract for `NewMethod`

```
Requires(5 <= x);
Ensures(Result<int>() ==  0);
```

satisfies the three requirements (a—c) above, but it is not the most general one. The most general one is the one shown in Fig. 1. As also illustrated in Ex. 2, the generality criterion rules are important to rule out the trivial solution of using the (projection of the) abstract states at the beginning and at the end of the selected code as the new contract.

When the four conditions (a—d) above are satisfied, the refactored code is verifiable with the same precision as the original code, so that method extraction is guaranteed not to perturb the verification process. In general, the problem is undecidable, hence requires approximate solutions as discussed in this paper or specific additional hypotheses to ensure decidability (such as unrealistic finite state/decidability hypotheses).

3. Informal introduction of the solution

An example of the user experience with the algorithms in this paper using `CCCheck` and `Roslyn` is shown in Fig. 1. Given the selected portion of code, the IDE provides the option of the standard refactoring or our new refactoring. When selecting the new option, the preview shows the extracted method with the suggested contracts.

We informally describe the main steps of our solution. When the user selects a piece of code `S` and asks for the "extract method with contracts", we first invoke the "usual" extract method service of the IDE. If the selection cannot be made into a new method, then the refactoring fails and we stop here. Otherwise we obtain a snapshot of the source program as it appears after the refactoring. Our goal is to annotate the `NewMethod` with *good* contracts, *i.e.*, contracts satisfying the four requirement (a–d) exposed in the previous section.

After a static analysis of the original program, the first step of our solution is to detect the pre-state and post-state of the selected code `S` on the variables of interest. We identify the variables of interest from the invocation of `NewMethod` in the refactored code. In Ex. 1, `x` is a variable of interest for both the pre-state and the post-state: for the former since `x` is the actual parameter in the invocation of `NewMethod` and for the latter since it is where the return value of `NewMethod` is stored. To get the pre-state (resp. post-state), we query the underlying static analysis for the *abstract* value of `x` at the beginning (resp. end) of the selection. In the example the pre-state is $P_{\mathtt{S}} \triangleq 5 \leq \mathtt{x}$. Intuitively, $P_{\mathtt{S}}$ is a lower bound for the desidered precondition: in general we seek a more general (weaker) precondition for `NewMethod`. Similarly, the post-state is the (abstract) value of `x` at the end of the selection, *i.e.*, $Q_{\mathtt{S}} \triangleq \mathtt{x} = 0$. Intuitively, $Q_{\mathtt{S}}$ is an upper bound for the desidered postcondtion: in general we seek a more specific (stronger) postcondition for `NewMethod`.

The next step of our solution is to infer the safety (or necessary) precondition for `NewMethod` (and the corresponding postcondition). The idea is that of pushing inevitable safety checks (*e.g.*, runtime errors) back to the entry of the method so to expose them to the callers. Suppose for a second that we have an effective algorithm: (i) to infer the *best* safety preconditions; and (ii) to compute the strongest postconditions. Then we have a solution to our problem: the precondition is the best safety precondition and the postcondition is the strongest postcondition starting from that precondition (Th. 10). In Ex. 1 the best safety precondition is $0 \leq \mathtt{x}$: an initial negative value for `x` definitely causes an arithmetic underflow. Unfortunately, in practice and in the general case, such a precise and terminating algorithm does not exist — the problem is undecidable. As a consequence we must perform some approximations to make the problem tractable. The next steps of

our solution are designed to cope with that issue. Intuitively we use a combination of over- and under-approximations, and of forward and backward analyses to compensate for the loss of precision inherent in the abstraction (Th. 11 and Th. 22).

We first compute an *under*-approximation of the best safety precondition [18]. In the example, our analysis infers a safety precondition $P_{\mathtt{m}} \triangleq \mathtt{x} \neq \mathsf{MinValue}$. If this fails to hold, execution certainly results in an error. However, it does not guarantee the absence of errors. Starting with the abstract state $P_{\mathtt{m}}$, we use an *over*-approximating forward analysis to compute the corresponding postcondition $Q_{\mathtt{m}}$. The postcondition captures the final states of executions that do not result in an error. In the example $Q_{\mathtt{m}} \triangleq \mathtt{x} = 0$ — if the loop terminates at all, it terminates in that state. The contract $\langle P_{\mathtt{m}},\ Q_{\mathtt{m}}\rangle$ is more general than $\langle P_{\mathtt{S}},\ Q_{\mathtt{S}}\rangle$ — the precondition enables more calling contexts and $Q_{\mathtt{m}} = Q_{\mathtt{S}}$. However, the contract is not safe, *e.g.*, an error still occurs when $\mathtt{x} = -1$. Therefore we need to further refine it. We use an *over*-approximating backwards analysis to infer a better precondition. In our example this precondition is $P_R \triangleq 0 \leq \mathtt{x}$ — note that P_R implies $P_{\mathtt{m}}$. The corresponding postcondition, obtained by the forward analysis remains $Q_R \triangleq \mathtt{x} = 0$. The contract $\langle P_R,\ Q_R\rangle$ is more general than $\langle P_{\mathtt{S}},\ Q_{\mathtt{S}}\rangle$. While in general continuing the iterations may improve the contract, in our example we already found a fixpoint. Our analysis proves that the contract is safe — no runtime error will occur in `NewMethod` body. Therefore, we annotate `NewMethod` with the contract $\langle P_R,\ Q_R\rangle$. Overall, we inferred a contract satisfying (a–d). In particular, it is a *better* contract than $\langle P_{\mathtt{S}},\ Q_{\mathtt{S}}\rangle$, *i.e.*, we improved over the simple abstract states projection.

The example above did not really exploit $P_{\mathtt{m}}$: the backwards analysis compensated for any imprecision in the safety precondition inference. The refactoring of `Insert` shows an example where a precise $P_{\mathtt{m}}$ and forward analysis are relevant. We already reported the projected contract $\langle P_{\mathtt{S}},\ Q_{\mathtt{S}}\rangle$ in (1). The inferred safety precondition is $P_{\mathtt{m}} \triangleq \mathtt{list} \neq \mathtt{null} \wedge 0 \leq 2 \cdot \mathtt{count} + 1$. It manifests the fact that `CopyArray` will fail if given a `null` reference and that the allocation will fail if `count` is negative *or* so large that doubling it causes overflow. The corresponding postcondition is $Q_{\mathtt{m}} \triangleq \mathtt{result} \neq \mathtt{null} \wedge \mathtt{result.Length} = 2 \cdot \mathtt{count} + 1$. $\langle P_{\mathtt{m}},\ Q_{\mathtt{m}}\rangle$ is a better contract than $\langle P_{\mathtt{S}},\ Q_{\mathtt{S}}\rangle$, as $P_{\mathtt{m}}$ is more general than $P_{\mathtt{S}}$ and $Q_{\mathtt{m}}$ is more specific than $Q_{\mathtt{S}}$. In this case, the backwards analysis does not improve to the contract. $\langle P_{\mathtt{m}},\ Q_{\mathtt{m}}\rangle$ is the fixpoint, and it satisfies (a–d).

These are the two extremes. In general, the combination of the safety precondition inference with the forward/backwards iterations improve each other, and provide a very powerful algorithm to infer *good* contracts (Alg. 5).

4. Main Results

In order to rigorously define the Extract Method with Contracts (EMC) problem and the refactoring with contracts in general, we need an opportune mathematical formalism, for instance to reason about method calls. Intuitively, Hoare Logic provides such a formalism. We seek generality so that our results can be applied in many different contexts, with different abstract domains and specification logics. Therefore, we do not want to be specific to a particular assertion language, *e.g.*, first order logic [7, 22], separation logic [39], or region logic [2]. To the best of our knowledge there is no such *general* theory to reason about contracts, and so we had to build it.

Our first contribution is the development of a new set-theoretic version of Hoare logic (Algebraic Hoare Logic, Sec. 5), with the idea that the particular logics used by the analysis/verification tools are just an abstraction of those sets. A surprising result is that common inference rules in Hoare Logic, like the conjuction and disjunction rules are false in general (Ex. 5).

We use algebraic Hoare logic to define the elements of the domain $\mathcal{C}[\![\mathtt{m}]\!]$ of the contracts for a method `m` and two orders over such domain, a covariant order $\overset{\cdot}{\Longrightarrow}$ and a contravariant order $\overset{\mathsf{cc}}{\Longrightarrow}$ (Sec. 6). The first order captures the intuition that a $\overset{\cdot}{\Longrightarrow}$-stronger contract is better for the callee: assuming more on the precondition let it guarantee more on the postcondition. The second order captures the intuition that a $\overset{\mathsf{cc}}{\Longrightarrow}$-stronger contract is better for the callers: it is more general and it can be used in more contexts. The set $\mathcal{C}[\![\mathtt{m}]\!]$ of contracts for `m` ordered either by $\overset{\cdot}{\Longrightarrow}$ or by $\overset{\mathsf{cc}}{\Longrightarrow}$ is a complete lattice.

The algebraic Hoare logic, the set of contracts and the two orders provide a basic framework for the definition of contract-based refactorings. We instantiate it to state the *new* problem of the extract method with contracts (EMC, Sec. 8). We formally define the four requirements (a–d) of the previous section in terms of algebraic Hoare logic and the two orders $\overset{\cdot}{\Longrightarrow}$ and $\overset{\mathsf{cc}}{\Longrightarrow}$.

A main theoretical result of the paper is that in the concrete the EMC problem has always a solution, and this solution is unique (Th. 10). Roughly, the suggested precondition is the *strongest safety* precondition $P_{\mathtt{m}}$ (to make sure the caller encounters no runtime error when `m` is executed) and the suggested postcondition is the strongest postcondition from $P_{\mathtt{m}}$. The result is of great theoretical interest, but of little practical application: the strongest safety precondition and the strongest postcondition are not computable in general. The direct abstraction is likely to produce a very imprecise result, as over-approximation may lose completeness. Therefore, we provide an equivalent characterization for the solution of the EMC problem, in terms of the iteration of a forwards analysis and of a backwards analysis (Th. 11) allowing for a more precise algorithm in the abstract.

To provide an effective solution to the EMC problem we should perform some abstraction. We define the notion of abstract contracts, essentially contracts where the assertion language is some abstraction of sets of states (Sec. 10). Surprisingly enough, we found that in general the property of being a more precise abstract contract is not preserved in

the concrete (Ex. 12). We restate the EMC problem in terms of the primitives of the underlying abstract domain ($\overline{\text{EMC}}$, Sec. 12). We prove soundness, *i.e.*, a solution to $\overline{\text{EMC}}$ is a solution for EMC (Th. 15). We present some examples proving that, in general, a complete $\overline{\text{EMC}}$ is impossible.

We abstract the iterated formulation of the EMC solution to provide an effective static analysis to compute $\overline{\text{EMC}}$ (Sec. 13). Let us assume to have an abstract transformer (roughly, the two-directional static analysis for the method body) safely approximating the concrete semantics of the method and a projection of the abstract states in the original method (before the extraction of the method) $\langle P^{\curlyvee}_{\mathtt{S}},\ Q^{\curlyvee}_{\mathtt{S}}\rangle$. Then the iterations of the abstract transformer starting from $\langle P^{\curlyvee}_{\mathtt{S}},\ Q^{\curlyvee}_{\mathtt{S}}\rangle$ provide a correct solution (Th. 20). When the abstract transformer is the *best* approximation of the concrete transformer, the abstract forwards/backwards iterations provide the *most precise* solution for $\overline{\text{EMC}}$ (Th. 21). When the underlying abstract domain does not satisfy the Ascending/Descending chain conditions, a fixpoint acceleration operator (narrowing [11]) should be used to enforce the convergence of the iterations (Algorithm $\overline{\text{EMC}}$ in Alg. 5). The resulting contract is still a correct solution (Th. 22), but we may not get the most general solution — just one more general than the simple projection.

We implemented the new algorithms by integrating the Code Contracts for .NET static analyzer (`CCCheck`) [20] with the Microsoft Roslyn CTP (`Roslyn`) [37]. We use `Roslyn`, a new implementation of .NET languages to support the compiler-as-service paradigm, as our refactoring engine. We use `CCCheck` as the underlying static analyzer. Unlike similar tools (*e.g.*, [22, 23]), `CCCheck` is based on abstract interpretation, and it automatically infers and propagates loop invariants intra-procedurarly, so that annotations are needed only for the method boundaries. The inferred invariants are used to validate both the user-provided contracts as well as the absence of runtime errors (*e.g.*, null dereference, underflow/overflow, buffer overruns, etc.). Our experience shows that the proposed method extraction with contracts is quite effective (Sec. 14).

5. Algebraic Hoare Logic

We use Hoare logic [29] to formalize Contracts [4, 35]. The *concrete* Hoare rules are used to specify the program axiomatic semantics, *i.e.*, all possible program executions.

We use an *abstract* version of Hoare logic to formalize contract-based separate static analyses. The abstract Hoare rules are used to specify how the static analyzer should work for a given abstraction. In this abstract Hoare logic, predicates are replaced by abstract properties chosen in computer-representable abstract domains with computable transformers and fixpoint approximation [12] such as intervals [10], octagons [36], subpolyhedra [30], or polyhedra [16].

The general correctness argument is that static analyzers are correct because they implement an abstract Hoare logic which is itself sound because it correctly abstracts a concrete Hoare logic describing precisely the language semantics.

Both concrete and abstract Hoare logics can be formalized in a single unified framework using *algebraic* Hoare triples and abstract interpretation to relate algebraic Hoare logics operating at different levels of abstraction.

In this context the conjunction rule is potentially problematic in the abstract (as illustrated in the forthcoming Ex. 5). We cannot get rid of this conjunction rule because it formalizes the use of reduced products [13] in static analyzers. Therefore we study sufficient conditions on the abstraction for this conjunction rule to be sound (Th. 6) which is useful beyond the specific problem of method refactoring (Ex. 7).

We first introduce some definitions and notations used in the rest of the paper.

Galois Connections We recall from [11] that a Galois connection $\langle C,\ \preceq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A,\ \sqsubseteq\rangle$ is such that $\langle C,\ \preceq\rangle$ and $\langle A,\ \sqsubseteq\rangle$ are partial orders, $\alpha \in C \to A$ and $\gamma \in C \to A$ satisfy $\forall x \in C : \forall y \in A : \alpha(x) \sqsubseteq y \iff x \preceq \gamma(y)$. We write $\langle C,\ \preceq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows\!\!\!\!\rightarrow}} \langle A,\ \sqsubseteq\rangle$ to denote that the abstraction function α is surjective, and hence that there are no multiple representations for the same concrete property in the abstract. If the C and A are complete lattices, and α is join-preserving, then it exists a unique γ such that $\langle C,\ \preceq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A,\ \sqsubseteq\rangle$.

Abstract domains We let $\mathtt{S} \in \mathbb{S}[\![\vec{\mathtt{v}}]\!]$ be a statement with visible variables $\vec{\mathtt{v}}$ and $\mathcal{P}[\![\vec{\mathtt{v}}]\!]$ be the set of unary predicates on variables $\vec{\mathtt{v}}$. Predicates can be isomorphically represented as Boolean functions $P \in \mathcal{P}[\![\vec{\mathtt{v}}]\!] \triangleq \vec{\mathcal{V}}[\![\vec{\mathtt{v}}]\!] \to \mathbb{B}$ mapping values $\vec{v} \in \vec{\mathcal{V}}[\![\vec{\mathtt{v}}]\!]$ of vector values of variables $\vec{\mathtt{v}}$ to Booleans: $P(\vec{v}) \in \mathbb{B} \triangleq \{\mathtt{true}, \mathtt{false}\}$. Predicates are ordered according to $\dot{\Longrightarrow}$, *i.e.*, the pointwise lifting of logical implication to functions:

$$P \mathrel{\dot{\Longrightarrow}} P' \ \triangleq\ \forall \vec{v} \in \vec{\mathcal{V}}[\![\vec{\mathtt{v}}]\!] : P(\vec{\mathtt{v}}) \Longrightarrow P'(\vec{\mathtt{v}}).$$

For example $\boldsymbol{\lambda}\, x \bullet x = 0 \mathrel{\dot{\Longrightarrow}} \boldsymbol{\lambda}\, x \bullet x \geqslant 0$. Predicates with partial order $\dot{\Longrightarrow}$ form a complete Boolean lattice:

$$\langle \mathcal{P}[\![\vec{\mathtt{v}}]\!],\ \dot{\Longrightarrow},\ \dot{\mathtt{false}},\ \dot{\mathtt{true}},\ \dot{\vee},\ \dot{\wedge},\ \dot{\neg}\rangle$$

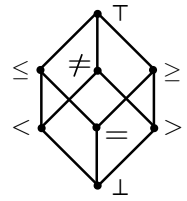
where $\dot{\mathtt{false}}$ is the infimum, $\dot{\mathtt{true}}$ is the supremum, $\dot{\vee}$ is the least upper bound (lub), $\dot{\wedge}$ is the greatest lower bound (glb), and $\dot{\neg}$ is the unique complement for the partial order $\dot{\Longrightarrow}$ on the set $\mathcal{P}[\![\vec{\mathtt{v}}]\!]$.

The *precondition abstract domain* $\langle A[\![\vec{\mathtt{v}}]\!], \sqsubseteq\rangle$ is an abstract domain expressing properties of the variables $\vec{\mathtt{v}}$ where the partial order $\sqsubseteq$ abstracts logical implication. The meaning of an abstract property $\overline{P} \in A[\![\vec{\mathtt{v}}]\!]$ is a concrete property $\gamma_1(\overline{P}) \in \mathcal{P}[\![\vec{\mathtt{v}}]\!]$ where the concretization

$$\gamma_1 \in \langle A[\![\vec{\mathtt{v}}]\!],\ \sqsubseteq\rangle \ \to\ \langle \mathcal{P}[\![\vec{\mathtt{v}}]\!],\ \dot{\Longrightarrow}\rangle$$

is increasing (*i.e.*, $\overline{P} \sqsubseteq \overline{P}'$ implies $\gamma_1(\overline{P}) \mathrel{\dot{\Longrightarrow}} \gamma_1(\overline{P}')$).

Example 3. Assume that $\vec{\mathtt{v}} \triangleq \mathtt{x}$ is reduced to a single variable $\mathtt{x}$. Let $A[\![\vec{\mathtt{v}}]\!]$ be the lattice with the ordering $\sqsubseteq$ defined by the following Hasse diagram:



$\langle A[\![\mathtt{x}]\!],\ \sqsubseteq^1 \rangle$

$$\begin{array}{lcl}
\gamma_1(\top) & \triangleq & \dot{\mathtt{true}} \\
\gamma_1(\leq) & \triangleq & \boldsymbol{\lambda}\, x \bullet x \leqslant 0 \\
\gamma_1(\neq) & \triangleq & \boldsymbol{\lambda}\, x \bullet x \neq 0 \\
\gamma_1(\geq) & \triangleq & \boldsymbol{\lambda}\, x \bullet x \geqslant 0 \\
\gamma_1(<) & \triangleq & \boldsymbol{\lambda}\, x \bullet x < 0 \\
\gamma_1(=) & \triangleq & \boldsymbol{\lambda}\, x \bullet x = 0 \\
\gamma_1(>) & \triangleq & \boldsymbol{\lambda}\, x \bullet x > 0 \\
\gamma_1(\bot) & \triangleq & \dot{\mathtt{false}}\,.
\end{array}$$

According to the definition of γ_1, $A[\![\mathtt{x}]\!]$ is interpreted in the concrete as specifying the sign of values $x \in \vec{\mathcal{V}}[\![\mathtt{x}]\!]$ of variable x [13]. □

The *postcondition abstract domain* $\langle B[\![\vec{\mathtt{v}}, \vec{\mathtt{v}}]\!],\ \sqsubseteq^2 \rangle$ is an abstraction of the complete lattice

$$\langle \mathcal{P}[\![\vec{\mathtt{v}}, \vec{\mathtt{v}}]\!],\ \dot{\Longrightarrow},\ \dot{\mathtt{false}},\ \dot{\mathtt{true}},\ \dot{\vee},\ \dot{\wedge},\ \dot{\neg} \rangle$$

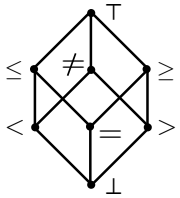
of binary predicates, *e.g.*, postconditions relating the initial and final values of variables. The meaning $\gamma_2(Q)$ of an abstract relation $\overline{Q} \in B[\![\vec{\mathtt{v}}, \vec{\mathtt{v}}]\!]$ is given by a *finite-meet-preserving* concretization

$$\gamma_2 \in \langle B[\![\vec{\mathtt{v}}, \vec{\mathtt{v}}]\!],\ \sqsubseteq^2 \rangle \ \rightarrow\ \langle \mathcal{P}[\![\vec{\mathtt{v}}, \vec{\mathtt{v}}]\!],\ \dot{\Longrightarrow} \rangle,$$

satifying $\gamma_2(\overline{Q} \sqcap^2 \overline{Q}\,') = \gamma_2(\overline{Q}) \,\dot{\wedge}\, \gamma_2(\overline{Q}\,')$.

The finite-meet hypothesis is needed to avoid the problems exposed in the forthcoming Ex. 5. The finite-meet hypothesis implies that γ_2 is increasing. The function γ_2 is *meet-preserving* if and only if it preserves infinite meets hence is the upper-adjoint of a Galois connection [13]. A meet-preserving function is trivially finite-meet preserving.

Example 4. Assume that $\vec{\mathtt{v}} \triangleq \mathtt{x}$ is reduced to a single variable x. Let $B[\![\vec{\mathtt{v}}, \vec{\mathtt{v}}]\!]$ be the lattice with the ordering $\sqsubseteq^2$ defined by the following Hasse diagram:



$\langle B[\![\mathtt{x}, \mathtt{x}]\!],\ \sqsubseteq^2 \rangle$

$$\begin{array}{lcl}
\gamma_2(\top) & \triangleq & \dot{\mathtt{true}} \\
\gamma_2(\leq) & \triangleq & \boldsymbol{\lambda}\, \mathtt{x}, \mathtt{x}' \bullet \mathtt{x} \leqslant \mathtt{x}' \\
\gamma_2(\neq) & \triangleq & \boldsymbol{\lambda}\, \mathtt{x}, \mathtt{x}' \bullet \mathtt{x} \neq \mathtt{x}' \\
\gamma_2(\geq) & \triangleq & \boldsymbol{\lambda}\, \mathtt{x}, \mathtt{x}' \bullet \mathtt{x} \geqslant \mathtt{x}' \\
\gamma_2(<) & \triangleq & \boldsymbol{\lambda}\, \mathtt{x}, \mathtt{x}' \bullet \mathtt{x} < \mathtt{x}' \\
\gamma_2(=) & \triangleq & \boldsymbol{\lambda}\, \mathtt{x}, \mathtt{x}' \bullet \mathtt{x} = \mathtt{x}' \\
\gamma_2(>) & \triangleq & \boldsymbol{\lambda}\, \mathtt{x}, \mathtt{x}' \bullet \mathtt{x} > \mathtt{x}' \\
\gamma_2(\bot) & \triangleq & \dot{\mathtt{false}}\,.
\end{array}$$

According to the definition of γ_2, $B[\![\mathtt{x}, \mathtt{x}]\!]$ is interpreted in the concrete as specifying a relation between the values x and x' of variable x (*e.g.*, before and after executing a piece of code to be refactored). □

Concrete Hoare triples A *concrete Hoare triple* $\{P\}\,\mathtt{S}\,\{Q\}$ denotes the partial correctness of a program statement $\mathtt{S} \in \mathbb{S}[\![\vec{\mathtt{v}}]\!]$. It denotes the fact that if the precondition $P \in \mathcal{P}[\![\vec{\mathtt{v}}]\!]$ holds of the values of the variables before executing statement S, and the execution of the statement S does terminate, then the postcondition/before-after relation $Q \in \mathcal{P}[\![\vec{\mathtt{v}}, \vec{\mathtt{v}}]\!]$ holds and relates the initial and final values of the variables $\vec{\mathtt{v}}$ before and after the execution of S. Concrete Hoare triples can be understood as Boolean functions: [3]

$$\{\bullet\} \bullet \{\bullet\} \ \in\ \mathcal{P}[\![\vec{\mathtt{v}}]\!] \times \mathbb{S}[\![\vec{\mathtt{v}}]\!] \times \mathcal{P}[\![\vec{\mathtt{v}}, \vec{\mathtt{v}}]\!] \to \mathbb{B}\,.$$

Concrete Hoare logic rules The classical axiomatization of Hoare logic remains valid in set-theoretical form as shown in Fig. 2. The disjunction rule $(\vee)$ and conjunction rule $(\wedge)$ in Fig. 2 are usually not shown in Hoare logic axiomatization since they derive from the other rules, by induction on the structure of programs.

Predicate transformers We use a generalization of the usual Dijkstra's strongest postconditions predicate tranformer [19] to sets (instead of logical formulas). The set-theoretic *forward* predicate transformer post $\in \mathbb{S}[\![\vec{\mathtt{v}}]\!] \to (\mathcal{P}[\![\vec{\mathtt{v}}]\!] \to \mathcal{P}[\![\vec{\mathtt{v}}, \vec{\mathtt{v}}]\!])$ of [9] provides such a generalization. The transformer post verifies the two properties:

$$\{\, P \,\}\, \mathtt{S}\, \{\, \mathrm{post}\,[\![\mathtt{S}]\!] P \,\}, \tag{2}$$
$$\forall Q \in \mathcal{P}[\![\vec{\mathtt{v}}, \vec{\mathtt{v}}]\!] : \{\, P \,\}\, \mathtt{S}\, \{\, Q \,\} \Longrightarrow (\mathrm{post}\,[\![\mathtt{S}]\!] P \Longrightarrow Q)\,.$$

The forward predicate transformer post $[\![\mathtt{S}]\!]$ is join-preserving. Therefore, it has a unique adjoint $\widetilde{\mathrm{pre}}[\![\mathtt{S}]\!]$ such that

$$\langle \mathcal{P}[\![\vec{\mathtt{v}}]\!],\ \dot{\Longrightarrow} \rangle \underset{\mathrm{post}[\![\mathtt{S}]\!]}{\overset{\widetilde{\mathrm{pre}}[\![\mathtt{S}]\!]}{\longleftarrow}} \langle \mathcal{P}[\![\vec{\mathtt{v}}, \vec{\mathtt{v}}]\!],\ \dot{\Longrightarrow} \rangle \tag{3}$$

is a Galois connection, *i.e.*, $\forall P \in \mathcal{P}[\![\vec{\mathtt{v}}]\!] : \forall Q \in \mathcal{P}[\![\vec{\mathtt{v}}, \vec{\mathtt{v}}]\!] : (\mathrm{post}\,[\![\mathtt{S}]\!] P \Longrightarrow Q) \iff (P \Longrightarrow \widetilde{\mathrm{pre}}[\![\mathtt{S}]\!] Q)$. Intuitively, $\widetilde{\mathrm{pre}}[\![\mathtt{S}]\!]$ is a generalization to sets of Dijkstra's weakest liberal preconditions predicate transformer.

Abstract Hoare triples An *abstract Hoare triple* is similar to a concrete Hoare triple except that the precondition and postcondition are chosen in abstract domains as used, *e.g.*, by a static analyzer or a SMT-solver [17]:

$$\bar{\{}\, \bullet \,\bar{\}} \bullet \bar{\{}\, \bullet \,\bar{\}} \ \in\ A[\![\vec{\mathtt{v}}]\!] \times \mathbb{S}[\![\vec{\mathtt{v}}]\!] \times B[\![\vec{\mathtt{v}}, \vec{\mathtt{v}}]\!] \to \mathbb{B}$$

The concrete Hoare triples are a particular case of abstract Hoare triples by choosing $A[\![\vec{\mathtt{v}}]\!] = \mathcal{P}[\![\vec{\mathtt{v}}]\!]$, $B[\![\vec{\mathtt{v}}, \vec{\mathtt{v}}]\!] = \mathcal{P}[\![\vec{\mathtt{v}}, \vec{\mathtt{v}}]\!]$, γ_1 and γ_2 to be the identity. We say that an abstract Hoare triple is *sound* if and only if

$$\bar{\{}\, \overline{P} \,\bar{\}}\, \mathtt{S}\, \bar{\{}\, \overline{Q} \,\bar{\}} \ =\ \{\, \gamma_1(\overline{P}) \,\}\, \mathtt{S}\, \{\, \gamma_2(\overline{Q}) \,\} \tag{4}$$

Abstract Hoare Logic Rules In the abstract, program statements are handled by abstraction [11]. The corresponding abstract rules are in Fig. 3. Surprisingly, the following counter-example shows that the abstract rules of Fig. 3 may be unsound in the sense of (4). This is because the classical version of Hoare logic makes implicit assumptions upon the acceptable interpretations of logical predicates/assertions which may not be preserved by the abstraction since, *e.g.*, for $(\overline{\wedge})$, $\gamma_1(\bar{\bigsqcap}^1_{i \in \Delta} \overline{P}_i) \Longrightarrow \dot{\bigwedge}_{i \in \Delta} \gamma_1(\overline{P}_i)$ but not inversely when γ_1 is increasing but not join-preserving.

Example 5 (Unsound abstract interpretation)**.** Consider the following abstractions where A is the pre-condition abstract domain

[3] This point of view consists in considering a particular interpretation of Hoare logic, the one corresponding to the programming language semantics.

$$\frac{}{\left\{\,\dot{\mathtt{false}}\,\right\}\,\mathtt{S}\,\left\{\,Q\,\right\}}\ (\bot) \qquad \frac{}{\left\{\,P\,\right\}\,\mathtt{S}\,\left\{\,\dot{\mathtt{true}}\,\right\}}\ (\top)$$

$$\frac{}{\left\{\,P\,\right\}\,\mathtt{skip}\,\left\{\,\boldsymbol{\lambda}\,\vec{v},\vec{v}'\bullet P(\vec{v})\wedge\vec{v}'=\vec{v}\,\right\}}\ (\mathtt{s}) \qquad \frac{}{\left\{\,P\,\right\}\,\mathtt{assert(E)}\,\left\{\,\boldsymbol{\lambda}\,\vec{v},\vec{v}'\bullet P(\vec{v})\wedge[\![\mathtt{E}]\!]\vec{v}\wedge\vec{v}'=\vec{v}\,\right\}}\ (\mathtt{a})$$

$$\frac{}{\left\{\,P\,\right\}\,\mathtt{x}\ =\ \mathtt{E}\,\left\{\,\boldsymbol{\lambda}\,\vec{v},\vec{v}'\bullet P(\vec{v})\wedge\vec{v}'=\vec{v}[\mathtt{x}\mapsto[\![\mathtt{E}]\!]\vec{v}]\,\right\}}\ (=)$$

$$\frac{\left\{\,P\,\right\}\,\mathtt{S}_1\,\left\{\,Q\,\right\},\left\{\,\boldsymbol{\lambda}\,\vec{v}'\bullet\forall\vec{v}:P(\vec{v})\Longrightarrow Q(\vec{v},\vec{v}')\,\right\}\,\mathtt{S}_2\,\left\{\,R\,\right\}}{\left\{\,P\,\right\}\,\mathtt{S}_1;\mathtt{S}_2\,\left\{\,\boldsymbol{\lambda}\,\vec{v},\vec{v}''\bullet\exists\vec{v}':Q(\vec{v},\vec{v}')\wedge R(\vec{v}',\vec{v}'')\,\right\}}\ (;)$$

$$\frac{\left\{\,\boldsymbol{\lambda}\,\vec{v}\bullet P(\vec{v})\wedge[\![\mathtt{E}]\!]\vec{v}\,\right\}\,\mathtt{S}_1\,\left\{\,Q_1\,\right\},\left\{\,\boldsymbol{\lambda}\,\vec{v}\bullet P(\vec{v})\wedge[\![\neg E]\!]\vec{v}\,\right\}\,\mathtt{S}_2\,\left\{\,Q_2\,\right\}}{\left\{\,P\,\right\}\,\mathtt{if(E)\ S_1\ else\ S_2}\,\left\{\,Q_1\,\dot{\vee}\,Q_2\,\right\}}\ (\mathtt{i})$$

$$\frac{\begin{array}{c}\forall\vec{v}:P(\vec{v})\Longrightarrow I(\vec{v},\vec{v}),\\ \left\{\,\boldsymbol{\lambda}\,\vec{v}'\bullet\forall\vec{v}:I(\vec{v},\vec{v}')\,\right\}\,\mathtt{assert(E);S}\,\left\{\,J\,\right\},\\ \forall\vec{v},\vec{v}',\vec{v}'':I(\vec{v},\vec{v}')\wedge J(\vec{v}',\vec{v}'')\Longrightarrow I(\vec{v},\vec{v}''),\\ \forall\vec{v},\vec{v}':I(\vec{v},\vec{v}')\wedge[\![\neg\mathtt{E}]\!]\vec{v}'\Longrightarrow Q(\vec{v},\vec{v}')\end{array}}{\left\{\,P\,\right\}\,\mathtt{while(E)\ S}\,\left\{\,Q\,\right\}}\ (\mathtt{w})$$

$$\frac{P\Longrightarrow P'\wedge\left\{\,P'\,\right\}\,\mathtt{S}\,\left\{\,Q'\,\right\}\wedge Q'\Longrightarrow Q}{\left\{\,P\,\right\}\,\mathtt{S}\,\left\{\,Q\,\right\}}\ (\Rightarrow)$$

$$\frac{\forall i\in\Delta:\left\{\,P_i\,\right\}\,\mathtt{S}\,\left\{\,Q_i\,\right\}}{\left\{\,\dot{\exists}i\in\Delta:P_i\,\right\}\,\mathtt{S}\,\left\{\,\dot{\exists}i\in\Delta:Q_i\,\right\}}\ (\vee) \qquad \frac{\forall i\in\Delta:\left\{\,P_i\,\right\}\,\mathtt{S}\,\left\{\,Q_i\,\right\}}{\left\{\,\dot{\forall}i\in\Delta:P_i\,\right\}\,\mathtt{S}\,\left\{\,\dot{\forall}i\in\Delta:Q_i\,\right\}}\ (\wedge)$$

Figure 2. Concrete Hoare triples axiomatization.

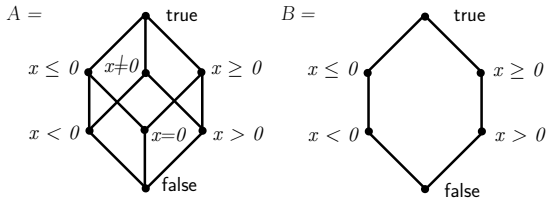
If $A[\![\vec{\mathtt{v}}]\!]$ has an infimum $\bot_A$ such that $\gamma_1(\bot_A)=\dot{\mathtt{false}}$ then for all $\overline{Q}\in B$, $\overline{\left\{\,\bot_A\,\right\}}\,\mathtt{S}\,\overline{\left\{\,\overline{Q}\,\right\}}=\mathtt{true}$ $(\overline{\bot})$

If $B[\![\vec{\mathtt{v}},\vec{\mathtt{v}}]\!]$ has a supremum $\top_B$ such that $\gamma_2(\top_B)=\dot{\mathtt{true}}$ then for all $\overline{P}\in A$, $\overline{\left\{\,\overline{P}\,\right\}}\,\mathtt{S}\,\overline{\left\{\,\top_B\,\right\}}=\mathtt{true}$ $(\overline{\top})$

$$\frac{\left\{\,\gamma_1(\overline{P})\,\right\}\,\mathtt{S}\,\left\{\,\gamma_2(\overline{Q})\,\right\}}{\overline{\left\{\,\overline{P}\,\right\}}\,\mathtt{S}\,\overline{\left\{\,\overline{Q}\,\right\}}}\ (\overline{\mathtt{S}}) \qquad \frac{\forall i\in\Delta:\overline{\left\{\,\overline{P}_i\,\right\}}\,\mathtt{S}\,\overline{\left\{\,\overline{Q}_i\,\right\}}}{\overline{\left\{\,\bigsqcup_{i\in\Delta}\overline{P}_i\,\right\}}\,\mathtt{S}\,\overline{\left\{\,\bigsqcup_{i\in\Delta}\overline{Q}_i\,\right\}}}\ (\overline{\vee})$$

$$\frac{\overline{P}\sqsubseteq\overline{P}'\wedge\overline{\left\{\,\overline{P}'\,\right\}}\,\mathtt{S}\,\overline{\left\{\,\overline{Q}'\,\right\}}\wedge\overline{Q}'\sqsubseteq\overline{Q}}{\overline{\left\{\,\overline{P}\,\right\}}\,\mathtt{S}\,\overline{\left\{\,\overline{Q}\,\right\}}}\ (\overline{\Rightarrow}) \qquad \frac{\forall i\in\Delta:\overline{\left\{\,\overline{P}_i\,\right\}}\,\mathtt{S}\,\overline{\left\{\,\overline{Q}_i\,\right\}}}{\overline{\left\{\,\bigsqcap_{i\in\Delta}\overline{P}_i\,\right\}}\,\mathtt{S}\,\overline{\left\{\,\bigsqcap_{i\in\Delta}\overline{Q}_i\,\right\}}}\ (\overline{\wedge})$$

Figure 3. Abstract Hoare triple axiomatization. Without additional hypotheses, the rules $(\overline{\vee})$ and $(\overline{\wedge})$ are unsound.



and the post-condition abstract domain B preserves neither joins nor meets. We have

$\overline{\left\{\,x\geq 0\,\right\}}\,\mathtt{x\ =\ -x}\,\overline{\left\{\,x\leq 0\,\right\}}$ and $\overline{\left\{\,x\leq 0\,\right\}}\,\mathtt{x\ =\ -x}\,\overline{\left\{\,x\geq 0\,\right\}}$

but definitely not the conjunction in $A\times B$

$$\overline{\left\{\,x\geq 0\sqcap x\leq 0\,\right\}}\,\mathtt{x\ =\ -x}\,\overline{\left\{\,x\leq 0\sqcap x\geq 0\,\right\}}$$

which is

$$\overline{\left\{\,x=0\,\right\}}\,\mathtt{x\ =\ -x}\,\overline{\left\{\,\mathtt{false}\,\right\}}$$

Similarly,

$\overline{\left\{\,x>0\,\right\}}\,\mathtt{x=x*x}\,\overline{\left\{\,x>0\,\right\}}$ and $\overline{\left\{\,x<0\,\right\}}\,\mathtt{x=x*x}\,\overline{\left\{\,x>0\,\right\}}$.

The disjunction in $B\times A$ is

$$\overline{\left\{\,x<0\sqcup x>0\,\right\}}\,\mathtt{x\ =\ x\ *\ x}\,\overline{\left\{\,x>0\sqcup x>0\,\right\}}$$

that is the unsound $\overline{\left\{\,\mathtt{true}\,\right\}}\,\mathtt{x\ =\ x\ *\ x}\,\overline{\left\{\,x>0\,\right\}}$. □

The Ex. 5 shows the necessity for γ_2 to preserve finite conjunction for the abstract conjunction rule $(\overline{\wedge})$ to be sound for finite abstract conjunctions. Similarly, $(\overline{\vee})$ is not sound when γ_1 is not join-preserving. More generally:

Theorem 6 (Sound abstract interpretation)**.** The abstract Hoare triple axiomatization of Fig. 3, without $(\overline{\wedge})$ and $(\overline{\vee})$ is sound in the sense of (4).

Moreover if γ_1 is increasing, the glbs do exist, γ_2 is finite-meet-preserving and Δ is finite then the abstract conjunction $(\overline{\wedge})$ is also sound. If γ_2 is increasing, the lub exists, and γ_1 is finite-join-preserving and Δ is finite or γ_1 is join preserving then the abstract disjunction rule $(\overline{\vee})$ is also sound. □

The notion of Algebraic Hoare Logic developped in this Sec. 5 and the issues with the unsoundness of the $(\overline{\wedge})$ and $(\overline{\vee})$ rules of the abstract Hoare logic of Fig. 3 as well as the discussion for when they are sound in Th. 6 are applicable beyond the specific problem of method re-factoring.

Example 7. Concurrent separation logic [38] is an example of algebraic Hoare logic where abstract domains are predicates over a separation algebra [27]. Because of the conjunction rule, the logic is unsound unless resource invariants are precise, *i.e.*, unambiguously carve out an area of the heap. For example, the separation logic assertion $x\mapsto 0$, denoting a cell

at the address x storing 0, is precise; however, the assertion $x \mapsto 0 \vee \mathtt{emp}$, denoting either the cell or the empty heap, is not. In particular, imprecise resource invariants allow the two premisses of the conjunction rule to make conflicting choices about how to partition the heap. For imprecise predicates, the concretization may not preserve intersection [27, Def. (14)].

As stated in Th. 6, one solution is to restrict the abstract domain (*i.e.*, the predicates over a separation algebra) to be finite-meet-preserving, which is the case for precise resource invariants, as in [6]. The second solution is to exclude the conjunction rule, as in [27]. □

Abstract Hoare Logic Rules for Static Analysis In the following we will use a sound version of the abstract Hoare logic with the conjunction rule ($\overline{\wedge}$) but not with disjunction rule ($\overline{\vee}$).

Abstract conjunction rule The conjunction rule ($\overline{\wedge}$) is of interest for static analyzers using reduced products [13]. Reduced products allow the automatic combination of separately designed analyzes so as to express conjunctions of different abstract properties.

Classical abstract domains such as intervals [10], octagons [36], subpolyhedra [30], or polyhedra [16] do satisfy the hypotheses of Th. 6 ensuring the soundness of the conjunction rule ($\overline{\wedge}$) since in those cases γ_2 is finite-meet-preserving (although not infinite meet-preserving since *e.g.*, for polyhedra [16] the concretization is not the upper-adjoint of a Galois connection).

Abstract disjunction rule For the disjunction rule, enforcing γ_1 to preserve (infinite) joins for ($\overline{\vee}$) is a very restrictive hypothesis essentially forbidding the approximation of joins which is the basis for static analysis.

However, the disjunction rule ($\overline{\vee}$) is not needed in static analyzers since disjunctions are usually handled specifically in each abstract domain.

6. Formalization of Contracts

We define the two notions of *valid* and *safe* method contracts in terms of Algebraic Hoare triples. We also introduce two partial orders needed for the formalization of the extract method with contracts.

Valid contract We write $\mathtt{S}|_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}$ to mean that all variables used or modified by a program statement S belong either to $\vec{\mathtt{p}}$ or $\vec{\mathtt{g}}$. The variables in $\vec{\mathtt{p}}$ are (potentially) read or written whereas those in $\vec{\mathtt{g}}$ are definitely unmodified by any execution of statement S. This is formalized as:

$$\left\{\, \boldsymbol{\lambda}\,(\vec{p},\vec{g})\bullet \mathtt{true} \,\right\} \mathtt{S}|_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}} \left\{\, \boldsymbol{\lambda}\,(\vec{p}\,',\vec{g}\,'),(\vec{p},\vec{g})\bullet \vec{g}=\vec{g}\,' \,\right\}. \quad (5)$$

When refactoring $\mathtt{S}|_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}$ into a new method $\mathtt{m}(\vec{\mathtt{p}})\{\ \mathtt{S}|_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}\ \}$, the variables $\vec{\mathtt{p}}$ are passed as parameters while the variables $\vec{\mathtt{g}}$ are global. It is always sound to $\subseteq$-over-approximate the set $\vec{\mathtt{p}}$ and $\subseteq$-under-approximate the set $\vec{\mathtt{g}}$.

We assume that the extracted method and the contracts are specified as follows:

```
private void m(p⃗) {                                   (6)
  Requires( P_R(p⃗) );
  Ensures( Q_R(OldValue(p⃗), p⃗) );
  Ensures( ∀x ∉ p⃗ : OldValue(x) == x );
  S|_{p⃗\g⃗}
}
```

where $\mathtt{OldValue}(\vec{\mathtt{p}})$ denotes the values of the actual parameters when calling method m.

The precondition $P_R(\vec{p})$ is checked when the method is called on the values $\vec{p}$ of the actual parameters $\vec{\mathtt{p}}$. The postcondition $Q_R(\vec{p},\vec{p}\,')$ relates the initial values $\vec{p}$ (denoted $\mathtt{OldValue}(\vec{\mathtt{p}})$ in (6)) of the parameters $\vec{\mathtt{p}}$ on method entry to their final values $\vec{p}\,'$ (denoted $\vec{\mathtt{p}}$ in (6)) on method exit. The postcondition is checked at runtime on exit. In the case of a contract failure, the execution halts. The fact that none of the variables other than $\vec{\mathtt{p}}$ can be modified by a method call is either specified explicitly, if allowed by contract specifications, or recorded together with the method contract, or else assumed implicitly. This assumption will be needed to guarantee the soundness of the separate method call proof rules of Sec. 7 and Sec. 11.

We let the set of all the *contracts for the method* m to be

$$\mathcal{C}[\![\mathtt{m}]\!] \ \triangleq\ \mathcal{P}[\![\vec{\mathtt{p}}]\!] \times \mathcal{P}[\![\vec{\mathtt{p}},\vec{\mathtt{p}}]\!]\ .$$

Definition 8. A contract $\langle P,\, Q\rangle \in \mathcal{C}[\![\mathtt{m}]\!]$ is a *valid* m contract if and only if $\left\{\, P \,\right\} \mathtt{S}|_{\vec{\mathtt{p}},\vec{\mathtt{g}}} \left\{\, Q \,\right\}$. □

In absence of valid contracts, we can always use the trivial $\overset{\text{cc}}{\mathtt{true}} \triangleq \langle \dot{\mathtt{true}},\ \dot{\mathtt{true}}\rangle$.

Safety pre-condition A property $P_{\mathtt{m}} \in \mathcal{P}[\![\vec{\mathtt{p}}]\!]$ is a *safety precondition* for a method "void m $\{\, \mathtt{S}|_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}} \,\}$" if and only if

$$\left\{\, \dot{\neg}\, P_{\mathtt{m}} \,\right\} \mathtt{S}|_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}} \left\{\, \dot{\mathtt{false}} \,\right\}.$$

Intuitively, if $P_{\mathtt{m}}$ does not hold then the execution of the method body $\mathtt{S}|_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}$ is doomed to fail either because of non-termination or because of a runtime error causes the program to stop.

By (2), we have $\mathrm{post}[\![\mathtt{S}|_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}]\!]\, \dot{\neg}\, P_{\mathtt{m}} \Longrightarrow \dot{\mathtt{false}}$, which, by (3), is equivalent to $\dot{\neg}\, \widetilde{\mathrm{pre}}[\![\mathtt{S}|_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}]\!]\dot{\mathtt{false}} \Longrightarrow P_{\mathtt{m}}$. In practice the strongest precondition $P^*_{\mathtt{m}} \triangleq \dot{\neg}\, \widetilde{\mathrm{pre}}[\![\mathtt{S}|_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}]\!]\dot{\mathtt{false}}$ is not computable and so will have to be over-approximated by a weaker precondition $P_{\mathtt{m}}$ such that $P^*_{\mathtt{m}} \Longrightarrow P_{\mathtt{m}}$. Any one of the backward static analyses in [18] can be used to effectively compute an abstract version of $P_{\mathtt{m}}$. It follows that $\dot{\neg}\, P_{\mathtt{m}}$ under-approximates $\dot{\neg}\, P^*_{\mathtt{m}}$ in that $\dot{\neg}\, P_{\mathtt{m}} \Longrightarrow \dot{\neg}\, P^*_{\mathtt{m}}$ and so, by ($\Rightarrow$), $\dot{\neg}\, P_{\mathtt{m}}$ satisfies $\left\{\, \dot{\neg}\, P_{\mathtt{m}} \,\right\} \mathtt{S}|_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}} \left\{\, \dot{\mathtt{false}} \,\right\}$.

Safety post-condition Once a safety pre-condition $P_{\mathtt{m}} \in \mathcal{P}[\![\vec{\mathtt{p}}]\!]$ has been inferred, a *safety post-condition* $Q_{\mathtt{m}} \in \mathcal{P}[\![\vec{\mathtt{p}},\vec{\mathtt{p}}]\!]$, relating the initial values $\vec{p}$ of the parameters $\vec{\mathtt{p}}$ and their final values $\vec{p}\,'$ must be inferred satisfying

$$\left\{\, P_{\mathtt{m}} \,\right\} \mathtt{S}|_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}} \left\{\, Q_{\mathtt{m}} \,\right\}$$

or equivalently, $P_{\mathtt{m}} \dot{\Longrightarrow} \widetilde{\mathrm{pre}}[\![\mathtt{S}|_{\vec{\mathtt{p}}\backslash\vec{\mathtt{g}}}]\!]Q_{\mathtt{m}}$. Again the strongest $Q_{\mathtt{m}}$ is not computable and so will have to be over-approximated in the abstract by a relational reachability analysis [11].

Safety contract The pair of a method safety pre-condition and post-condition yields a safety contract.

Definition 9. A method *safety contract* for the method "`void m ( p ) { S|p\g }`" is a pair $\langle P_{\mathtt{m}},\, Q_{\mathtt{m}}\rangle \in \mathcal{C}[\![\mathtt{m}]\!]$ such that

$$\left\{ \dot{\neg}\, P_{\mathtt{m}} \right\} \mathtt{S}|_{\vec{\mathtt{p}}\backslash\vec{\mathtt{g}}} \left\{ \dot{\mathtt{false}} \right\} \quad \text{and} \quad \left\{ P_{\mathtt{m}} \right\} \mathtt{S}|_{\vec{\mathtt{p}}\backslash\vec{\mathtt{g}}} \left\{ Q_{\mathtt{m}} \right\} . \ \square$$

The intuition is that either the safety pre-condition $P_{\mathtt{m}}$ does not hold and the method call is doomed to fail, so on exit of `S` (which never happens) $Q_{\mathtt{m}}$ does hold. Otherwise the safety pre-condition $P_{\mathtt{m}}$ does hold in which case the post-condition $Q_{\mathtt{m}}$ describes the effect of the call, if it ever terminates. In the abstract, over-approximations are inferred by the static analysis. By Def. 9, this abstract safety contract will always be valid but it may not be precise enough to ensure completeness or generality. For example, in absence of precise method safety contract, we can always choose $\langle P_{\mathtt{m}},\, Q_{\mathtt{m}}\rangle \triangleq \overset{\mathrm{cc}}{\dot{\mathtt{true}}}$.

Safety versus validity For contracts, validity and safety are two different concepts. Any safety contract is valid but some valid contracts may not be safe. For example $\left\{ x = 1 \right\}$ `x=1/x` $\left\{ x = 1 \right\}$ is valid but not safe since $\left\{ x \neq 1 \right\}$ `x=1/x` $\left\{$ `false` $\right\}$ does not hold. However $\left\{ x \neq 0 \right\}$ `x=1/x` $\left\{ x \neq 0 \right\}$ is safe hence valid.

Callee/covariant partial order on contracts We define the *callee/covariant partial order on concrete contracts*

$$\langle P,\, Q\rangle \overset{\dot{\times}}{\Longrightarrow} \langle P',\, Q'\rangle \ \triangleq\ P \dot{\Longrightarrow} P' \wedge Q \dot{\Longrightarrow} Q'.$$

The intuition is that stronger is better for the callee (assuming more on the precondition to guarantee more on the postcondition). The order $\overset{\dot{\times}}{\Longrightarrow}$ will be used in Sec. 8 to define the safety of the extracted method contract. The set of concrete contracts for method `m` is the complete lattice

$$\langle \mathcal{C}[\![\mathtt{m}]\!],\, \overset{\dot{\times}}{\Longrightarrow},\, \overset{\dot{\times}}{\bot},\, \overset{\dot{\times}}{\top},\, \overset{\dot{\times}}{\bigvee},\, \overset{\dot{\times}}{\bigwedge}\rangle$$

where $\overset{\dot{\times}}{\bot}$ is the infimum, $\overset{\dot{\times}}{\top}$ is the supremum, $\overset{\dot{\times}}{\bigvee}$ is the lub, and $\overset{\dot{\times}}{\bigwedge}$ is the glb for the partial order $\overset{\dot{\times}}{\Longrightarrow}$ on the set $\mathcal{C}[\![\mathtt{m}]\!]$.

Caller/contravariant partial order on contracts We define the *contract caller/contravariant partial order* $\overset{\mathrm{cc}}{\Longrightarrow}$ on $\mathcal{C}[\![\mathtt{m}]\!]$ as

$$\begin{aligned}\langle P,\, Q\rangle \overset{\mathrm{cc}}{\Longrightarrow} \langle P',\, Q'\rangle \ \triangleq\ & (P' \dot{\Longrightarrow} P) \wedge \\ & (\boldsymbol{\lambda}\, \vec{p}', \vec{p} \bullet P'(\vec{p}') \wedge Q(\vec{p}', \vec{p}) \dot{\Longrightarrow} Q')\ .\end{aligned} \tag{7}$$

The intuition behind this order is that a $\overset{\mathrm{cc}}{\Longrightarrow}$-stronger contract is more general and a $\overset{\mathrm{cc}}{\Longrightarrow}$-weaker contract is more specific, since if $\langle P,\, Q\rangle \overset{\mathrm{cc}}{\Longrightarrow} \langle P',\, Q'\rangle$ and $\left\{ P \right\} \mathtt{S} \left\{ Q \right\}$ hold then $\left\{ P' \right\} \mathtt{S} \left\{ Q' \right\}$ does hold. Concretely it means that from the caller point of view all proofs done with the contract $\langle P',\, Q'\rangle$ can also be done with $\langle P,\, Q\rangle$. This intuition is therefore that stronger is better for the caller (assuming less on precondition to get more on postcondition). The order $\overset{\mathrm{cc}}{\Longrightarrow}$ will be used in Sec. 8 to define the most general extracted method contracts.

The set $\langle \mathcal{C}[\![\mathtt{m}]\!],\, \overset{\mathrm{cc}}{\Longrightarrow},\, \overset{\mathrm{cc}}{\bot},\, \overset{\mathrm{cc}}{\top},\, \overset{\mathrm{cc}}{\bigvee},\, \overset{\mathrm{cc}}{\bigwedge}\rangle$ of all contracts for a method `m` is a complete lattice for partial order $\overset{\mathrm{cc}}{\Longrightarrow}$ where $\overset{\mathrm{cc}}{\bot} \triangleq \langle \dot{\mathtt{true}},\, \dot{\mathtt{false}}\rangle$ is the infimum, $\overset{\mathrm{cc}}{\top} \triangleq \langle \dot{\mathtt{false}},\, \dot{\mathtt{true}}\rangle$ is the supremum, the (infinitary) join is $\overset{\mathrm{cc}}{\bigvee\limits_{i\in\Delta}} \langle P_i,\, Q_i\rangle \triangleq \langle \dot{\bigwedge\limits_{i\in\Delta}} P_i,\, \dot{\bigvee\limits_{i\in\Delta}} Q_i\rangle$. The definition of $\overset{\mathrm{cc}}{\bigwedge}$ is dual.

7. Separate method verification

In order to formalize the problem of extract method with contracts, we need to reason about method calls. We now formalize what we mean by separate verification of the correctness of the callee and the method caller. We assume the simplifying hypotheses of Sec. 6 for the variables modified by a method call. In general, *e.g.*, to handle the heap or concurrency, more complex rules are needed to express the frame conditions. The problem is orthogonal to this paper, and so we assume sequential programs with only scalar variables.

Let $\mathtt{m}(\vec{\mathtt{p}})\{\ \mathtt{S}\ \}$ be a method definition with contract $\langle P, Q\rangle$ and let $\mathtt{m}(\vec{\mathtt{q}})$ be a method call where the actual parameters $\vec{\mathtt{q}}$ are variables such that $\vec{\mathcal{V}}[\![\vec{\mathtt{q}}]\!] = \vec{\mathcal{V}}[\![\vec{\mathtt{p}}]\!]$.

We define the separate method call proof rule. First, the contract $\langle P,\, Q\rangle$ of `m` should be valid, *i.e.*, $\left\{ P \right\} \mathtt{S}|_{\vec{\mathtt{p}}\backslash\vec{\mathtt{g}}} \left\{ Q \right\}$. Second, the call precondition P' should imply the method precondition P when projecting away the unmodified global variables: $\dot{\exists}\vec{\mathtt{g}} : \mathtt{P}' \dot{\Longrightarrow} \mathtt{P}$. If the two conditions hold then the caller can assume the postcondition Q:

$$\frac{\left\{ P \right\} \mathtt{S}|_{\vec{\mathtt{p}}\backslash\vec{\mathtt{g}}} \left\{ Q \right\}, \qquad \dot{\exists}\vec{\mathtt{g}} : P' \dot{\Longrightarrow} P}{\left\{ P' \right\} \mathtt{m}(\vec{\mathtt{q}}) \left\{ \boldsymbol{\lambda}\, ((\vec{q}, \vec{g}), (\vec{q}', \vec{g}')) \bullet Q(\vec{q}, \vec{q}') \right\}}\ . \tag{8}$$

As the global values $\vec{\mathtt{g}}$ are unaffected by the call, the information available on them before the call is still valid after the call:

$$\frac{\left\{ P \right\} \mathtt{S}|_{\vec{\mathtt{p}}\backslash\vec{\mathtt{g}}} \left\{ Q \right\}}{\left\{ P \right\} \mathtt{S} \left\{ \boldsymbol{\lambda}\, (\vec{p}, \vec{g}), (\vec{p}', \vec{g}') \bullet P(\vec{p}, \vec{g}) \wedge Q((\vec{p}, \vec{g}), (\vec{p}', \vec{g}')) \wedge \vec{g}' = \vec{g} \right\}}\ . \tag{9}$$

The two rules (8) and (9) can be combined via the conjunction rule $(\wedge)$ to provide the concrete separate method call proof rule.

8. Extract Method with Contracts

We devise a two-step algorithm for the extract method with contracts. The classical syntactic extract method is first applied to the user selection. If it succeeds (*e.g.*, a syntactically correct program is generated), we apply our algorithm $\overline{\mathrm{EMC}}$ in Alg. 5 to infer good contracts for the new method. In order to formalize (and solve!) the problem both in the concrete and in the abstract, we need first to make explicit the assumptions on the underlying syntactic refactoring engine and on the analysis. These assumptions are formulated in the concrete but should also hold in the abstract, up to concretization, as considered in Sec. 12.

Assumptions When the end-user selects a piece of code $\mathtt{S}$, the refactoring engine produces a new program with the refactored code only if this is a syntactically valid program. Otherwise stated, we rule out syntactically ill-formed programs. We only consider in-out parameters and procedures for simplicity, but we handle the general case in our implementation.

The new method appears in the same class of the selected code. The method is marked as `private` — so there is no need to ensure that the class invariant is preserved[4]. We assume the extracted method to be in the form of:

```
private void m(p⃗) {
Contract.Ensures(∀x ∉ p⃗ : Contract.OldValue(x) == x);
  S|_{p⃗\g⃗}
}.
```

We explicitly record in the contract which variables are neither read nor written by the method (otherwise the assumption remains implicit, or guaranteed by the semantics of the language, *e.g.*, for parameters of struct type).

At the call site, the selected code $\mathtt{S}|_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}$ is refactored into a method call $\mathtt{m}(\vec{\mathtt{p}})$, where $\vec{\mathtt{p}}$ is the vector of actual parameters.

We assume that a pre-invariant $P_{\mathtt{S}} \in \mathcal{P}[\![(\vec{\mathtt{p}},\vec{\mathtt{g}})]\!]$ and a post-invariant $Q_{\mathtt{S}} \in \mathcal{P}[\![(\vec{\mathtt{p}},\vec{\mathtt{g}}),(\vec{\mathtt{p}},\vec{\mathtt{g}})]\!]$ are available for the selected code $\mathtt{S}$ such that $\{\ P_{\mathtt{S}}\ \}\ \mathtt{S}\ \{\ Q_{\mathtt{S}}\ \}$. The pre-(post-)invariants can be derived by projecting the abstract state of the analyzer in the program point just before (after) $\mathtt{S}$ (formally followed by a concretization when reasoning in the concrete). Otherwise, it is always possible to use $\overset{cc}{\mathtt{true}}$. These assumptions can be summarized as

$$\{\ P_{\mathtt{S}}\ \}\ \mathtt{S}|_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}\ \{\ Q_{\mathtt{S}}\ \}\ .$$

The projection of $\langle P_{\mathtt{S}},\ Q_{\mathtt{S}}\rangle$ for $\mathtt{S}$ on the read/written variables $\vec{\mathtt{p}}$ is $\langle P_{\mathtt{S}}^{\curlyvee},\ Q_{\mathtt{S}}^{\curlyvee}\rangle$. It satisfies the following conditions:

$$\begin{array}{rcll} P_{\mathtt{S}}^{\curlyvee}(\vec{p}\,') & \triangleq & \exists \vec{g} \in \vec{\mathcal{V}}[\![\vec{\mathtt{g}}]\!] : P_{\mathtt{S}}(\vec{p}\,',\vec{g}) & \text{and} \\ Q_{\mathtt{S}}^{\curlyvee}(\vec{p}\,',\vec{p}) & \triangleq & \exists \vec{g}\,'' \in \vec{\mathcal{V}}[\![\vec{\mathtt{g}}]\!] : Q_{\mathtt{S}}((\vec{p}\,',\vec{g}\,''),(\vec{p},\vec{g}\,'')) \ . & (10) \end{array}$$

From what said above and (10), it immediately follows that the following triples are valid, stating that the extracted method does not modify the globals and that the projected pre- and post-invariants are still valid contracts:

$$\{\ \boldsymbol{\lambda}\,(\vec{p},\vec{g})\bullet \mathtt{true}\ \}\ \mathtt{m}(\vec{\mathtt{p}})\ \{\ \boldsymbol{\lambda}\,(\vec{p}\,',\vec{g}\,'),(\vec{p},\vec{g})\bullet \vec{g}=\vec{g}\,'\ \} \quad \text{and}$$
$$\{\ P_{\mathtt{S}}^{\curlyvee}\ \}\ \mathtt{S}|_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}\ \{\ Q_{\mathtt{S}}^{\curlyvee}\ \}\ .$$

We assume that a safety contract $\langle P_{\mathtt{m}},\ Q_{\mathtt{m}}\rangle$ (cf. Def. 9) for the extracted method $\mathtt{m}$ can be inferred by running an isolated analysis for $\mathtt{m}$ (formally followed by a concretization when reasoning in the concrete). At worst, $\overset{cc}{\mathtt{true}}$ is always a safe choice.

The problem of method extraction with contract (EMC) We want to generate a contract $\langle P_R,\ Q_R\rangle \in \mathcal{C}[\![\mathtt{m}]\!]$ for the (new) extracted method $\mathtt{m}$. The extracted method will then be analyzed separately (to prove its contract $\langle P_R,\ Q_R\rangle$ correct) and the contract $\langle P_R,\ Q_R\rangle$ will be used to derive the post-invariant $Q_{\mathtt{S}}$ from the pre-invariant $P_{\mathtt{S}}$ in a forward analysis of the method call (and/or the pre-invariant $P_{\mathtt{S}}$ from the post-invariant $Q_{\mathtt{S}}$ in case of backward analysis). The contract $\langle P_R,\ Q_R\rangle$ for extracted method $\mathtt{m}$ must guarantee that the proof/analysis that succeeded before the refactoring still succeeds after the refactoring.

Differently stated, the problem is to find an appropriate *refactored contract* $\langle P_R,\ Q_R\rangle$ with pre-condition P_R and post-condition Q_R of the form (6). We put the following requirements on this refactored contract $\langle P_R,\ Q_R\rangle$:

$(\mathbf{s})$ – **validity** Assuming the refactored contract pre-condition, the post-condition must hold. Formally:

$$\{\ P_R\ \}\ \mathtt{S}|_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}\ \{\ Q_R\ \}\ .$$

$(\mathbf{b})$ – **safety** The refactored contract $\langle P_R,\ Q_R\rangle$ is stronger than the method safety contract $\langle P_{\mathtt{m}},\ Q_{\mathtt{m}}\rangle$:

$$\langle P_R,\ Q_R\rangle \overset{\dot{\times}}{\Longrightarrow} \langle P_{\mathtt{m}},\ Q_{\mathtt{m}}\rangle\ .$$

The refactored contract requires more (so that P_R implies $P_{\mathtt{m}}$ which ensures the absence of runtime errors when executing the extracted method) and ensures more (so Q_R implies $Q_{\mathtt{m}}$ and so takes at least into account on method exit what can be learned from the method precondition $P_{\mathtt{m}}$ followed by the execution of the method body).

$(\mathbf{c})$ – **completeness** The refactored code is *still provable* with the same precision as the original code. The triple $\{P_{\mathtt{S}}\}\,\mathtt{m}(\vec{\mathtt{p}})\,\{Q_{\mathtt{S}}\}$ is provable by the separate method call proof rule (8) using the extracted method contract $\langle P_R,\ Q_R\rangle$.

$(\mathbf{d})$ – **generality** The refactored contract $\langle P_R,\ Q_R\rangle$ is the most general possible: the pre-condition of the refactored contract $\langle P_R,\ Q_R\rangle$ is the weakest possible (so that the extracted method applicability is as general as possible) and its post-condition is the strongest possible (so that calls to the extracted method get as much information as possible on its effect). However we do not consider type generalization [42], which is a separate problem.

Independent requirements The validity, safety, completeness and generality requirements are all mutually independent. For example, $\{\ \mathtt{false}\ \}\ \mathtt{S}\ \{\ \mathtt{true}\ \}$ is always safe, invalid for reachable code, validity for unreachable code but (in general) incomplete and not general.

Consequences We report some consequences of our requirements and definitions.

From the requirement (a) – validity and (8) it follows that the (opportunely instantiated) refactored contract is valid at the call site:

$$\begin{array}{l} \{\ \boldsymbol{\lambda}\,(\vec{q}\,',\vec{g}\,')\bullet\ P_R(\vec{q}\,')\ \}\mathtt{m}(\vec{\mathtt{q}}) \\ \qquad \{\ \boldsymbol{\lambda}\,((\vec{q}\,',\vec{g}\,'),(\vec{q},\vec{g}))\bullet Q_R(\vec{q}\,',\vec{q})\wedge\vec{g}=\vec{g}\,'\ \} \end{array} \qquad (11)$$

After refactoring, $\{\ P_{\mathtt{S}}\ \}\ \mathtt{m}(\vec{\mathtt{q}})\ \{\ Q_{\mathtt{S}}\ \}$ can be proved using (11) if and only if

[4] The situation is slightly different for `public` methods, and orthogonal to our problem.

$$\forall \vec{p}', \vec{g} : P_{\mathtt{S}}(\vec{p}', \vec{g}) \Longrightarrow P_R(\vec{p}') \qquad (12)$$
$$\forall \vec{p}', \vec{p}, \vec{g} : Q_R(\vec{p}', \vec{p}) \Longrightarrow Q_{\mathtt{S}}((\vec{p}', \vec{g}), (\vec{p}, \vec{g})) \ .$$

The conditions in (12) can be strengthened to take run-time errors into account. Although mathematically useless, this is useful to minimize the loss of information in abstract interpretation. Therefore, after refactoring, $\{\ P_{\mathtt{S}}\ \}\,\mathtt{m}(\vec{\mathtt{q}})\,\{\ Q_{\mathtt{S}}\ \}$ can be proved if and only if

$$\forall \vec{p}', \vec{g} : \big(P_{\mathtt{S}}(\vec{p}', \vec{g}) \wedge P_{\mathtt{m}}(\vec{p}')\big) \Longrightarrow P_R(\vec{p}') \qquad (13)$$

$$\forall \vec{p}', \vec{p}, \vec{g} : \big(P_{\mathtt{S}}(\vec{p}', \vec{g}) \wedge P_{\mathtt{m}}(\vec{p}') \wedge Q_R(\vec{p}', \vec{p})\big) \Longrightarrow \qquad (14)$$
$$Q_{\mathtt{S}}((\vec{p}', \vec{g}), (\vec{p}, \vec{g})) \ .$$

Please note that if the method pre-condition $P_{\mathtt{m}}$ does *not* hold, then the selected code S would have definitely failed on some language or programmer assertion while the refactored code will also definitely fail, but earlier, when calling method m. So it is possible that $P_{\mathtt{S}}(\vec{p}, \vec{g})$ does hold and the execution goes on (until definitely failing later somewhere within S) whereas $P_{\mathtt{m}}$ does not hold on method call so that execution just fails right on call. However, this changes nothing as far as the post-condition $Q_{\mathtt{S}}$ is concerned.

Finally, the most general contract refactoring requirement $(\mathrm{d})-\mathrm{generality}$ can be equivalently restated as

"if $\langle P'_R,\ Q'_R\rangle$ satisfies $(\mathrm{a})-\mathrm{validity}$, $(\mathrm{b})-\mathrm{safety}$, and $(\mathrm{c})-\mathrm{completeness}$, then $\langle P_R,\ Q_R\rangle \overset{cc}{\Longrightarrow} \langle P'_R,\ Q'_R\rangle$". (15)

9. Exact method refactoring

We show that the EMC problem has a unique solution, and we give two equivalent formulations of the solution. The first one is nicer from a mathematical point of view, but less suitable for abstraction. The second one involves a combination of backwards and forwards iterations, and it will be the base for our static analysis.

Concrete solution of EMC We devise a solution to EMC as follows. The precondition P_R for the method is the safety precondition $P_{\mathtt{m}}$ — all the internal safety checks are made explicit to the caller. The postcondition is the strongest postcondition from $P_{\mathtt{m}}$.

Theorem 10 (Exact contract refactoring)**.** The *unique* contract satisfying $(\mathrm{a})-\mathrm{validity}$, $(\mathrm{b})-\mathrm{safety}$, $(\mathrm{c})-\mathrm{completeness}$, and $(\mathrm{d})-\mathrm{generality}$ is:

$$\langle P_R,\ Q_R\rangle \ \triangleq\ \langle P_{\mathtt{m}},\ \mathrm{post}\,[\![\mathtt{S}|_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}]\!]P_{\mathtt{m}}\rangle. \qquad (16) \qquad \square$$

In an ideal world (*e.g.*, finite and small enough) where everything is exactly computable, EMC is very simple: compute the safety precondition and then propagate it forwards to get the postcondition (as in model checking).

In practice $\mathrm{post}\,[\![\mathtt{S}|_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}]\!]P_{\mathtt{m}}$ is not effectively computable — the set of states is infinite or extremely large. Therefore an approximation is needed — all the *fully* automatic static analysis methods for infinite state systems are necessarily approximate. An abstract version of Th. 10 is essentially useless: static analyses compute an over-approximation of post and this over-approximation may easily cause the requirement $(\mathrm{c})-\mathrm{completeness}$ not to be satisfied. We propose a solution to EMC *nicer* to abstract than (16). First we need to recall some facts on greatest fixpoints.

Greatest fixpoints We write $\mathrm{gfp}^{\sqsubseteq}_{a}\, f$ for the $\sqsubseteq$-greatest fixpoint of $f \in L \to L$ $\sqsubseteq$-less than or equal to $a \in L$, if any (*e.g.*, $\langle L,\ \sqsubseteq\rangle$ is a dual cpo, f is increasing and $a \in L$ is a post-fixpoint of f, *i.e.*, $f(a) \sqsubseteq a$). Otherwise, $\mathrm{gfp}^{\sqsubseteq}_{a}\, f$ is the limit, if any, of the iterates of $\boldsymbol{\lambda}\, x \bullet x \sqcap f(x)$ from a (which yield the same definition with the previous hypotheses), see [14].

Iterated solution of EMC We propose a solution to EMC based on the combination of a forward and a backward analysis, inspired by [8]. The idea is to compensate for the loss of information in the abstract by an iterated forward/backward analysis. Starting with the projection of the pre- and post-conditions $\langle P^{\curlyvee}_{\mathtt{S}}, Q^{\curlyvee}_{\mathtt{S}}\rangle$ at the original call site on the relevant variables, the contract is iteratively generalized by successive forward fixpoint propagations strengthening the postcondition and backwards fixpoint propagations weakening the precondition. The iteration of these fixpoint computations ultimately stabilize, in general after infinitely many decreasing iterations in the concrete, which we express as a greatest fixpoint (which is therefore a fixpoint of fixpoints).

The *method contract transformer* $F_R[\![\mathtt{S}]\!] \in \mathcal{C}[\![\mathtt{m}]\!] \to \mathcal{C}[\![\mathtt{m}]\!]$ refines the safety contract $\langle P_{\mathtt{m}}, Q_{\mathtt{m}}\rangle$ with the precondition and postcondition transformers:

$$F_R[\![\mathtt{S}]\!](\langle X,\ Y\rangle) \triangleq \langle P_{\mathtt{m}} \mathbin{\dot\wedge} \widetilde{\mathrm{pre}}[\![\mathtt{S}|_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}]\!]Y,\ Q_{\mathtt{m}} \mathbin{\dot\wedge} \mathrm{post}[\![\mathtt{S}|_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}]\!]X\rangle. \qquad (16)$$

Observe that $\widetilde{\mathrm{pre}}[\![\mathtt{S}|_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}]\!]Y$ and $\mathrm{post}[\![\mathtt{S}|_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}]\!]X$ both involve fixpoint computations [11, 13].

The fixpoint of the descending iterations of F_R from $\langle P^{\curlyvee}_{\mathtt{S}}, Q^{\curlyvee}_{\mathtt{S}}\rangle$ is the solution to EMC:

Theorem 11 (Iterated contract refactoring)**.** Under the assumptions of this paper,

$$\langle P_{\mathtt{m}},\ \mathrm{post}\,[\![\mathtt{S}|_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}]\!]P_{\mathtt{m}}\rangle \ =\ \mathrm{gfp}^{\overset{cc}{\Longrightarrow}}_{\langle P^{\curlyvee}_{\mathtt{S}},\ Q^{\curlyvee}_{\mathtt{S}}\rangle} F_R[\![\mathtt{S}]\!] \qquad (17)$$

and, by Th. 10, is the unique solution to $(\mathrm{a})-\mathrm{validity}$, $(\mathrm{b})-\mathrm{safety}$, $(\mathrm{c})-\mathrm{completeness}$, and $(\mathrm{d})-\mathrm{generality}$. $\square$

The fixpoint formulation of the solution to EMC, (17), is the concrete solution to our problem. As stated earlier, in the general case, the computation is unfeasible and we need to perform some approximation. Next we provide abstract counterparts to the separate method analysis rules of Sec. 7 and the formulation in the abstract of EMC.

10. Abstract Contracts

Abstract domain primitives In addition to the requirements of Sec. 5, we assume the precondition abstract domain A and the postcondition abstract domain B to define: (i) a predicate for the unchanged variables; (ii) an embedding from A to B; (iii) a variable projection; and, (iv) a variable

anti-projection. Please note that those assumptions are in no measure restrictive, we just make them explicit — all static analyzers implement those primitives, *e.g.*, the projection to remove variables when they go out of scope.

The predicate $\doteq[\![\cdot]\!]$ denotes the *unmodified* variables. Given a set of variables $\vec{\mathtt{g}} \subseteq \vec{\mathtt{v}}$, then $\doteq[\![\vec{\mathtt{g}}]\!] \in B[\![\vec{\mathtt{v}}, \vec{\mathtt{v}}]\!]$ is the abstract statement that none of the values of the variables $\vec{\mathtt{g}}$ has changed, that is

$$\gamma_2(\doteq[\![\vec{\mathtt{g}}]\!]) \triangleq \boldsymbol{\lambda}\, \vec{v}', \vec{v} \bullet \forall \mathtt{x} \in \vec{\mathtt{g}} : \vec{v}'(\mathtt{x}) = \vec{v}(\mathtt{x}).$$

The *embedding* $\uparrow_1^2 \in A[\![\vec{\mathtt{v}}]\!] \to B[\![\vec{\mathtt{v}}, \vec{\mathtt{v}}]\!]$ embeds unary predicates into binary predicates. It respects the soundness condition:

$$\forall \overline{P} \in A[\![\vec{\mathtt{v}}]\!] : \forall \vec{v}', \vec{v} \in \vec{\mathcal{V}}[\![\vec{\mathtt{v}}]\!] : \gamma_2(\uparrow_1^2(\overline{P}))(\vec{v}', \vec{v}) = \gamma_1(\overline{P})(\vec{v}')\ .$$

We assume that the embedding is increasing, *i.e.*, if $\overline{P} \sqsubseteq^{\boxed{1}} \overline{P}'$ then $\uparrow_1^2(\overline{P}) \sqsubseteq^{\boxed{2}} \uparrow_1^2(\overline{P}')$.

The *abstract projection* $\downarrow_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}$ projects onto the parameters and global variables. It satisfies the following soundness criteria: ($\overline{P} \in A[\![\vec{\mathtt{p}}, \vec{\mathtt{g}}]\!]$ and $\overline{Q} \in B[\![(\vec{\mathtt{p}}, \vec{\mathtt{g}}), (\vec{\mathtt{p}}, \vec{\mathtt{g}})]\!]$)

$$\begin{array}{rcl} \overline{P} & \sqsubseteq^{\boxed{1}} & \downarrow_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}(\overline{P}) \\ \left(\dot{\exists}\vec{\mathtt{g}} : \gamma_1(\overline{P})\vec{\mathtt{p}}, \vec{\mathtt{g}}\right) & = & \gamma_1(\downarrow_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}(\overline{P})) \\ \left(\dot{\exists}\vec{\mathtt{g}} : \gamma_2(\overline{Q})\right) & \dot{\Longrightarrow} & \gamma_2(\downarrow_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}(\overline{Q})) \\ \multicolumn{3}{c}{\downarrow_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}} \text{ is increasing}} \end{array}$$

Global variables can be reintroduced by the *abstract antiprojection* $\uparrow_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}} \in B[\![\vec{\mathtt{p}}, \vec{\mathtt{p}}]\!] \to B[\![(\vec{\mathtt{p}}, \vec{\mathtt{g}}), (\vec{\mathtt{p}}, \vec{\mathtt{g}})]\!]$. It satisfies the soundness requirement: ($\overline{Q} \in B[\![\vec{\mathtt{q}}, \vec{\mathtt{q}}]\!]$)

$$\boldsymbol{\lambda}\, ((\vec{q}', \vec{g}'), (\vec{q}, \vec{g})) \bullet \gamma_2(\overline{Q})(\vec{q}', \vec{q}) \wedge \vec{g} = \vec{g}' \dot{\Longrightarrow} \gamma_2\Big(\uparrow_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}(\overline{Q})\Big)\ .$$

In the following, we leave variable renaming implicit, identifying $A[\![\vec{\mathtt{p}}]\!]$ and $A[\![\vec{\mathtt{q}}]\!]$ whenever $\vec{\mathcal{V}}[\![\vec{\mathtt{q}}]\!] = \vec{\mathcal{V}}[\![\vec{\mathtt{p}}]\!]$.

Abstract contracts In analogy to what was done in Sec. 6 for the concrete contracts, we define *abstract* contracts and a covariant and contravariant order on those. An abstract contract is an element of $A[\![\vec{\mathtt{p}}]\!] \times B[\![\vec{\mathtt{p}}, \vec{\mathtt{p}}]\!]$.

The callee/covariant partial order on abstract contracts is defined as

$$\langle \overline{P},\, \overline{Q} \rangle \sqsubseteq^{\boxtimes} \langle \overline{P}',\, \overline{Q}' \rangle \ \triangleq\ \overline{P} \sqsubseteq^{\boxed{1}} \overline{P}' \wedge \overline{Q} \sqsubseteq^{\boxed{2}} \overline{Q}'\ .$$

The meaning is given by the concretization function $\gamma_\times \in \langle A[\![\vec{\mathtt{p}}]\!] \times B[\![\vec{\mathtt{p}}, \vec{\mathtt{p}}]\!],\, \sqsubseteq^{\boxtimes} \rangle \to \langle \mathcal{C}[\![\mathtt{m}]\!],\, \overset{\dot{\times}}{\Longrightarrow} \rangle$ defined as

$$\gamma_\times(\langle \overline{P}, \overline{Q} \rangle) \ \triangleq\ \langle \gamma_1(\overline{P}), \gamma_2(\overline{Q}) \rangle.$$

The caller/contravariant partial order on abstract contracts is defined as

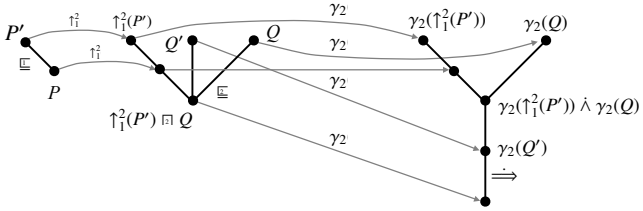
$$\langle \overline{P},\, \overline{Q} \rangle \overset{\text{cc}}{\sqsubseteq} \langle \overline{P}',\, \overline{Q}' \rangle \ \triangleq\ \overline{P}' \sqsubseteq^{\boxed{1}} \overline{P} \wedge \uparrow_1^2(\overline{P}') \sqsupseteq^{\boxed{2}} \overline{Q} \sqsubseteq^{\boxed{2}} \overline{Q}'\ .$$

The meaning is given by the concretization function $\gamma_{\text{cc}} \in \langle A[\![\vec{\mathtt{p}}]\!] \times B[\![\vec{\mathtt{p}}, \vec{\mathtt{p}}]\!],\, \overset{\text{cc}}{\sqsubseteq} \rangle \to \langle \mathcal{C}[\![\mathtt{m}]\!],\, \overset{\text{cc}}{\Longrightarrow} \rangle$ defined as

$$\gamma_{\text{cc}}\left(\langle \overline{P},\, \overline{Q} \rangle\right) \ \triangleq\ \langle \gamma_1(\overline{P}),\, \gamma_2(\overline{Q}) \rangle\ . \tag{18}$$

It is easy to observe that γ_{cc} is increasing. However, if γ_2 is increasing but not meet-preserving then the property that an abstract contract is more precise than another one may not be preserved in the concrete, as shown by the following counter-example.

Example 12. Let us consider the two abstract domains and the concrete domain below:



We have $P' \sqsubseteq^{\boxed{1}} P \wedge \uparrow_1^2(P') \sqsupseteq^{\boxed{2}} Q \sqsubseteq^{\boxed{2}} Q'$ and so $\langle P,\, Q \rangle \overset{\text{cc}}{\sqsubseteq} \langle P',\, Q' \rangle$ but not $\gamma_{\text{cc}}\left(\langle P,\, Q \rangle\right) \overset{\text{cc}}{\Longrightarrow} \gamma_{\text{cc}}\left(\langle P',\, Q' \rangle\right)$ since $\gamma_2(\uparrow_1^2(P')) \mathbin{\dot\wedge} \gamma_2(Q) \not\dot\Longrightarrow \gamma_2(Q')$. So even if in the abstract the contract $\langle P,\, Q \rangle$ is more precise that the contract $\langle P',\, Q \rangle$, this is not true in the concrete. So in this case, improving the precision of a contract in the abstract does not guarantee that the concretization also improves. □

We now have two reasons for assuming γ_2 to be finite-meet-preserving. One is to ensure the soundness of the abstract conjunction rule ($\overline{\wedge}$), *i.e.*, the reduced product (Th. 6) and the other to preserve the precision relation between abstract contracts in the concrete.

11. Abstract separate method analysis

We are now ready to formalize the rule in the abstract Hoare logic to handle the method call. We abstract the corresponding rules for method call of Sec. 7.

We obtain the *abstract separate method call analysis rule* by replacing the concrete Hoare triples, implications, projection *etc.* of (8) with their abstract counterparts defined above: ($\overline{P} \in A[\![\vec{\mathtt{p}}, \vec{\mathtt{g}}]\!]$, $\overline{P}' \in A[\![\vec{\mathtt{p}}]\!]$, and $\overline{Q} \in B[\![\vec{\mathtt{p}}, \vec{\mathtt{p}}]\!]$):

$$\frac{\overline{\{\!|}\, \overline{P}' \,\overline{|\!\}}\ \mathtt{S}|_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}\ \overline{\{\!|}\, \overline{Q} \,\overline{|\!\}}, \qquad \downarrow_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}(\overline{P}) \sqsubseteq^{\boxed{1}} \overline{P}'}{\overline{\{\!|}\, \overline{P} \,\overline{|\!\}}\ \mathtt{m}(\vec{\mathtt{q}})\ \overline{\{\!|}\, \uparrow_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}(\overline{Q}) \,\overline{|\!\}}}\ . \tag{19}$$

The abstract version of (9) propagates the properties of the unmodified variables $\vec{\mathtt{g}}$ through the call.

$$\frac{\overline{\{\!|}\, \overline{P} \,\overline{|\!\}}\ \mathtt{S}|_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}\ \overline{\{\!|}\, \overline{Q} \,\overline{|\!\}}}{\overline{\{\!|}\, \overline{P} \,\overline{|\!\}}\ \mathtt{S}|_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}\ \overline{\{\!|}\, \uparrow_1^2(\overline{P}) \sqsupseteq^{\boxed{2}} \overline{Q} \sqsupseteq^{\boxed{2}} \doteq[\![\vec{\mathtt{g}}]\!] \,\overline{|\!\}}}\ . \tag{20}$$

Theorem 13 (Soundness of the abstract separate method call analysis rule)**.** Abstract rules (19) and (20) are sound in the sense of (4). □

12. Extract Method with Abstract Contracts

We define the problem of the Approximate Extract Method with Contracts $\overline{\text{EMC}}$, by providing the abstract counterparts for the definitions and requirements of Sec. 8. We prove that the $\overline{\text{EMC}}$ implies EMC, but that the converse does not hold.

Assumptions The assumptions on code and invariant selection are similar to the concrete ones in Sec. 8, but now relative to abstract predicates.

We assume the variable decomposition is $\vec{\mathtt{v}} = \vec{\mathtt{p}}, \vec{\mathtt{g}}$ and $\overline{\{}\ \overline{P}_{\mathtt{S}}\ \overline{\}}\ \mathtt{S}|_{\vec{\mathtt{p}}\backslash\vec{\mathtt{g}}}\ \overline{\{}\ \overline{Q}_{\mathtt{S}}\ \overline{\}}$ so that the analysis of the selected code is sound. Those hypotheses essentially ensure the correctness of the code to be extracted. The next two hypotheses are completeness hypotheses requiring the abstraction to be expressive enough.

We assume that the post-condition of the selected code is strong enough in that $\overline{Q}_{\mathtt{S}} \mathrel{\overline{\sqsubseteq}} (\uparrow_1^2(\overline{P}_{\mathtt{S}}) \mathbin{\overline{?}} \doteqdot[\![\vec{\mathtt{g}}]\!])$. This implies that the information known on the initial values of the parameters and the fact that variables $\vec{\mathtt{g}}$ are unchanged is not lost, that is for all $\vec{p}', \vec{g}', \vec{p}, \vec{g}$,

$$\begin{array}{rcl}\gamma_2(\overline{Q}_{\mathtt{S}})((\vec{p}', \vec{g}'), (\vec{p}, \vec{g})) & \Longrightarrow & \gamma_1(\overline{P}_{\mathtt{S}})(\vec{p}', \vec{g}') \\ \gamma_2(\overline{Q}_{\mathtt{S}})((\vec{p}', \vec{g}'), (\vec{p}, \vec{g})) & \Longrightarrow & (\vec{g}' = \vec{g})\ .\end{array} \tag{21}$$

Furthermore, we assume that the analysis of the selected code is independent of the unread/unwritten variables, viz.

$$\left(\uparrow_{\vec{\mathtt{p}}\backslash\vec{\mathtt{g}}}\left(\downarrow_{\vec{\mathtt{p}}\backslash\vec{\mathtt{g}}}(\uparrow_1^2(\overline{P}_{\mathtt{S}}) \mathbin{\overline{?}} \overline{Q}_{\mathtt{S}} \mathbin{\overline{?}} \doteqdot[\![\vec{\mathtt{g}}]\!])\right) \mathbin{\overline{?}} \doteqdot[\![\vec{\mathtt{g}}]\!]\right) \mathrel{\overline{\sqsubseteq}} \overline{Q}_{\mathtt{S}} \tag{22}$$

The following contrived example shows why we need this hypothesis. Even if the selected code S does not depend upon the variables $\vec{\mathtt{g}}$ the analysis of this code might nevertheless depend upon these neither used nor modified variables $\vec{\mathtt{g}}$.

Example 14. Let us consider the following syntactic refactoring:

```
...                          ...
{ g in [11, 11] }  ⇝         { g in [11, 11] }
p = 0;                       NewMethod(p);
while (p < 10)               { p in [10, 11] }
   p = p + 1;                ...
{ p in [10, 11] }            private static void NewMethod (int p)
                             {
                                { Pr(p') = true }
                                  p = 0;
                                  while (p < 10) p = p + 1;
                                { Qr(p', p) = p in [10, +oo] }
                             }
```

We assume that the static analysis is an interval analysis with a widening using thresholds. The thresholds are assumed to be obtained by looking at all visible variables with a constant interval. So the analysis of the selected code uses the threshold 11 from the value of g while the analysis of the extracted method has no threshold at all so widen to $+\infty$. Then the method post-condition is too weak to prove the selected code post-invariant. Of course the constants of the program (*e.g.*, 10) could also be used as thresholds or a narrowing could improve the result but we assume that this is not the case in this contrived example. □

Finally, we assume that the abstract safety contract for the extracted method:

$$\langle \overline{P}_{\mathtt{m}}|_{\vec{\mathtt{p}}},\ \overline{Q}_{\mathtt{m}}|_{\vec{\mathtt{p}},\vec{\mathtt{p}}}\rangle \text{ is such that } \overline{\{}\ \overline{P}_{\mathtt{m}}\ \overline{\}}\ \mathtt{S}\ \overline{\{}\ \overline{Q}_{\mathtt{m}}\ \overline{\}}.$$

The safety precondition $\overline{P}_{\mathtt{m}}$ is first obtained by a backward analysis of the method body S [18] and then $\overline{Q}_{\mathtt{m}}$ is derived form $\overline{P}_{\mathtt{m}}$ by a forward reachability analysis of S [11].

Method extraction with abstract contracts, $\overline{\text{EMC}}$ We provide abstract counterparts for the requirements of EMC of Sec. 8. We call the problem $\overline{\text{EMC}}$.

$(\overline{\mathrm{a}})$ – $\overline{\textbf{validity}}$ The abstract refactored contract $\langle \overline{P}_R,\ \overline{Q}_R\rangle$ is valid: assuming the refactored abstract contract pre-condition, the post-condition must hold:

$$\overline{\{}\ \overline{P}_R\ \overline{\}}\ \mathtt{S}|_{\vec{\mathtt{p}}\backslash\vec{\mathtt{g}}}\ \overline{\{}\ \overline{Q}_R\ \overline{\}}\ .$$

$(\overline{\mathrm{b}})$ – $\overline{\textbf{safety}}$ The abstract refactored contract $\langle \overline{P}_R,\ \overline{Q}_R\rangle$ is stronger than the abstract method safety contract $\langle \overline{P}_{\mathtt{m}},\ \overline{Q}_{\mathtt{m}}\rangle$: $\overline{P}_{\mathtt{m}}$ is a necessary but possibly not sufficient condition for the absence of run-time error when the method is called [18]. $\overline{Q}_{\mathtt{m}}$ over-approximates the post-condition resulting from the execution of the method body assuming $\overline{P}_{\mathtt{m}}$ on entry. The abstract refactored contract requires more, so that $\overline{P}_R$ implies $\overline{P}_{\mathtt{m}}$ which is necessary (but possibly not sufficient) for the absence of runtime errors when executing the extracted method. The abstract refactored contracts ensures more, so $\overline{Q}_R$ implies $\overline{Q}_{\mathtt{m}}$. It takes at least into account on method exit what can be learned in the abstract from the method pre-condition $\overline{P}_{\mathtt{m}}$ followed by the execution of the method body, which can be summarized by:

$$\langle \overline{P}_R,\ \overline{Q}_R\rangle \mathrel{\overline{\boxtimes}} \langle \overline{P}_{\mathtt{m}},\ \overline{Q}_{\mathtt{m}}\rangle\ . \tag{23}$$

$(\overline{\mathrm{c}})$ – $\overline{\textbf{completeness}}$ The refactored code is *still provable* in the abstract with the same precision as the original code: $\overline{\{}\ \overline{P}_{\mathtt{S}}\ \overline{\}}\ \mathtt{m}(\vec{\mathtt{p}})\ \overline{\{}\ \overline{Q}_{\mathtt{S}}\ \overline{\}}$ is provable by the abstract separate method call analysis rule of Th. 13 using the extracted method abstract contract $\langle \overline{P}_R,\ \overline{Q}_R\rangle$.

$(\overline{\mathrm{d}})$ – $\overline{\textbf{generality}}$ *Optionally*, the abstract refactored contract $\langle \overline{P}_R,\ \overline{Q}_R\rangle$ is the most general: The pre-condition of the refactored contract $\langle \overline{P}_R,\ \overline{Q}_R\rangle$ is the weakest possible (so that the extracted method applicability is as general as possible) and its post-condition is the strongest possible (so that calls to the extracted method get as much information as possible on its effect) for the considered abstract domains. It can be shown that if $\langle \overline{P}'_R,\ \overline{Q}'_R\rangle$ satisfies requirements $(\overline{\mathrm{a}})$ – $\overline{\text{validity}}$, $(\overline{\mathrm{b}})$ – $\overline{\text{safety}}$, and $(\overline{\mathrm{c}})$ – $\overline{\text{completeness}}$ then $\langle \overline{P}_R,\ \overline{Q}_R\rangle \mathrel{\overline{\overset{\text{cc}}{\sqsubseteq}}} \langle \overline{P}'_R,\ \overline{Q}'_R\rangle$.

Theorem 15 (Correctness of the abstract requirements)**.** The abstract requirements $(\overline{\mathrm{a}})$ – $\overline{\text{validity}}$, $(\overline{\mathrm{b}})$ – $\overline{\text{safety}}$, and $(\overline{\mathrm{c}})$ – $\overline{\text{completeness}}$ respectively imply the concrete requirements (a) – validity, (b) – safety, and (c) – completeness for the concretization of the abstract predicates. Therefore, by Th. 13, method extraction with abstract contracts is sound. □

Notice that Th. 15 does not state that abstract completeness implies concrete completeness for any concrete contract. It states that abstract completeness implies concrete completeness for the concretization of abstract contracts. So it should be understood as meaning that properties of abstract contracts hold in the concrete up to their concretization. The intuition is that the separate method call analysis rule is more

powerful in the concrete than in the abstract. Of course, some concrete contracts are not the concretization of any abstract contract and Th. 15 states nothing on these contracts. Th. 10 and 11 are stronger than Th. 15 since Th. 15 is only valid in the concrete for concrete contracts expressible in the abstract without any loss of information while Th. 10 and 11 hold for any concrete contract.

Impossibility of complete abstract refactoring Approximations introduce new difficulties. In practice, the abstract requirement $(\overline{c})$ – $\overline{\text{completeness}}$ can only be optional — the concretization of the best abstract refactored contract, if any, might not be the best concrete refactored contract considered in (c) – completeness. The following counter-example proves that abstract refactoring is necessarily incomplete.

Example 16 (Impossibility of complete abstract refactoring, I)**.** Consider the following refactoring

```
       ...
       { Ps(p, g) = (g == 0) }
       while (1) p = 0;
       { Qs(p, g, p', g') = (g == g' == 7) }
⇝      ...
       { Ps(p, g) = (g == 0) }
       NewMethod(p);
       { Qs(p, g, p', g') = (g' == g+1 == 7) }
       ...
       private static void NewMethod (int p)
       {
          { Pr(p) = true }
          while (1) p = 0;
          { Qr(p, p') = (p' == p-2 == 17) }
       }
```

The loop does not terminate so the exit invariant is `false` which is over-approximated by $Q_{\mathtt{S}}(p,g,p',g') \triangleq (g = g' = 7)$ in the original code and by $Q_R(p,p') \triangleq (p' = p - 2 = 17)$ in the extracted method. $Q_{\mathtt{S}}$ and Q_R are a perfectly correct partial-correctness invariants/post-conditions since $\dot{\mathtt{false}} \dot{\Longrightarrow} Q_{\mathtt{S}}$ and $\dot{\mathtt{false}} \dot{\Longrightarrow} Q_R$. However, assuming $P_{\mathtt{S}}(p,g) \triangleq (g = 0)$ and $Q_R(p,p') \triangleq (p' = p - 2 = 17)$, and ignoring the method body, it is impossible to prove that $Q_{\mathtt{S}}(p,g,p',g') \triangleq (g = g' = 7)$ does hold. This proves that *abstract* refactoring is necessarily incomplete (since termination is undecidable). Please note that this is not in contradiction with the fact that there is no problem (except incomputability) with *exact* refactoring, since the method body exact post-condition Q_R shall be $\dot{\mathtt{false}}$. □

Example 17 (Impossibility of complete abstract refactoring, II)**.** Consider the following situation where the selected code S does not read or modify g.

```
       ...
       { Ps(p, g) = (g == 10) }
       while (1) do { p = p };
       { Qs(p, g, p', g') = ( g == g' == 1 ) }
⇝      ...
       { Ps(p, g) = (g == 10) }
       NewMethod(p);
       { Qs(p, g, p', g') = ( g == g' == 1 ) }
       ...
       private static void NewMethod (int p)
       {
          { Pr(p) = true }
          while (1) do { p = p };
          { Qr(p, p') = true }
       }
```

The separate analysis of the extracted method cannot prove the post-condition $Q_{\mathtt{S}}$ despite the fact that $Q_R(p',p) \Longrightarrow \exists g,g' : Q_{\mathtt{S}}((p,g),(p',g'))$ (choose $g = g' = 1$). The problem comes from the fact that `false`, which is the strongest post-condition for the selected code, was over-approximated by $Q_{\mathtt{S}}(p,g,p',g') \triangleq (g = g' = 1)$ and by $Q_R(p,p') \triangleq$ `true` after the body of the refactored procedure. Since g' is not available in the procedure body, it is impossible to make the same over-approximation of `false` in the method body as it was done in the selected code.

The counter-example is based on the fact that, in case of non-termination, since $Q_{\mathtt{S}}$ can state properties of g which are completely different form those stated by $P_{\mathtt{S}}$ although g is not modified by the loop body. This situation can hardly happen in practice since abstract transformers and widening/-narrowing will leave g abstract properties unchanged since the selected code neither reads nor writes g. □

Examples 16 and 17 are based on non-termination, in which case `false` can be approximated differently in the selected and refactored code.

13. Approximate iterated method refactoring

We want a static analysis to effectively solve $\overline{\text{EMC}}$. The main idea is to abstract the iterated exact refactoring of Th. 11. We see under which hypotheses the computed solution is the best one and how we can derive an approximated solution.

Initial state When the user selects a piece of code S, the underlying static analyzer extracts a pair $\langle \overline{P}_{\mathtt{S}}, \overline{Q}_{\mathtt{S}} \rangle$ containing the pre-state and the post-state for S. The pre-state (resp. the post-state) is the semantic information known to the analyzer at the program point just before (resp. after) the selected code:

$$P_{\mathtt{S}} \triangleq \gamma_1(\overline{P}_{\mathtt{S}}) \quad \text{and} \quad Q_{\mathtt{S}} \triangleq \gamma_2(\overline{Q}_{\mathtt{S}}). \tag{24}$$

The pre-state and the post-state are projected onto the parameters of the extracted method:

$$\overline{P}^{\curlyvee}_{\mathtt{S}} \triangleq \downarrow_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}(\overline{P}_{\mathtt{S}}) \quad \text{and} \quad \overline{Q}^{\curlyvee}_{\mathtt{S}} \triangleq \downarrow_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}(\overline{Q}_{\mathtt{S}}) \tag{25}$$

The initial abstract state for the (greatest) fixpoint computation soundly approximates the initial concrete state:

Lemma 18. Equations (25) and (24) imply that $\langle P^{\curlyvee}_{\mathtt{S}},\ Q^{\curlyvee}_{\mathtt{S}} \rangle \overset{cc}{\Longrightarrow} \gamma_{cc}\ (\langle \overline{P}^{\curlyvee}_{\mathtt{S}},\ \overline{Q}^{\curlyvee}_{\mathtt{S}} \rangle)$.

The underlying static analyzer may infer an approximated safety condition $\overline{P}_{\mathtt{m}}$, in which case we let $P_{\mathtt{m}} \triangleq \gamma_1(\overline{P}_{\mathtt{m}})$. Otherwise we assume $P_{\mathtt{m}}$ to be the strongest safety precondition. In both cases $Q_{\mathtt{m}} = \text{post}\,[\![\mathtt{S}_{\restriction_{\vec{\mathtt{p}}}}]\!]P_{\mathtt{m}}$ is the corresponding strongest relational post-condition.

Abstract transformer An abstract contract transformer $\overline{F}_R[\![\mathtt{S}]\!]$ has to be designed that soundly overapproximates the concrete contract transformer (16). The specification of $\overline{F}_R[\![\mathtt{S}]\!]$ is therefore:

$$F_R[\![\mathtt{S}]\!] \circ \gamma_{cc} \quad \dot{\overset{cc}{\Longrightarrow}} \quad \gamma_{cc} \circ \overline{F}_R[\![\mathtt{S}]\!] \tag{26}$$

In practice this means that we have, for a program statement S, either a forward static analysis, or a backwards static

analysis, or, preferably, both of them. Knowing the concrete transformer $F_R[\![\mathtt{S}]\!]$ defined by structural induction on $\mathtt{S}$, the design of an abstract transformer $\overline{F}_R[\![\mathtt{S}]\!]$ is classical in abstract interpretation [11].

Best iterated solution The iterations of $\overline{F}_R[\![\mathtt{S}]\!]$ provide a sound appoximation of the concrete fixpoint:

Theorem 19. Equations (25) and (26) imply that

$$\mathrm{gfp}^{\overset{\mathrm{cc}}{\Longrightarrow}}_{\langle P_{\mathtt{S}}^{\curlyvee},\, Q_{\mathtt{S}}^{\curlyvee}\rangle} F_R[\![\mathtt{S}]\!] \quad \overset{\mathrm{cc}}{\Longrightarrow} \quad \gamma_{\mathrm{cc}}\left(\mathrm{gfp}^{\overset{\mathrm{cc}}{\sqsubseteq}}_{\langle \overline{P}_{\mathtt{S}}^{\curlyvee},\, \overline{Q}_{\mathtt{S}}^{\curlyvee}\rangle} \overline{F}_R[\![\mathtt{S}]\!]\right). \qquad \square$$

In general $\overline{F}_R[\![\mathtt{S}]\!]$ may be any over-approximation of $F_R[\![\mathtt{S}]\!]$. Therefore it may not ensure that

$$\mathrm{gfp}^{\overset{\mathrm{cc}}{\sqsubseteq}}_{\langle \overline{P}_{\mathtt{S}}^{\curlyvee},\, \overline{Q}_{\mathtt{S}}^{\curlyvee}\rangle} \overline{F}_R[\![\mathtt{S}]\!] \mathrel{\overline{\boxtimes}} \langle \dot{\top},\, \overline{Q}_{\mathtt{m}}\rangle,$$

i.e., it does not satisfy the abstract requirement $(\overline{\mathrm{b}})-\overline{\mathrm{safety}}$. In order to guarantee that the limit of the iterations of the abstract transformer is a correct solution to $\overline{\mathrm{EMC}}$ we need the additional requirement:

$$\forall\langle X,\, Y\rangle : \overline{F}_R[\![\mathtt{S}]\!](\langle X,\, Y\rangle) \mathrel{\overline{\boxtimes}} \langle \dot{\top},\, \overline{Q}_{\mathtt{m}}\rangle \qquad (27)$$

This requirement can always be met by refining a given abstract transformer $\overline{F}_R$ such that the postcondition is no weaker than $Q_{\mathtt{m}}$:

$$\boldsymbol{\lambda}\,\langle X,\, Y\rangle \bullet \overline{F}_R[\![\mathtt{S}]\!](\langle X,\, Y\rangle) \overset{\dot{\times}}{\sqcap} \langle \dot{\top},\, \overline{Q}_{\mathtt{m}}\rangle. \qquad (28)$$

With the extra requirement (27), the iterative application of $\overline{F}_R$ from $\langle \overline{P}_{\mathtt{S}}^{\curlyvee},\, \overline{Q}_{\mathtt{S}}^{\curlyvee}\rangle$ provides a correct solution to $\overline{\mathrm{EMC}}$:

Theorem 20. Let $\overline{F}_R$ be an abstract transformer satisfying (25), (26) and (27). Then

$$\langle \overline{\overline{P}}_R,\, \overline{\overline{Q}}_R\rangle \quad \triangleq \quad \mathrm{gfp}^{\overset{\mathrm{cc}}{\sqsubseteq}}_{\langle \overline{P}_{\mathtt{S}}^{\curlyvee},\, \overline{Q}_{\mathtt{S}}^{\curlyvee}\rangle} \overline{F}_R[\![\mathtt{S}]\!] \qquad (29)$$

satisfies the abstract requirements $(\overline{\mathrm{a}})-\overline{\mathrm{validity}}$, $(\overline{\mathrm{b}})-\overline{\mathrm{safety}}$, and $(\overline{\mathrm{c}})-\overline{\mathrm{completeness}}$. $\square$

Equation (29) ensures that $\langle \overline{\overline{P}}_R,\, \overline{\overline{Q}}_R\rangle \overset{\mathrm{cc}}{\sqsubseteq} \langle \overline{P}_{\mathtt{S}}^{\curlyvee},\, \overline{Q}_{\mathtt{S}}^{\curlyvee}\rangle$, *i.e.*, the result of the fixpoint computation is a more precise contract than the trivial solution consisting of projecting the pre-state and post-state of the selected code.

Most general abstract contract refactoring In general the abstract refactoring $\langle \overline{\overline{P}}_R,\, \overline{\overline{Q}}_R\rangle$ in Th. 20 is not the most precise abstract contract refactoring — the abstract requirement $(\overline{\mathrm{d}})-\overline{\mathrm{generality}}$ does not hold in general. There are three possible reasons for that.

First, there is no most precise abstraction of the concrete solution of Th. 10 or 11 for $\overset{\mathrm{cc}}{\sqsubseteq}$ in Def. (18) of γ_{cc} a case illustrated in Fig. 4. This can be remedied, *e.g.*, by requiring: (i) γ_{cc} to be the upper-adjoint of a Galois connection — equivalently γ_{cc} is a complete meet morphism; and (ii) the α is to be surjective — equivalently γ_{cc} is injective — to avoid a redundant representation in the abstract of the same concrete property:

$$\langle \mathcal{P}[\![\vec{\mathrm{v}}]\!],\, \dot{\Longrightarrow}\rangle \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \langle A[\![\vec{\mathrm{v}}]\!],\, \sqsubseteq\rangle$$

$$\langle \mathcal{P}[\![\vec{\mathrm{v}},\vec{\mathrm{v}}]\!],\, \dot{\Longrightarrow}\rangle \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \langle B[\![\vec{\mathrm{v}},\vec{\mathrm{v}}]\!],\, \sqsubseteq\rangle$$

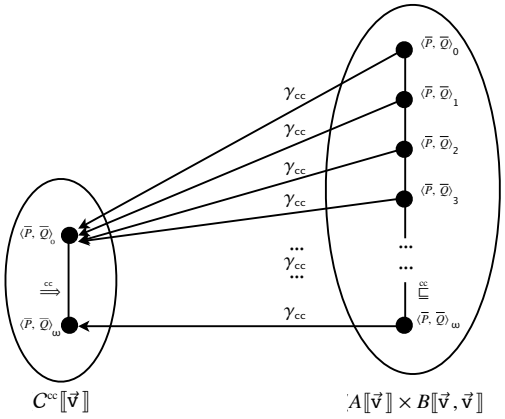


Figure 4. Absence of most precise abstraction of the concrete contract.

Second, the abstract transformer $\overline{F}_R[\![\mathtt{S}]\!]$ satisfying (26) might not be the most precise one. This situation can be avoided by requiring the abstract transformer $\overline{F}_R[\![\mathtt{S}]\!]$ to be the most precise abstract transformer, in which case (26) must be strengthened into:

$$F_R[\![\mathtt{S}]\!] \circ \gamma_{\mathrm{cc}} \quad = \quad \gamma_{\mathrm{cc}} \circ \overline{F}_R[\![\mathtt{S}]\!] \qquad (30)$$

Third, the abstract projection $\langle \overline{P}_{\mathtt{S}}^{\curlyvee},\, \overline{Q}_{\mathtt{S}}^{\curlyvee}\rangle$ (cf. (25)) might be too approximated. This can be excluded by requiring the abstract projection $\downarrow_{\vec{\mathrm{p}}\setminus\vec{\mathrm{g}}}$ to be the most precise possible.

Theorem 21 (Most precise abstract contract refactoring)**.** Under the hypotheses of Th. 20 and of this subsection (including (30)), $\langle \overline{\overline{P}}_R,\, \overline{\overline{Q}}_R\rangle$ also satisfies the abstract requirement $(\overline{\mathrm{d}})-\overline{\mathrm{generality}}$. $\square$

The best abstract transformer condition of (30) is rarely met in practice. A consequence is that the abstract requirement $(\overline{\mathrm{d}})-\overline{\mathrm{generality}}$ is mostly of theoretical interest. However, experience in Sec. 14 shows that abstract completeness is achieved in many cases.

Iterated Solution with Convergence Accelerators The underlying abstract domains A, B may not satisfy the Ascending/Descending chain conditions. As a consequence a *narrowing operator* [11] should be used to enforce the convergence of the greatest fixpoint computation of (29) to an (over-) approximate solution $\langle \overline{P}_R, \overline{Q}_R\rangle$. The greatest fixpoint iteration with narrowing ensures that

$$\langle \overline{\overline{P}}_R,\, \overline{\overline{Q}}_R\rangle \overset{\mathrm{cc}}{\sqsubseteq} \langle \overline{P}_R,\, \overline{Q}_R\rangle \overset{\mathrm{cc}}{\sqsubseteq} \langle \overline{P}_{\mathtt{S}}^{\curlyvee},\, \overline{Q}_{\mathtt{S}}^{\curlyvee}\rangle\,. \qquad (31)$$

We need to prove that $\langle \overline{P}_R, \overline{Q}_R\rangle$ is effectively a solution of $\overline{\mathrm{EMC}}$, and therefore it can be used in practice. This is guaranteed by the following theorem:

Theorem 22 (Correctness of the approximate abstract contract refactoring)**.** In addition to the hypotheses of Th. 20, let $\langle \overline{P}_R,\, \overline{Q}_R\rangle$ be satisfying (31) and $\overline{Q}_R \mathrel{\overline{\sqsubseteq}} \overline{Q}_{\mathtt{m}}$. Then $\langle \overline{P}_R,\, \overline{Q}_R\rangle$ satisfies the abstract requirements $(\overline{\mathrm{a}})-\overline{\mathrm{validity}}$ and $(\overline{\mathrm{b}})-\overline{\mathrm{safety}}$. $\square$

/* $\overline{P}_{\mathtt{S}}$: pre-state, S:refactored code, $\vec{\mathtt{p}}$: variables potentially used in S, $\vec{\mathtt{g}}$: variables definitely unmodified by S, $\overline{Q}_{\mathtt{S}}$:post-state such that $\overline{\{}\ \overline{P}_{\mathtt{S}}\ \overline{\}}\,\mathtt{S}|_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}\,\overline{\{}\ \overline{Q}_{\mathtt{S}}\ \overline{\}}$ holds. */

RefactorContract($\overline{P}_{\mathtt{S}}$, S, $\vec{\mathtt{p}}$, $\vec{\mathtt{g}}$, $\overline{Q}_{\mathtt{S}}$) {

use $\langle A[\![\vec{\mathtt{p}}]\!], \boxed{\sqcap}, \Delta_1\rangle$ // precondition abstract domain
$\langle B[\![\vec{\mathtt{p}}, \vec{\mathtt{p}}]\!], \boxed{\sqsupset}, \Delta_2\rangle$ // postcondition abstract domain
$\overline{\mathrm{post}}$ // forward analyser with widening/narrowing
$\overline{\widetilde{\mathrm{pre}}}$ // backward analyser with widening/narrowing

// abstract projection on potentially used variables $\vec{\mathtt{p}}$
$\langle \overline{P}^{\curlyvee}_{\mathtt{S}}, \overline{Q}^{\curlyvee}_{\mathtt{S}}\rangle = \langle \downarrow_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}(\overline{P}_{\mathtt{S}}), \downarrow_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}(\overline{Q}_{\mathtt{S}})\rangle$;
// infer a correct safety abstract contract
Let $\overline{P}_{\mathtt{m}}$ be the abstract safety pre-condition for S computed by the static analysis [18];
$\overline{Q}_{\mathtt{m}} = \overline{\mathrm{post}}[\![\mathtt{S}\restriction_{\vec{\mathtt{p}}}]\!]\overline{P}_{\mathtt{m}}$; // forward abstract static analysis
// $\overline{\{}\ \overline{P}_{\mathtt{m}}\ \overline{\}}\,\mathtt{S}|_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}\,\overline{\{}\ \overline{Q}_{\mathtt{m}}\ \overline{\}}$ holds
$\langle \overline{P}_R, \overline{Q}_R\rangle = \langle \overline{P}^{\curlyvee}_{\mathtt{S}}, \overline{Q}^{\curlyvee}_{\mathtt{S}}\rangle$;
do
// compute $\langle X, Y\rangle = \overline{F}_R[\![\mathtt{S}]\!](\langle \overline{P}_R, \overline{Q}_R\rangle)$
$X = \overline{P}_{\mathtt{m}} \boxed{\sqcap} \overline{P}_R \boxed{\sqcap} \overline{\widetilde{\mathrm{pre}}}[\![\mathtt{S}\restriction_{\vec{\mathtt{p}}}]\!]\overline{Q}_R$; // backward analysis
$Y = \overline{Q}_{\mathtt{m}} \boxed{\sqsupset} \overline{Q}_R \boxed{\sqsupset} \overline{\mathrm{post}}[\![\mathtt{S}\restriction_{\vec{\mathtt{p}}}]\!]\overline{P}_R$; // forward analysis
$\langle \overline{P}_R, \overline{Q}_R\rangle = \langle \overline{P}_R\ \Delta_1\ X, \overline{Q}_R\ \Delta_2\ Y\rangle$; // narrowing
while $\langle \overline{P}_R, \overline{Q}_R\rangle \neq \langle X, Y\rangle$;
// $\mathrm{gfp}^{\overset{\mathrm{cc}}{\sqsubseteq}}_{\langle \overline{P}^{\curlyvee}_{\mathtt{s}}, \overline{Q}^{\curlyvee}_{\mathtt{s}}\rangle} \overline{F}_R[\![\mathtt{S}]\!] \overset{\mathrm{cc}}{\sqsubseteq} \langle \overline{P}_R, \overline{Q}_R\rangle \overset{\mathrm{cc}}{\sqsubseteq} \langle \overline{P}^{\curlyvee}_{\mathtt{S}}, \overline{Q}^{\curlyvee}_{\mathtt{S}}\rangle$ holds
return $\langle \overline{P}_R, \overline{Q}_R\rangle$; // $(\bar{\mathrm{a}}) - \overline{\mathrm{validity}}$ & $(\bar{\mathrm{b}}) - \overline{\mathrm{safety}}$ hold
}

Algorithm 5. Algorithm $\overline{\mathrm{EMC}}$ (Extract Methods with Abstract Contracts) computing an approximation of a greatest fixpoint with convergence acceleration.

Th. 22 states that all the abstract contracts included between the best solution (29) and the abstract projections of the abstract states are a solution of our problem. A natural way to compute $\langle \overline{P}_R, \overline{Q}_R\rangle$ is to perform the downwards iterates of $\overline{F}_R[\![\mathtt{S}]\!]$ from $\langle \overline{P}^{\curlyvee}_{\mathtt{S}}, \overline{Q}^{\curlyvee}_{\mathtt{S}}\rangle$ with narrowing. The algorithm $\overline{\mathrm{EMC}}$ is given in Alg. 5. An optimization using chaotic iterations with memory [8] would have

$Y = \overline{Q}_{\mathtt{m}} \boxed{\sqsupset} \overline{Q}_R \boxed{\sqsupset} \overline{\mathrm{post}}[\![\mathtt{S}\restriction_{\vec{\mathtt{p}}}]\!]X$; // forward analysis

This is the solution we implemented, with more details given in the next section.

14. Experience

The underlying tools We implemented the algorithms of the previous section on top of two industrial-strength tools, `Roslyn` and `CCCheck`.

`Roslyn` exposes the (C# and VB) compiler internals (syntax trees, object model, data-flow analyses, refactoring, etc.) to external developers, so that they can develp new plugins (code analyses, refactorings) on top of it.

`CCCheck` is a static contract verifier for CodeContracts. It analyzes each method in isolation, assuming the precondition and asserting the postcondition. `CCCheck` can also do backward analyses to infer a precondition from the postcondition. `CCCheck` is based on abstract interpretation and hence has more advanced inference capabilities than similar tools. For instance, it infers loop invariants and it suggests method preconditions and postconditions (the $\langle \overline{P}_{\mathtt{m}}, \overline{Q}_{\mathtt{m}}\rangle$ in this paper). `CCCheck` contains several abstract domains for the heap, non-nullness, numerical properties, array contents, enums, but also to track (simple) existential and quantified properties [20]. Most of these abstract domains use widenings so completeness cannot be guaranteed in the theoretical sense for tortuous counter-examples and the contracts cannot technically be the most general. The benchmarks ran with the default settings show that the inferred contracts can hardly be improved manually for the abstraction used by the static analyzer.

The implementation We preferred not to implement ourselves the syntactic extract method from scratch. We used `Roslyn`, which takes care of both the user interface (*e.g.*, code selection, right click, previews, etc.) and the basic refactorings. Furthermore, we did not wanted to try our examples on toy implementations or abstract domains, hence we (modified and) used `CCCheck` to implement the $\overline{\mathrm{EMC}}$ algorithm. `CCCheck` runs as a background service in `Roslyn`. While `Roslyn` provides syntactic, source-level, ASTs, `CCCheck` analyzes bytecode. Therefore there is some (non-trivial) glue code connecting the two.

The extract method with contracts is implemented as a Visual Studio extension for C#. When the user selects a piece of code `S`, `Roslyn` in the background (and concurrently), invokes the extension asking it to provide a refactoring, if any. Our extension first forwards the call to the refactoring engine of `Roslyn`. If no method is extracted from the selection (*e.g.*, not all the branches of `S` are terminated by a `return` statement), the extraction fails, and we stop there. If the extraction succeeds, then we generate a contract for the new method.

The *first step* of the algorithm $\overline{\mathrm{EMC}}$ is to deduce $\langle \overline{P}^{\curlyvee}_{\mathtt{S}}, \overline{Q}^{\curlyvee}_{\mathtt{S}}\rangle$, the starting point for the greatest fixpoint computation. In theory, this information can be obtained by fetching the program points corresponding to the user selection, and then asking `CCCheck` for the corresponding invariants and `Roslyn` for $\mathtt{S}|_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}$. Unfortunately there are some practical issues that complicate the theoretical schema. First, `CCCheck` does not keep an explicit map from source locations to bytecode offsets, but only the inverse map, used to report warnings and suggestions. Second, for memory consumption reasons, `CCCheck` throws away the inferred invariants once it is done with the

analysis of a method. So at the time the refactoring is invoked, that information is already gone. Third, because of the heap analysis, the mapping between source level variables and internal variables used by the abstract domains in `CCCheck` is pretty complex (*e.g.*, the same syntactic variable may have different internal names at different program points). Luckily, the refactoring engine of `Roslyn` indirectly provides the partition $\langle \vec{\mathtt{p}}, \vec{\mathtt{g}} \rangle$ and the information on modified variables via the parameters. Roughly, the actual parameters are the variables read/written in `S`, and the actual parameters passed by `ref` are those that may be modified in `S` and whose value may be used in the callers. Our solution is then to use two dummy method calls as markers for the precondition and the postcondition, inserted, respectively, at the beginning and at the end of the selection. The first marker, the precondition marker, is a fresh method call whose actual parameters are the variables in $\vec{\mathtt{p}}$ that can be modified inside `S`. For the other variables in $\vec{\mathtt{p}}$, we have the guarantee that their value either does not change or does not affect the method on return (*i.e.*, they are dead variables). The second marker, the postcondition marker, is a fresh method call whose actual parameters are as above plus an extra one denoting the return value.

Example 23 (Markers)**.** For the initial example in Sec. 2, the annotated code is:

```
__PreconditionMarker();
 while (x != 0)  x--;
__PostconditionMarker(x, true);
```

The Boolean flag indicates whether or not the next-to-last parameter is the variable the return value is assigned to (refactoring may generate *void* methods, in which case the flag is `false`). When the Boolean flag is set, then all the occurrences of the next-to-last variable in $\overline{Q}_{\mathtt{S}}^{\curlyvee}$ are replaced by `Contract.Result`, *i.e.*, the return value of the method is made explicit in the postcondition. □

We then analyze the annotated method with a switch to trigger the generation of $\langle \overline{P}_{\mathtt{S}}^{\curlyvee}, \overline{Q}_{\mathtt{S}}^{\curlyvee} \rangle$: `CCCheck` analyzes the method, collects at the marked points $\langle \overline{P}_{\mathtt{S}}, \overline{Q}_{\mathtt{S}} \rangle$, and then uses the actual parameters to project them onto the variables of interest to emit $\langle \overline{P}_{\mathtt{S}}^{\curlyvee}, \overline{Q}_{\mathtt{S}}^{\curlyvee} \rangle$.

The *second step* of the algorithm, inserts $\langle \overline{P}_{\mathtt{S}}, \overline{Q}_{\mathtt{S}} \rangle$ for the extracted method, and then runs `CCCheck` to infer $\langle \overline{P}_{\mathtt{m}}, \overline{Q}_{\mathtt{m}} \rangle$.

In the *third step*, we add $\langle \overline{P}_{\mathtt{m}}, \overline{Q}_{\mathtt{m}} \rangle$ (to enforce (27)) to the extracted method and we iterate the forward/backwards schema until we reach a fixpoint, or we run out of stamina, in which case we return the current approximation — this never happened in our experience, though.

Finally, we instrument the extracted method with $\langle \overline{P}_{\mathtt{R}}, \overline{Q}_{\mathtt{R}} \rangle$, and we propose it as a refactoring to the user, *e.g.*, Fig. 1.

Benchmarks It is very hard, if not impossible, to evaluate automatically the effect of the extract method, as it depends on user interaction. A random selection of `S` is not very meaningful either. It is very likely to generate ill-formed programs, and it may not be representative of the effective use. Furthermore, in order to evaluate our analysis, we should first fix *what* we evaluate. Our goal is to have the extract method with contracts integrated in a continuous verification (or semantic) IDE. As such, two metrics are relevant: (i) performance (the analysis should happen in real time); and (ii) precision and generality of the results (no new warning should be introduced, and the result should be as general as possible). We evaluated those two aspects on some benchmarks (randomly) extracted from the `CCCheck` regression suite. The `CCCheck` regression test suite contains many corner cases and small, yet tricky, bug repros reported by users in order to stress the analyzer.

We report the experimental results in Fig. 6. The first column is the name of the test. The second column contains the time required for `Roslyn` to extract the method. The third column is the cost of step one (inference of $\langle \overline{P}_{\mathtt{S}}^{\curlyvee}, \overline{Q}_{\mathtt{S}}^{\curlyvee} \rangle$) and the fourth column is the combined cost of steps 2 and 3 (inference of $\langle \overline{P}_{\mathtt{m}}, \overline{Q}_{\mathtt{m}} \rangle$ and $\langle \overline{P}_{R}, \overline{Q}_{R} \rangle$). The last column is the total time taken by the extract with contracts refactoring. Note that the total is slightly larger than the sum of the other three columns because it also includes the cost of annotating the syntax trees, context switching, etc., due to multithreading. The tests are not very long *per se*, but rather complex, as can be noticed by the raw time spent by the optimized refactoring engine to perform the syntactic method extraction. In general, the cost of our analysis is comparable with that of the extract method alone. In most of the cases, the total time remains well below one second, meeting the first requirement (real time). The only real slow-down is in the Loop-2 test, which is caused by the overhead of using exceptions as control flow in the analysis for certain corner cases. This idiom causes an extreme slowdown while running with the debugger attached, which was the easiest way for us to record the timings. Without the debugger attached, the wall-clock time improved dramatically.

In all of the tests we succeeded in extracting a contract which was both precise enough to not break the verification of the caller and general enough to be used elsewhere. We were positively impressed by the inferred invariants. For instance, for BeyerEtAl (Fig. 1 of [5]), we selected the body of the loop. The extract method with contracts was able to infer the right pre- and post-conditions (`3*i = a + b`), generalizing it for non-negative values of `i`, `a`, and `b` but also restraining `i` to be less than $2^{31} - 1$ (otherwise an overflow may occurr).

In the PeronHalbwachs example — computing the max of an array (Fig. 1(a) of [28]):

```
int Max(int[] a) {
 Requires(a != null && a.Length > 0);
 Ensures(ForAll(0, a.Length, j => Result<int>() <= a[j]));
 Ensures(Exists(0, a.Length, j => Result<int>() == a[j]));
 var max = a[0];
 for(var i = 1; i < a.Length; i++)
   if(a[i] > max) max = a[i];
 return max; }
```

`CCCheck` infers the loop invariant $\forall j \in [0, \mathtt{i}).\mathtt{a}[j] \leq \mathtt{max}$ and $\exists j \in [0, \mathtt{i}).\mathtt{a}[j] = \mathtt{max}$, and uses it to prove the postcondition.

Test	Extraction	Step 1	Steps 2/3	Total
Decrement	0.18	0.10	0.12	0.42
Generalize	0.20	0.09	0.14	0.45
BinarySearch	0.23	0.14	0.32	0.70
Abs	0.23	0.07	0.12	0.43
Arithmetic	0.20	0.07	0.28	0.56
Rem	0.20	0.09	0.20	0.49
Guard	0.17	0.07	0.14	0.40
Loop	0.18	0.07	0.10	0.37
Exp	0.34	0.18	0.24	0.79
Main	0.20	0.14	0.20	0.56
Karr	0.35	0.09	0.14	0.71
Loop-2	0.28	0.18	1.99	2.43
Loop-3	0.21	0.10	0.14	0.46
SankaEtAl [40]	0.24	0.09	0.00	0.35
McMillan [33]	0.24	0.18	0.43	0.93
BeyerEtAl [5]	0.34	0.18	0.28	0.82
PeronHalbwachs [28]	0.47	0.33	0.31	1.13

Figure 6. The experimental results (in seconds). The additional cost is of the same order of magnitude as the syntactic method extraction. The precision was good enough in all tests to preserve the verification of the caller and generalize the precondition of the extracted method.

In the benchmark, we selected the body of the loop, and got the following contract:

```
Requires(a != null && 0 <= i && i < a.Length);
Requires(Exists(0, a.Length, j => max == a[j]));
Ensures(Exists(0, a.Length, j => Result<int>() == a[j]));
Ensures(a[i] <= Result<int>());
Ensures(max <= Result<int>());
```

This is the most general contract possible for `CCCheck` abstract domains, which are not disjunctive. On entry, the array `a` should be non-null, the index `i` should be in its bounds and `max` should be equal to some element in `a`. On exit, the returned value is an array element, larger than both `max` and `a[i]`. The precondition of the extracted method is proven in the refactored `Max`. The postcondition is used to infer the same loop invariant as in the original `Max`. Note that the correctness proof of `Max` does not need a stronger contract with universally quantified invariants. The analysis is smart enough to *automatically* deduce it. Otherwise, the postcondition `ForAll(0, i, j => a[j] <= Result<int>())` would have generated the precondition `ForAll(0, i-1, j => a[j] <= max)`, a requirement that would be too strong on the caller, dramatically reducing the admissible calling contexts.

15. Related work

Program source to source transformation and supporting tools were very popular in the late 70's and early 80's ([32, 41], to cite a few). The research subject went out of fashion probably because program transformation systems needed too large catalogues of transformation rules that were hard to master in batch mode by programmers and transformation enabling conditions ensuring the correctness/incorrectness of program behavior preservation/refinement were hard to prove (either manually or automatically) [1]. The subject rose from the ashes through *code refactoring* [24, 34, 43], a computer-aided reorganization of the code preserving its behavior (and hopefully improving its readability, modifiability and maintainability). Program transformation and refactoring look very similar in particular since they share similar catalogues of transformations. However, the correctness of refactoring transformations (so-called "semantic preservation") is ultimately left over the shoulders of programmers, not on the refactoring tools, which is therefore never falling short nor faulty.

Various formalizations of refactoring have been proposed in the concrete such as [3, 25, 31]. Refactoring can also be formalized as a special case of semantic program transformation [15]. [26] consider the problem of merging similar classes (but not method extraction). To the best of our knowledge, we are the first to address the refactoring problem in the general context of abstract semantics and abstract proof preservation, not only types [42]. The problem of unsoundness of the conjunction rule in Hoare logic was already shown by John Reynolds in the context of concurrent Separation Logic [38]. However, the proposed solution was tied to the particular logic (enforcing the resources to be precise). We give a more general characterization, in terms of abstract Hoare Logic, and a general solution (Th. 6) to the problem.

16. Further Work

Static analysis might be used to help the end-user select the code to be extracted. She might be proposed to extend the code to include initializations, or exclude code irrelevant to the computed result.

Static analysis would also be useful to automatically check whether the extracted method preserves the class invariant, in which case the end-user may be offered the wider choice of declaring the extracted method either as `public` or `private`. Alternatively, the class invariant may be weakened automatically for `public` methods. A extracted method can also be placed outside the class of the refactored code if and only if it does not depend on the state of the objects of that class.

Static analysis could also help in generalizing the parameter types to the most general types that preserve the extracted method semantics. Of course semantic smelling and cloning would then be useful to help identify clones of the method body in the code that could be replaced by procedure calls.

17. Conclusions

Method refactoring is very useful in everyday practice. In the design by contract programming methodology, this must be accompanied by providing a contract for the new method. We have given conditions on this contract to ensure that the verification of the original program is not broken when refactoring the code. We have provided an exact solution to the undecidable problem which is not computable. This led to an iterated forward-backward abstract interpretation to automat-

ically compute an approximate solution. The implementation using `Roslyn` and `CCCheck` shows that the proposed algorithm is fast and precise enough in practice to entirely support the verification task of the programmer during design time.

Acknowledgments Peter O'Hearn and Mooly Sagiv for pointing out that the conjunction rule is unsound in concurrent separation logic (Ex. 7). We thanks the anonymous referees who proposed significant presentation improvements. This material is based upon work supported by the National Science Foundation under Grant No. 0926166.

References.

[1] J. Arsac. Syntactic source to source transforms and program manipulation. *Comm. ACM*, 22(1):43–54, 1979.

[2] A. Banerjee, D. A. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *ECOOP*, pp. 387–411, 2008.

[3] F. Bannwart and P. Müller. Changing programs correctly: Refactoring with specifications. In *FM 2006*, volume 4085 of *LNCS*, pp. 492–507, 2006.

[4] M. Barnett, M. Fähndrich, and F. Logozzo. Embedded contract languages. In *SAC'10*, pp. 2103–2110. ACM, 2010.

[5] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *PLDI*, pp. 300–309, 2007.

[6] S. Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270, 2007.

[7] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *STTT*, 7(3):212–232, 2005.

[8] P. Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes (in French)*. Thèse d'État ès sciences mathématiques, Université scientifique et médicale de Grenoble, 1978.

[9] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *TCS*, 277(1–2): 47–103, 2002.

[10] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. Second Int. Symp. on Programming*, pp. 106–130. Dunod, Paris, France, 1976.

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

[12] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E. Neuhold, editor, *IFIP Conf. on Formal Description of Programming Concepts*, pp. 237–277. North-Holland, 1977.

[13] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, pp. 269–282, 1979.

[14] P. Cousot and R. Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific J. Math.*, 82(1):43–57, 1979.

[15] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *POPL*, pp. 178–190, 2002.

[16] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pp. 84–97, 1978.

[17] P. Cousot, R. Cousot, and L.Mauborgne. The reduced product of abstract domains and the combination of decision procedures. In *FOSSACS*, pp. 456–472, 2011.

[18] P. Cousot, R. Cousot, and F. Logozzo. Contract precondition inference from intermittent assertions on collections. In *VMCAI*, pp. 150–168, 2011.

[19] E. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *CACM*, 18(8):453–457, 1975.

[20] M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS*, pp. 10–30, 2010.

[21] A. Feldthaus, T. D. Millstein, A. Møller, M. Schäfer, and F. Tip. Tool-supported refactoring for JavaScript. In *OOPSLA*, pp. 119–138, 2011.

[22] J.-C. Filliâtre and M. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV*, pp. 173–177, 2007.

[23] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pp. 234–245, 2002.

[24] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[25] A. Garrido and J. Meseguer. Formal specification and verification of Java refactorings. Technical Report UIUCDCS-R-2006-2731, Univ. of Illinois at Urbana-Champaign, 2006.

[26] M. Goldstein, Y. Feldman, and S. Tyszberowicz. Refactoring with contracts. In *AGILE*, pp. 53–64. IEEE Computer Society, 2006.

[27] A. Gotsman, J. Berdine, and B. Cook. Precision and the conjunction rule in concurrent separation logic. *Electr. Notes Theor. Comput. Sci.*, 276:171–190, 2011.

[28] N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *PLDI*, pp. 339–348, 2008.

[29] C. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[30] V. Laviron and F. Logozzo. Subpolyhedra: a family of numerical abstract domains for the (more) scalable inference of linear inequalities. *STTT*, 13(6):585–601, 2011.

[31] Q. Long, J. He, and Z. Liu. Refactoring and pattern-directed refactoring: A formal perspective. UNU-IIST Research Report 318, The United Nations Univ., 2005.

[32] D. Loveman. Program improvement by source-to-source transformation. *Journal of the ACM*, 24(1):121–145, 1977.

[33] K. L. McMillan. Relevance heuristics for program analysis. In *POPL*, pp. 145–146, 2008.

[34] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Trans. Software Eng.*, 30(2):126–139, 2004.

[35] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1991.

[36] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19:31–100, 2006.

[37] K. Ng, M. Warren, P. Golde, and A. Hejlsberg. The Roslyn Project, Exposing the C# and VB compiler's code analysis. `http://msdn.microsoft.com/en-us/roslyn`, 2011.

[38] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

[39] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL*, pp. 268–280, 2004.

[40] S. Sankaranarayanan, F. Ivancic, and A. Gupta. Program analysis using symbolic ranges. In *SAS*, pp. 366–383, 2007.

[41] T. Standish, D. Kibler and J. Neighbors. Improving and refining programs by program manipulation. In *ACMNC*, pp. 509–516, 1976.

[42] F. Tip. Refactoring using type constraints. In *SAS*, pp. 1–17, 2007.

[43] W. Wake. *Refactoring Workbook*. Addison-Wesley, 2003.