Inference of Necessary Field Conditions with Abstract Interpretation

Mehdi Bouaziz [1], Francesco Logozzo [2], Manuel Fähndrich [2]

[1] École Normale Supérieure, Paris
[2] Microsoft Research, Redmond, WA (USA)

Abstract. We present a new static analysis to infer necessary field conditions for object-oriented programs. A necessary field condition is a property that should hold on the fields of a given object, for otherwise there exists a calling context leading to a failure due to bad object state. Our analysis also infers the provenance of the necessary condition, so that if a necessary field condition is violated then an explanation containing the sequence of method calls leading to a failing assertion can be produced.
When the analysis is restricted to readonly fields, i.e., fields that can only be set in the initialization phase of an object, it infers object invariants. We provide empirical evidence on the usefulness of necessary field conditions by integrating the analysis into `cccheck`, our static analyzer for .NET. Robust inference of readonly object field invariants was the #1 request from `cccheck` users.

1 Introduction

Design by Contract [24] is a programming methodology which systematically requires the programmer to provide the preconditions, postconditions, and object invariants (collectively called contracts) at design time. Contracts allow the automatic generation of documentation, amplify the testing process, and naturally enable assume/guarantee reasoning for static program verification.

Assume/guarantee reasoning is a divide and conquer methodology, where the correctness proof is split between the callee and the caller. When the body of the callee is analyzed, its precondition is assumed and the postcondition must be proved. At a call site, the precondition must be proved and the postcondition can be assumed by the caller. In object-oriented programs, an *object invariant* (sometimes incorrectly called class invariant) is a property on the object fields that holds in the *steady* states of the object, i.e., it is at the same time a precondition and a postcondition of all the methods of a class.

In a perfect (Design by Contract) world, the programmer provides contracts for all the methods and all the classes, and a static verifier would leverage them to prove the program correctness. In the real world, relatively few classes and methods have contracts, for various reasons. First, the programming language or the programming environment may not support contracts at all. Programmers may add checks on the input parameters of a method and on the object fields,

but there is no systematic way of expressing those in a way that can be exploited by a static analyzer to perform assume/guarantee reasoning. Second, even if the programming environment supports contracts, programmers may have only partially annotated their code, for instance by adding preconditions only to the externally visible (public) methods, ignoring object fields. Third, the programmer may also have avoided adding contracts which may appear evident from the code, e.g., setting a private field to a non-null value in the constructor and never changing it again. Fourth, the provided contracts may be too weak, for instance a stronger object invariant may be needed to ensure the absence of errors such as runtime failures or assertion violations.

Inference has been advocated as the holy grail to solve the problems above. Ideally, an automatic static analyzer will infer preconditions and postconditions for each method, and object invariants for each class, exploiting the existing annotations and parameter checking code to get more precise results. The inferred contracts will then be propagated and used in the assume/guarantee reasoning.

Much research has been conducted to characterize *when* the object invariant can be assumed and when it should be checked, e.g. [11]. Orthogonally, some static analyses have been developed to infer object invariants when those points are known, e.g. [23, 4]. Those analyses over-approximate the strongest object invariant which in turn over-approximates the trace-based object semantics [23].

In this paper we tackle the problem of inferring *necessary* conditions on object invariants, i.e., conditions on object fields that should hold, for otherwise, there exists an execution trace starting with an object construction and a series of method invocations that leads to an assertion failure in one of the object's methods due to bad object state. Necessary object invariants differ from "usual" programmer-written object invariants in that they typically under-approximate the object invariant. Necessary object invariants are *necessary* in the sense that if they don't hold, there exists an execution trace that is guaranteed to fail. Satisfying all necessary object invariants on the other hand does not guarantee the absence of failures, due to, e.g., method internal non-determinism.

Main contributions. We discuss and define the problem of brittleness of class-level modular analyses (Sect. 2) – solving that problem was the #1 request of `cccheck` [15] users[3]. We introduce a solution to the problem based on the inference of necessary field conditions (Sect. 4). Our solution builds on the top of previous work on precondition inference [8]. We show that when readonly fields are concerned, necessary conditions are object invariants (Sect. 5). We validate our analysis on large and complex libraries (Sect. 6). We compare it with: (i) a baseline run (BR) where no properties for object fields are inferred (only method preconditions and postconditions); and (ii) an optimized implementation of the class-level modular analysis (CLMA). Experimental results show that our analysis: (i) introduces a modest slowdown (and in some cases a speedup!) over BR; (ii) is up to 2× faster than CLMA; and (iii) induces a precision improvement comparable to CLMA. To the best of our knowledge we are the first to system-

[3] `cccheck` is the abstract interpretation-based [7] contract static checker for CodeContracts [13]. At the moment of writing it counts more than 75,000 external downloads.

```
public class Person {
  private readonly string Name;
  private readonly JobTitle JobTitle;

  public Person(string name, JobTitle jobTitle)  {
    Contract.Requires(jobTitle != null && name != null);

    this.Name = name;
    this.JobTitle = jobTitle;
  }

  public string GetFullName()  {
    if (this.JobTitle != null)
      return string.Format("{0}␣({1})", PrettyPrint(this.Name), this.JobTitle.ToString());
    return PrettyPrint(this.Name);
  }

  public int BaseSalary()  {
    return this.JobTitle.BaseSalary;
  }

  public string PrettyPrint(string s)  {
    Contract.Requires(s != null);
    // ...
  }
}
```

Fig. 1. The running example. Without the object invariant `this.Name != null` ∧ `this.JobTitle != null` a modular static analyzer issues two false warnings.

atically evaluate and compare different approaches to the static inference of field conditions. In the light of the experimental results, we have chosen to deploy it as the standard analysis for object-invariant inference in `cccheck`.

2 Motivation

Let us consider the code in Fig. 1. A `Person` object contains two private fields, the name of the person and his job title. The C# `readonly` marker specifies that those fields can *only* be assigned inside the constructors. The method `GetFullName` returns the name of the person and the job title, if any. The method `BaseSalary` returns the base salary for the job title of the given person.

2.1 Separate Method-Level Analysis

A method-level modular analysis of `Person` will analyze each constructor/method in isolation. At the entry point, it will assume the precondition and the object invariant. At the exit point, it will assert the postcondition and the object invariant. In our simple example the object invariant is empty (trivially the `true` invariant). The analysis reports 2 possible null dereferences. In the method `GetFullName`, the field `Name` may be null, violating the precondition of `PrettyPrint`. In the method `BaseSalary` the field `JobTitle` may be null (in `GetFullName`, `JobTitle` is checked before being dereferenced). Those are *false* warnings as both fields are initialized to non-null values in the constructor. The readonly fields semantics guarantees that they cannot be modified anymore.

The usual solution to this problem is to ask the programmer to provide the object invariant. This is for instance the approach of tools like ESC/Java [18, 3], Krakatoa [16], Spec# [1], and the default behavior of `cccheck` [15]. The advantage of programmer-provided invariants is that they clearly state programmer

intent and can be used as documentation. The drawback is that too many annotations may be required, quickly overwhelming the programmer. Our goal is to help the programmer by inferring (or suggesting in the IDE) object invariants.

2.2 Class-Level Modular Analysis

A class-level modular analysis [22] exploits the information that the methods are executed after the constructor to improve precision. The main insight is that an object invariant can be given a fixpoint characterization as follows:

$$I = \bigsqcup_{\mathtt{c} \in Constrs} \mathsf{s}[\![\mathtt{c}]\!] \sqcup \bigsqcup_{\mathtt{m} \in Methods} \mathsf{s}[\![\mathtt{m}]\!](I), \tag{1}$$

i.e., an object invariant is a property that holds after the execution of the constructors and before and after the execution of the public methods[4]. Once we fix an abstract domain A and the corresponding static analysis $\bar{\mathsf{s}}[\![\cdot]\!]$, equation (1) can be solved with standard fixpoint techniques. A widening operator may be required to enforce the convergence of the iterations.

In our example, we let A be the non-nullness abstract domain [14]. At the first iteration, the analysis infers that `JobTitle` and `Name` are not null at the exit of the constructor: $I_0 = \langle \mathtt{Name} \mapsto \mathsf{NN}, \mathtt{JobTitle} \mapsto \mathsf{NN} \rangle$. The analysis then propagates I_0 to the entry point of the two methods: $\bar{\mathsf{s}}[\![\mathtt{GetFullName}]\!](I_0) = \bar{\mathsf{s}}[\![\mathtt{BaseSalary}]\!](I_0) = I_0$ so that I_0 is an object invariant. It is easy to see that I_0 is the *strongest* state-based invariant for `Person`. Using the invariant I_0, a static analyzer can prove the safety of all the field dereferences in the class.

2.3 Drawbacks of Class-Level Modular Analysis

In general a class-level modular analysis is brittle with respect to source modifications. A small change in one part of a class may cause a warning in a distant (apparently) totally unrelated part of the same class, causing major confusion for the programmer. We identify two main sources of brittleness. The first source arises from adding new members (constructors, methods) to a class. The second source arises from changes to the allowed initial method states and object initialization.

Addition of New Class Members: Suppose that a new constructor is added to the class in our example as follows:

```
public Person(string name) {
  Contract.Requires(name != null);

  this.Name = name; }
```

Let us call the modified class `Person′`. The field `JobTitle` is not assigned in the constructor, and it gets the default value `null`. The C# compiler does not issue any warning: it is perfectly legal to *not* assign a field (even if marked as readonly) in the constructor. The class-level modular analysis now considers the

[4] Here for simplicity we omit the treatment of aliasing, of inheritance, of the projection operators, and of method calls. The interested reader can find the extension of (1) for the treatment of those features in [23].

two constructors, and hence two ways of initializing the object. The object state after constructor invocation is:

$$J_0^0 = \bar{\mathsf{s}}[\![\texttt{Person(string, JobTitle)}]\!] = \langle \texttt{Name} \mapsto \mathsf{NN}, \texttt{JobTitle} \mapsto \mathsf{NN} \rangle$$
$$J_0^1 = \bar{\mathsf{s}}[\![\texttt{Person(string)}]\!] = \langle \texttt{Name} \mapsto \mathsf{NN}, \texttt{JobTitle} \mapsto \mathsf{T} \rangle$$
$$J_0 = J_0^0 \sqcup J_0^1 = J_0^1.$$

The analysis of the methods yields:

$$J_1^0 = \bar{\mathsf{s}}[\![\texttt{GetFullName}]\!](J_0) = \langle \texttt{Name} \mapsto \mathsf{NN}, \texttt{JobTitle} \mapsto \mathsf{T} \rangle$$
$$J_1^1 = \bar{\mathsf{s}}[\![\texttt{BaseSalary}]\!](J_0) = \langle \texttt{Name} \mapsto \mathsf{NN}, \texttt{JobTitle} \mapsto \mathsf{NN} \rangle$$
$$J_1 = J_1^0 \sqcup J_1^1 = J_0.$$

It is easy to see that J_0 is the strongest state-based object invariant for the modified class `Person'`. Therefore no imprecision is introduced by the abstract domain (or in general by the widening).

The analysis verifies the field dereferences in `GetFullName`: the explicit check for `JobTitle` nullness ensures that the successive access is correct. On the other hand, the analysis now issues a warning for the dereference of `JobTitle` in `BaseSalary`. This warning is no longer a false alarm, but a symptom for a real defect in the code. However, *what* is the real origin of this problem? Who should be blamed? Was the original class correct? Or, could it be that the verification of `BaseSalary` in the first version of `Person` was just a "lucky" accident? After all, the programmer was protecting against `JobTitle` being null in `GetFullName`, but she forgot to do so in the other method. The class-level static analysis on `Person` was smart enough to prove that the field `JobTitle` was always not-null. But was it the intent of the programmer?

Ideally, if the programmer wanted `JobTitle` to be not-null, then we should emit the warning in the new version of the class where the field is assigned a null value (i.e., the constructor). If, on the other hand, the programmer allowed `JobTitle` to be null, then we should emit the warning where the field is dereferenced (i.e., `BaseSalary`), and we should produce an inter-procedural trace leading to the alarm.

One can argue that if `Person(string)` is not used in the program, no warnings would have been emitted. However it is a good design pattern to consider and analyze a class as a whole, regardless of how it is used. Moreover a class can be compiled as a library and used outside the project: the analysis cannot rely on the context.

Changing the Initial State of the Object: Suppose that in the example of Fig. 1 the precondition for `Person(string, JobTitle)` was omitted or deleted. Call the resulting class `Person''`. Then the object fields can be assigned `null` so that invariant (1) is T (trivially true invariant). As a consequence the analysis can no longer prove the safety of the field dereference in `BaseSalary`, and it emits a warning in that method. While the analyzer is correct in pointing out that dereference, because that is the point where the runtime error will occur, the *real* error may be in the constructor where the programmer has not prevented

her implementation from leaving `jobTitle` initialized to null. Catching this kind of weakness in realistic and large classes is in general quite difficult.

Usability Impact: In general, an apparently innocent and harmless code addition like a new constructor caused a warning in a piece of code that was previously verified. Adding new constructors or new methods may cause the object invariant to be weaker, hence causing the analysis to emit alarms in many places that are apparently unrelated to the changes. Debugging those problems can be very painful for the programmer: it is hard from invariant (1) to trace back the origin of a warning. In a realistic setting, a class definition may contain dozens of methods, some of them with a complex control flow graph. Furthermore, the underlying abstract domain used by the analyzer may be very complex—in general, the reduced product of many abstract domains, e.g., alias, numerical, symbolic, arrays, etc. As a consequence, the object invariant inferred according to (1) may not be immediately evident to the programmer. Pretty-printing the inferred invariant is of little help. The programmer would have to understand why the tool inferred the object invariant, and how this invariant was used to prove the assertions in the previous version of the class. Then, she would have to inspect the newly inferred invariant, understand why it is different from the previous one, i.e., to identify the root cause of the alarm. In real code this process is time consuming, and requires the programmer to have some expertise about the internals of the analyzer, something we want to avoid.

In principle, the above noted brittleness is problematic for all inference problems, e.g., loop invariants and postconditions. However, according to our experience, loop invariants and postcondition inference is pretty stable. We guess that these inferences are more stable because they are locally inferred. Object invariant inference, on the other hand, manifests a more chaotic behavior: according to equation (1), the effects of small changes are amplified by the interleaved propagation to all the other class members, *de facto* losing locality.

2.4 Necessary Object Invariants

We propose a different, new approach to the object-invariant inference problem. We detect conditions that *should* hold on the fields of an object, for otherwise we can exhibit a trace from object construction and a series of method calls that leads to an assertion failure inside a method of the class. These conditions are *necessary* for the object's correctness. Technically, instead of performing a forward analysis as in equation (1), we perform a goal-directed backward interprocedural propagation of potentially failing assertions. By proceeding backwards, we can infer an abstract error trace, producing a more meaningful message for the programmer. We illustrate our technique with the `Person` example.

In the original class `Person`, we first run a method-level separate modular analysis of the methods, assuming the object invariant to be T [5]. `cccheck` will report the warnings as in Sect. 2.1. Using the techniques of [8] we try to infer preconditions by pushing the assertions that cannot be proven to the method entry points. In our example, we get the two following conditions on the entry state: $\mathcal{I}$(`GetFullName`) = `this.Name != null` and $\mathcal{I}$(`BaseSalary`) =

[5] If the class contains a programmer-provided invariant, we will use it.

`this.JobTitle != null`. $\mathcal{I}(\texttt{m})$ denotes a necessary precondition for the method `m`. The conditions are necessary for the correctness of the method: if violated, an error will definitely occur. In this example, they happen to also be sufficient: if they hold at entry, then no error will appear at runtime. However, they cannot be made *pre*-conditions as they violate the visibility rules of the language: a precondition cannot be less visible than the enclosing method [24]. The two fields are *private* to the object, but the conditions are on the entry of *public* methods. Thus, there is no way for the client to understand and satisfy these conditions at call-sites. The conditions are internal to the object, but they should hold whenever the respective method is called. In particular, they should hold just after object construction, i.e., just after the constructor is done. Our analysis pushes the necessary conditions to the exit point of constructor `Person(string, JobTitle)` as postconditions that should be established by the constructor. The analyzer can easily prove the two assertions (they follow from the constructor precondition). As the fields are marked as readonly, their value cannot be changed by methods after construction, and so $\mathcal{I}(\texttt{GetFullName}) \wedge \mathcal{I}(\texttt{BaseSalary})$ is an object invariant. Overall, no warning is raised for `Person`.

In `Person′`, the conditions $\mathcal{I}(\texttt{GetFullName})$ and $\mathcal{I}(\texttt{BaseSalary})$ are propagated backwards to both constructors. In the newly added constructor, the assertion `this.JobTitle != null` is false. `cccheck` emits an alarm pointing to the newly added constructor (instead of the field dereference in `BaseSalary` as in Sect. 2.2). Furthermore, `cccheck` produces an error *trace*: the sequence of calls $\langle \texttt{Person}'(s), \texttt{BaseSalary} \rangle$, for any s, will drive the execution into a null dereference.

In `Person″`, the conditions are propagated to the constructor. None of the conditions are satisfied by the constructor at the exit point: both `this.Name` and `this.JobTitle` can be null. The analysis further propagates those assertions to the constructor entry, as preconditions: `name != null && jobTitle != null`. It also infers a provenance trace: violating the first (resp. second) precondition will point to a failure in `GetFullName` (resp. `BaseSalary`).

3 Preliminaries

A class is a tuple $\langle \texttt{F}, \texttt{C}, \texttt{M}, \texttt{I}_\texttt{F} \rangle$, where `F` is a set of fields, `C` is a set of object constructors, `M` is a set of methods, and $\texttt{I}_\texttt{F}$ is an object invariant. A field $\texttt{f} \in \texttt{F}$ has a type, a visibility modifier `private` or `public`, and an optional `readonly` flag specifying if the field can be assigned only in constructors. We refer to constructors and methods as members ($\texttt{m} \in \texttt{C} \cup \texttt{M}$). Each member has a signature (return type, parameter types), a visibility modifier `private` or `public`, a body $\texttt{b}_\texttt{f} \in \mathbb{S}$, an optional precondition $\texttt{Pre}_\texttt{f}$, and an optional postcondition $\texttt{Post}_\texttt{f}$. When clear from the context, we omit the subscript for the member from the body, the precondition, and the postcondition. The optional object invariant $\texttt{I}_\texttt{F}$ is a property only on the fields in `F`. We assume the contracts are expressed in a side-effect free language $\mathbb{B}$. For the sake of simplicity, we focus our attention on private fields and public constructors and methods. It is difficult to state an object invariant on public fields (preconditions and postconditions are better

suited for it) and private methods do not contribute to the object invariant. We do not consider inheritance in this paper.

Let Σ be a set of states and $\tau \in \mathcal{P}(\Sigma \times \Sigma)$ be the transition function. The partial-trace semantics of the body $\mathtt{b}$ of a member has a fixpoint characterization:

$$\tau_{\mathtt{b}}^{+}(S) = \mathsf{lfp}\lambda T.\ S \cup \{\sigma_0 \dots \sigma_n \sigma_{n+1} \mid \sigma_0 \dots \sigma_n \in T \wedge \tau_{\mathtt{b}}(\sigma_n, \sigma_{n+1})\}.$$

The concretization function $\gamma_{\mathbb{B}} \in \mathbb{B} \to \mathcal{P}(\Sigma)$ gives the semantics of a contract in terms of the set of states Σ. The initial states for the execution of a member are $S_0 = \gamma_{\mathbb{B}}(I_{\mathtt{F}}) \cap \gamma_{\mathbb{B}}(\mathtt{Pre})$. The partial-trace semantics of a member is $\tau_{\mathtt{b}}^{+}(S_0)$. The bad states B are the ones violating some code or language assertions ($B_s \subseteq \Sigma$), or the postconditions: $B = B_s \cup \gamma_{\mathbb{B}}(\neg\mathtt{Post})$. The finite bad traces are those that contain at least one bad state: $B^* = \{\sigma_0 \dots \sigma_n \in \Sigma^* \mid \exists i \in [0, n].\sigma_i \in B\}$. The good runs of a member from $S \subseteq S_0$ are $\mathcal{G}(\mathtt{b}, S) = \tau_{\mathtt{b}}^{+}(S) \setminus B^*$. Dually, the bad runs of a member from $S \subseteq S_0$ are $\mathcal{B}(\mathtt{b}, S) = \tau_{\mathtt{b}}^{+}(S) \cap B^*$.

We assume an abstract domain $\mathcal{A}$ soundly approximating the set of states, i.e., $\langle \mathcal{P}(\Sigma), \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathcal{A}, \sqsubseteq \rangle$ [7]. When $\bar{\mathtt{a}} \in \mathcal{A}$ is such that $S_0 \subseteq \gamma(\bar{\mathtt{a}})$, then the abstract semantics $\bar{\mathtt{s}}[\![\mathtt{b}]\!](\bar{\mathtt{a}}) \in \mathcal{A}$ overapproximates $\alpha_{\Sigma}(\tau_{\mathtt{b}}^{+}(S_0))$ — α_{Σ} being the abstraction collecting the states in a set of traces.

In [8] we defined the problem of necessary initial conditions inference as finding an initial condition $\mathtt{e} \in \mathbb{B}$ such that: $\mathcal{G}(\mathtt{b}, \gamma_{\mathbb{B}}(\mathtt{e}) \cap S_0) = \mathcal{G}(\mathtt{b}, S_0)$ and $\mathcal{B}(\mathtt{b}, \gamma_{\mathbb{B}}(\mathtt{e}) \cap S_0) \subseteq \mathcal{B}(\mathtt{b}, S_0)$, i.e., $\mathtt{e}$ is a condition at the entry of $\mathtt{b}$ such that: (i) all good runs are preserved; and (ii) only bad runs are eliminated. Please note that the conditions for the necessary precondition inference are equivalent, because of monotonicity, to only requiring that $\mathcal{G}(\mathtt{b}, \gamma_{\mathbb{B}}(\mathtt{e}) \cap S_0) \supseteq \mathcal{G}(\mathtt{b}, S_0)$. Also, the problem is different from the inference of the weakest (liberal) preconditions, which imposes that *all* the bad traces are eliminated. For instance, a trivial solution to our problem is $\mathtt{e} = \mathtt{true}$, but $\mathtt{true}$ is not (in general) a solution of $\lambda X.X \implies \mathsf{wlp}(\mathtt{b}, \mathtt{Post})$. Cousot, Cousot, and Logozzo [8] presented several static analyses $\mathcal{I}(\mathtt{b}) \in \mathcal{A} \to \mathbb{B}$ to infer non-trivial initial necessary conditions, parameterized by the abstract domain $\mathcal{A}$. Those analyses are more or less precise. They can discover simple predicates (e.g., `x != null`), disjunctive conditions (e.g., $\mathtt{x} \leq 0 \vee 0 < \mathtt{y}$), or quantified conditions over collections (e.g., $\forall i.0 \leq \mathtt{arr}[i]$ or $\exists i.\mathtt{arr}[i] = 123$). The common idea behind the analyses of [8] is to find a Boolean expression $\mathtt{e}$ in terms of the member entry point values, such that if $\mathtt{e}$ is violated at the entry point, then an assertion $\mathtt{a}$ will later definitely fail (up to non-termination). We denote this relation as $\mathtt{e} \rightsquigarrow \mathtt{a}$. We do not repeat the details of these analyses here, leaving them as a building block to our analysis of necessary object field conditions. Our goal here is: (i) to show how such analyses can be lifted to infer necessary field conditions; and (ii) to prove that this inference is competitive with the forward class-level modular analysis in terms of precision and performance.

4 Inference of Necessary Conditions on Object Fields

Our goal is the inference of necessary conditions on object fields. A necessary field condition is a property that should hold on the instance fields of an object,

otherwise there exists a context (sequence of public method calls) that causes the object to be in a bad state. In this section we design a new static analysis, built on the top of a generic abstract interpretation of the class members, to infer such conditions.

The verification of a member $\mathtt{m}$, $\mathtt{cccheck}(\mathtt{m}, \mathbf{out}\ \bar{\mathtt{a}})$[6], works in two steps: (i) first analyze the member to infer program-point specific invariants, call these $\bar{\mathtt{a}}$; (ii) use these inferred invariants to prove the assertions in $\mathtt{m}$ body. Assertions can be user-defined assertions or language assertions (e.g., pointer dereference, arithmetic overflow, division by zero ...). The member precondition and the postconditions of the called functions are turned into assumptions (nothing to be proven). The member postcondition, the object invariant, and the preconditions of the called functions are turned into assertions (they should be proven). For methods only, the object invariant is assumed at their entry point.

If no alarm is raised, i.e., the member has been verified, there is nothing to do. Otherwise, `cccheck` tries to infer an initial condition. The necessary initial condition for a member $\mathtt{m}$ is $\mathcal{I}_0 := \mathcal{I}(\mathtt{m})(\bar{\mathtt{a}})$. If $\mathcal{I}_0 \equiv \mathtt{true}$ we are done: there is no way to push any of the failing assertions to the entry point.

If $\mathtt{m}$ is a constructor, then $\mathcal{I}_0$ contains predicates on parameters, on the public fields reachable from the parameters, or on private fields of an object of the same type as the type $\mathtt{m}$ belongs to.

In the first two cases, $\mathcal{I}_0$ can be made a precondition of $\mathtt{m}$. We denote by $\phi_P := \pi_1(\mathcal{I}_0)$ the components of $\mathcal{I}_0$ that are valid preconditions according to the language visibility rules. Note that in general, when $\mathtt{m}$ contains loops, ϕ_P does not guarantee its *safety*. The safety in general can be checked by re-running the verification with the new precondition: $\mathtt{cccheck}(\mathtt{m}[\mathtt{Pre} \mapsto \mathtt{Pre}_\mathtt{m} \wedge \phi_P], \mathbf{out}\ \bar{\mathtt{a}})$. In some cases, safety can be guaranteed by construction, e.g., if $\mathtt{m}$ contains no loops. If we can prove safety, we can mask the warning (the precondition is a necessary and sufficient condition).

In the last case, when an object of the same type as the type $\mathtt{m}$ belongs to is passed as a parameter, we emit the warning, and point out the assumption the programmer is making on the private field of a different object, suggesting a refactoring.

If $\mathtt{m}$ is a method then there is a fourth case for $\mathcal{I}_0$: it may contain some condition ϕ_I on the private fields of the `this` object. The condition ϕ_I cannot be made a precondition of $\mathtt{m}$ as a precondition less visible than the enclosing member is not allowable by the Design-by-Contracts methodology: there is no way for the client to satisfy the precondition[7]. Nevertheless, ϕ_I is a necessary condition on the object fields, which should hold whenever the method is invoked. In particular, it is a condition that should hold just after any of the constructors $\mathtt{c}$ or any of the (public) methods $\mathtt{m}'$ are executed. Otherwise we can construct a context where the call to $\mathtt{c}$ or $\mathtt{m}'$ is immediately followed by a call to $\mathtt{m}$, and we know by construction that invoking $\mathtt{m}$ in a state satisfying $\neg\phi_I$ will definitely

[6] As usual in programming languages the keyword **out** denotes out parameters.

[7] Remember that we are only considering public methods. If the method was private, then the condition could have been made a precondition.

```
Result: A necessary condition 𝓘* on object fields
while true do
    φ ← true
    foreach m ∈ M do
        if ¬cccheck(m, out ā) then // Strengthen precondition and invariant
            ⟨φ_P, φ_I⟩ ← π2(𝓘(m)(ā))
            Pre_m ← Pre_m ∧ φ_P
            φ ← φ ∧ φ_I
        end
    end
    if φ = true then
        break// no change on I_F, we are done
    else
        I_F ← I_F ∧ φ
    end
end
foreach c ∈ C do
    if ¬cccheck(c, out ā) then // Strengthen the precondition
        Pre_c ← Pre_c ∧ π1(𝓘(c)(ā))
    end
end
```

Algorithm 1: The necessary field conditions inference algorithm. The algorithm can be easily instrumented to trace the provenance of the ϕs appearing in $\mathcal{I}^*$ and hence to construct a failing context.

cause a failure (up to termination). Therefore we can strengthen the object invariant to $\mathtt{I_F} \mapsto \mathtt{I_F} \wedge \phi_I$, and repeat the analysis of all the class members. We denote by $\langle \phi_p, \phi_I \rangle := \pi_2(\mathcal{I}_0)$ the pair containing the new precondition for $\mathtt{m}$ and the necessary condition on the object field.

For each new condition ϕ added to the object invariant we can remember its provenance (the failing assertion and the containing member). In general, we can generate a provenance chain $\mathtt{a}_n \rightsquigarrow \mathtt{a}_{n-1} \rightsquigarrow \ldots \mathtt{a}_0$. Suppose that $\mathtt{a}_n$ is an assertion in one of the object constructors. By construction we know (up to termination) that if $\mathtt{a}_n$ is violated, then we can construct a sequence of public method calls (the ones containing the assertions $\mathtt{a}_i, i \in [0, n]$) causing $\mathtt{a}_0$ to fail at runtime. Therefore, we can produce more meaningful warnings to the programmer by pointing out the assertion $\mathtt{a}_n$ and the sequence of public method calls that will lead to an assertion failure at runtime if $\mathtt{a}_n$ is violated.

Algorithm 1 formalizes our explanations above. It computes a greatest fixpoint on the abstract domain $\mathbb{B} \times (\mathtt{C} \to \mathbb{B}) \times (\mathtt{M} \to \mathbb{B})$. The partial order is the pointwise lifting of the logical implication $\Rightarrow$. Please note that the algorithm may not terminate, necessitating widening for convergence. A simple widening is to limit the number of iterations (k-widening). The algorithm can be optimized using Gauss-Seidl chaotic iterations [6], i.e., by strengthening $\mathtt{I_F}$ after the analysis of each method.

It is easy to modify it to track the origin of the assertions. The object condition is in conjunctive form: $\phi_I = \phi_I^0 \wedge \cdots \wedge \phi_I^n$. By construction of $\mathcal{I}$, each $\phi_I^i, i \in [0, n]$ originates from some assertion $\mathtt{a}$ that cannot be proven: $\exists \mathtt{a}.\ \phi_I^i \rightsquigarrow \mathtt{a}$. In turn, $\mathtt{a}$ may be an inferred condition with its own provenance, and so on, effectively building a provenance chain. We denote by $\mathcal{I}^*$ the conjunction of the ϕs computed by the algorithm at each step.

5 Object Invariants for Readonly Fields

The predicate $\mathcal{I}^*$ inferred by the Algorithm 1 is a necessary condition on the object fields. It states a condition that must hold on the object's fields in between calls to public methods for otherwise, there exists a series of method calls from object construction to a guaranteed assertion failure. In general, the definition of object invariants is tied to a particular methodology for when object invariants are supposed to be valid. These programming methodologies define at which program points the object invariant holds, and which aliasing containment policy is assumed and enforced. Different methodologies adopt different policies (e.g., `cccheck` [15], Spec# [1], jStar [10] or VeriFast [20]). The programming methodology is an orthogonal issue to the solution presented in this paper, since it is mainly concerned with guaranteeing that object invariants are *sufficient* for proving the safety of all methods. In contrast, we are inferring *necessary* conditions on fields without which failure is guaranteed. This distinction is analogous to the distinction between necessary and sufficient preconditions [8]. Thus, in this paper we do not consider any particular object invariant methodology.

Yet, it is still useful to point out one special case, where we can indeed talk about an object invariant: If we restrict ourselves to include in Algorithm 1 only `readonly` fields, then the inferred predicate is truly an object invariant, since it needs to hold at every program point after construction (independent of methodology).

A field marked as readonly can only be updated in a constructor of the class it belongs to [19]. Assignment to a readonly field is not compulsory. If a readonly field is not initialized, it gets the default value (e.g., `null` for heap objects). A readonly field is different from a *constant* field in that it is not a compile time constant.

The algorithm for the inference of object invariants for readonly fields differs from Algorithm 1 in the way the inferred necessary conditions for the methods are selected: $\langle \phi_p, \phi_I, \phi_A \rangle := \pi_3(\mathcal{I}(\mathtt{m})(\bar{\mathtt{a}}))$. The function $\pi_3 \in \mathbb{B} \to \mathbb{B}^3$ partitions the inferred conditions according to visibility rules. The precondition ϕ_P contains only variables as-visible-as the method. The readonly object invariant ϕ_I contains only readonly fields of the `this` object. Finally the input assumption ϕ_A contains conditions necessary for the method correctness, but which cannot be included in the method precondition or the readonly invariant.

It is easy to see that $\mathcal{I}^*$ computed by this modified algorithm is an invariant on object fields, no matter which methodology for object invariants is chosen. Indeed, the initial, programmer-provided condition $I_{\mathtt{F}}$ is an invariant. All the successive additions are properties on readonly fields that should hold at the end of the constructor and — by the semantics of readonly fields — cannot be further changed.

6 Experiments

We evaluate the cost and the precision of our analysis by comparing it to runs of `cccheck`: (i) without object invariants inference; (ii) with object invariants inference based on class-level modular analysis. We want to measure the extra

cost and precision induced by our analysis. We use the main libraries of the .NET framework (mscorlib, System, System.Core) and some randomly chosen libraries as benchmarks. The libraries are available in all Windows installations.

Experimental setting. We ran `cccheck` with four different settings:

(BR) with the object invariant inference disabled;
(NCR) with the object invariant inference enabled for readonly fields only (Sect. 5);
(NC1) with the object invariant inference enabled for all fields (Sect. 4), with the constraint of analyzing every method only once;
(CLMAR) with the forward class-level modular analysis enabled for readonly fields only (Sect. 2.2).

In the (BR) experience, we ran `cccheck` with the default abstract domains and options. The default abstract domains include a domain for the heap analysis, one for the non-null checks, several numerical abstract domains, and an abstract domain for collections (more details in [15]). In this configuration, `cccheck` verifies the absence of: (i) contract violations, (ii) null dereferences, and (iii) out-of-bounds array accesses. `cccheck` performs intra-procedural modular analysis: it infers invariants on the member body which it uses to discharge the proof obligations. It exploits contracts to achieve inter-procedural analysis. To reduce the annotation burden, `cccheck` infers and propagates preconditions and post-conditions. First, `cccheck` computes an approximation of the call-flow graph of the assembly under analysis. Then it schedules the analysis of the members in a bottom-up fashion: callees are analyzed before callers. Inferred preconditions (resp. postconditions) are propagated to the callers as additional asserts that should be proven (resp. assumes it can rely over). Finally, to improve performance, `cccheck` implements a caching mechanism to avoid re-analysis: if it detects that a member has not changed (in our case, no new contract has been added to the member or to its callees) it skips its analysis, and simply replays the previous result (warnings, inferred contracts).

(NCR) adds to (BR) the algorithm to infer object invariants for *readonly* fields. We include (NC1) in the experiments to get a better sense of the power of inferring necessary field conditions. It computes field conditions for *all* fields, but without iterating the propagation until a fixpoint is reached. It is essentially Algorithm 1 instantiated with the Gauss-Seidl iteration schema with a 1-widening (i.e., the widening is set to one iteration). (CLMAR) adds to (BR) object invariants inference for readonly fields using a class-level modular analysis.

Results. In Fig. 2, we report, for each library, the total number of methods, the number of proof obligations (language assertions or inferred contracts) generated during the analysis, the number of proof obligations that cannot be proven and the overall execution time. The first thing to note is the increase on checks. This is due to the field conditions being inserted as postconditions to be checked at constructor exits and method exits, as well as propagation of necessary field conditions to preconditions of constructors, and on to call-sites. In return, we assume the invariant at method entries. The increase in proof obligations is paid for by the reduction in overall warnings. In the (NCR) experience, we have at worst 45 (Data.OracleClient) and at most 741 (Data.Entity) fewer

		(BR)			(NCR)			(NC1)			(CLMAR)		
Library	# Meth.	Checks	Top	Time	Checks	Top	Time	Checks	Top	Time	Checks	Top	Time
mscorlib	22,904	113,551	13,240	31:41	113,750	13,084	27:36	115,002	11,053	32:22	116,116	13,152	26:12
Addin	552	4,170	682	4:15	4,148	605	4:07	4,295	485	4:11	4,067	571	12:55
Composition	1,340	6,228	909	0:44	6,356	791	0:46	6,302	743	0:47	8,095	885	1:57
Core	5,952	34,324	5,323	29:57	36,100	4,820	33:50	36,196	4,463	34:54	42,602	5,715	72:31
Data.Entity	15,239	88,286	12,460	23:13	87,743	11,719	24:02	91,591	15,861	27:59	88,125	11,569	43:36
Data.OracleClient	1,961	9,596	1,070	2:38	9,738	1,025	2:21	9,736	887	2:26	107,23	1,018	3:25
Data.Services	2,448	18,255	3,118	6:45	18,518	2,938	7:23	18,733	2,749	6:54	21,818	2,989	24:18
System	15,586	94,038	8,702	15:03	93,948	8,644	15:15	96,154	10,693	15:30	94,008	8,648	17:37

Fig. 2. Experiments results showing the impact of our static analysis in reducing the false warnings and its comparison with a class-level modular analysis. Time is in minutes and seconds. Columns Checks and Top are respectively the total number of proof obligations and the number of proof obligations that cannot be decided.

warnings. When percentages are observed, the best improvement is in the Core benchmark (2.16 %). (NC1) in general provides an even more dramatic reduction of the total number of warnings (up to 2, 187 less in mscorlib) but in two cases it adds to the baseline (e.g., 3, 401 more warnings for Data.Entity). The reason for it is that in some cases the inferred object invariant is quite complex (many disjuncts). It gets propagated as precondition for the constructor. The constructor is called in many places and either `cccheck` cannot prove the precondition at those call sites, or it is further propagated, sometime originating in an even more complex call-site assertions, etc. As one may expect (CLMAR) is generally more precise than (BR) – up to 891 fewer warnings on Data.Entity. Nevertheless there is no definite answer whether (CLMAR) is more precise than (NCR). When absolute numbers are compared, in 5 cases (mscorlib, Composition, Core, Data.Services, System) (NCR) emits fewer warnings than (CLMAR). On the other hand, when the ratio Top/Checks is considered, in 6 cases (mscorlib, Addin, Composition, Data.Entity, Data.OracleClient, Data.Services) (CLMAR) provides better results than (NCR). Overall, precision-wise there is no big difference between (NCR) and (CLMAR).

When performances are considered, results are somehow surprising. A more precise analysis does not always mean a slower analysis. For instance the analyis of mscorlib was faster in both (NCR) and (CLMAR) than in (BR). On the other hand, in some benchmarks (CLMAR) was way slower than (NCR) and (NC1) (e.g., Addin, Core, Data.Entity, Data.Service). In those benchmarks the bottom-up propagation of inferred pre/post-conditions caused (CLMAR) to converge very slowly. For instance, a new inferred contract may trigger the re-analysis of a constructor, originating in a new object invariant, which at its turn impacts, e.g., the postconditions inferred for the methods of this type, etc. Of course, theoretically (NCR) may suffer from the same problem, but we did not experienced it in our experiments. Overall, performance-wise (NCR) seems a better choice than (CLMAR): in all but one experiment (mscorlib) it is faster, it adds a modest slowdown w.r.t. (BR), and it seems to be less prone to hit performance corner cases.

In Fig. 3 we report the number of field conditions inferred (after simplification, to remove redundant ones). For (NCR) and (NC1) we also report the number of violations, i.e., the number of necessary field conditions for which

Library	(NCR) Inferred	Violations	(NC1) Inferred	Violations	(CLMAR) Inferred
mscorlib	41	1	823	80	61
AddIn	9	5	48	8	2
Composition	36	0	63	0	33
Core	150	0	323	6	81
Data.Entity	230	4	531	18	378
Data.OracleClient	13	0	75	8	54
Data.Services	53	1	96	11	49
System	22	0	349	55	131

Fig. 3. Experiments results showing the number of inferred conditions and the number of definite violations found.

`cccheck` was able to automatically find an instantiation context that *definitely* leads to a failure. When comparing (NCR) and (NC1), it is worth noting that the effectiveness of invariant inference is limited by the fact that in "old" libraries (e.g., mscorlib) few fields are actually declared as readonly. The inferred conditions distribute evenly between (NCR) with (CLMAR): in 4 benchmarks (NCR) infers more conditions than (CLMAR), and vice versa. Our static analysis was able to find several instantiation contexts definitely leading to a failure in some classes. Experiment (NCR) shows that in 11 cases (overall) it is possible to cause a class to fail by breaking an invariant on readonly fields. The failing context creates the object by invoking a particular constructor and just after calling a particular method (*e.g.*, the sequence `Person(string)`, `BaseSalary` in Sect. 2). In general, in experiment (NC1) the instantiation context is more complex, requiring longer invocation sequences to manifest.

Discussion. The experimental results are encouraging, but they require further study. In particular, given that programs are partially specified only, important usage conditions may be absent from the code. To the programmer, our necessary field conditions may appear too strong or the failing context may appear unfeasible. E.g., consider a class that requires an initialization method A to be called before allowing a call to B. If this initialization pattern is not specified using preconditions and postconditions on A and B, necessary field condition inference will probably find a condition for B that is only established after A, but not by the constructors. Our analysis will suggest it as an invariant to be added to the constructors, which may confuse the programmer.

In general, a necessary field condition $\phi_{\mathtt{m}}$ can be decomposed into a precondition $\mathtt{Pre}_{\mathtt{m}}$ and a weaker invariant I, such that $\mathtt{Pre}_{\mathtt{m}} \wedge I \Rightarrow \phi_{\mathtt{m}}$. In our example above, assume a public flag `init` is used to indicate that the object is initialized. In that case, the precondition for B is `init`, and the weaker invariant I is $\mathtt{init} \Rightarrow \phi_{\mathtt{m}}$. We plan to investigate how to decompose necessary field conditions into the above form.

Overall evaluation. The inference of field conditions improves the precision of the basic modular analysis (BR) – this was expected. The question is then which analysis to use, and in particular which one to provide to our users, who were asking for a readonly fields inference algorithm. When precision is considered (NC1) generally performs slightly better than (NCR) and (CLMAR), except one case where it adds thousands more warnings. Precision-wise, there

is no clear winner between (NCR) and (CLMAR) – the choice seems to depend on the particular code base. When the extra cost of the analysis is considered, (NCR) performs better than (NC1) and way better than (CLMAR) in most cases. As far as usability is concerned, (NCR) and (NC1) should produce a better programmer experience since warnings are more understandable and easier to fix. We have no formal user study on that issue, only anedoctal evidence from the interaction with our customers (e.g., on the CodeContracts MSDN forum [25]). Overall, we think that (NCR) is the analysis best suited for our users, and we decided to set it as the default object invariant inference in the CodeContracts distribution.

7 Related Work

Daikon [12] pioneered the dynamic invariant inference approach. Given a suite of test cases, Daikon observes the values, and then generalizes it to likely invariants (pre, post, and object invariants). As a dynamic technique, Daikon requires a good set of test cases in order to produce useful invariants. In contrast, our approach is purely static, it works without test inputs and it is not limited by the pre-set candidate invariants. DySy [9] uses dynamic symbolic execution to infer preconditions, postconditions, and invariants. Like Daikon, the approach depends on test suites but uses symbolic path condition computations to identify candidate invariants (so it is not limited to a pre-defined set of invariants).

On the purely static side of invariant inference, Logozzo [23] introduced the forward analysis described in the introduction (equation (1)). Chang and Leino [4] instantiated [23] using the Spec# methodology [1] and stronger heap invariants. Houdini [17] is an annotation inference system based on ESC/Java. It guesses invariants and uses ESC/Java to prove them. Houdini is limited by the pre-set initial candidates for invariants.

What sets our work apart from all the above is the focus on *necessary* conditions that, when violated, lead to failures. Instead, all the above analyses compute a form of strongest invariants given a set of possible candidates or abstract domains. We are not aware of work describing the problem of analysis brittleness with respect to small or simple program changes.

Surprisingly little research focuses on the problem of detecting the origin of alarms in abstract interpretation-based analyzers. We are aware of Rival [26] who studies the source of warnings in the context of the ASTREE analyzer, and Brauer and Simon [2] who use under-approximations to present a counter-example to the programmer. Our work differs from theirs because we do not only want to report the cause for a warning but we also want to infer a contract for the program. The problem is studied more widely in model checking [5] and deductive verification, e.g., [21], where the finiteness hypotheses make the problem more tractable. Our traces leading to failure can be viewed as an approach to finding the origin and explanations of warnings.

8 Conclusion

Necessary object field conditions provide an alternative approach to finding predicates that are candidates for object invariants. These conditions are computed

using a backward analysis from assertions within methods that will fail, unless the object field condition is satisfied on entry to the method. This approach produces invariant candidates that are demand-driven, as opposed to accidental implementation details that are often inferred by forward analyses, drastically eliminating the usual brittleness of object invariant inference caused by changes in the program. We evaluated our analysis in the context of the CodeContract static checker and found that it significantly reduces the number of warnings in a variety of large code bases.

References

[1] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: the Spec# experience. *CACM*, 54(6):81–91, 2011.

[2] J. Brauer and A. Simon. Inferring definite counterexamples through under-approximation. In *NASA Formal Methods*, 2012.

[3] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *FMCO*, 2006.

[4] B.-Y. E. Chang and K. R. M. Leino. Inferring object invariants: Extended abstract. *ENTCS*, 131, 2005.

[5] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 2001.

[6] P. Cousot. Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice. Res. rep. 88, Laboratoire IMAG, Université scientifique et médicale de Grenoble, France, 1977.

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

[8] P. Cousot, R. Cousot, and F. Logozzo. Contract precondition inference from intermittent assertions on collections. In *VMCAI*, 2011.

[9] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: dynamic symbolic execution for invariant inference. In *ICSE*, 2008.

[10] D. Distefano and M. J. Parkinson. jStar: Towards practical verification for Java. In *OOPSLA*, 2008.

[11] S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. A unified framework for verification techniques for object invariants. In *ECOOP*, 2008.

[12] M. D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington, 2000.

[13] M. Fähndrich, M. Barnett, and F. Logozzo. Code Contracts, 2009.

[14] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA*, 2003.

[15] M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS*, 2010.

[16] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV*, 2007.

[17] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME: Formal Methods for Increasing Software Productivity*, 2001.

[18] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI*, 2002.

[19] A. Hejlsberg, M. Torgersen, S. Wiltamuth, and P. Golde. *The C# Programming Language*. Addison-Wesley Professional, 2010.

[20] B. Jacobs, J. Smans, and F. Piessens. A quick tour of the VeriFast program verifier. In *APLAS*, 2010.

[21] C. Le Goues, K. R. M. Leino, and M. Moskal. The Boogie Verification Debugger (Tool Paper). In *SEFM*, 2011.

[22] F. Logozzo. Class-level modular analysis for object oriented languages. In *SAS*, 2003.

[23] F. Logozzo. *Modular static analysis of Object-oriented languages*. PhD thesis, École polytechnique, 2004.

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

[25] MSDN. CodeContracts Forum. `http://social.msdn.microsoft.com/Forums/en-US/codecontracts/threads/`.

[26] X. Rival. Understanding the origin of alarms in Astrée. In *SAS*, 2005.