Internship report : Abstract interpretation, contracts and object invariants

Michaël Monerau

July - October 2009

Directed by **Manuel Fähndrich**

at Microsoft Research, Redmond, WA, USA

Microsoft Research

Table des matières

1 The internship

1.1 Acknowledgements

I would like to thank Manuel Fähndrich for his mentoring during this internship, his availibilty and enthousiasm. It was very motivating and pleasant. I would also like to thank Francesco Logozzo for his relentless help and availibity. On a more general ground, thanks to all the people I chatted or worked with at MSR and the other interns (hey Jean !). They have all contributed to make my summer an awesome summer, it was really enjoyable to work with this team. And I would especially like to thank Radhia and Patrick Cousot for their always pleasant and interesting discussions.

1.2 Microsoft Research

Microsoft is the big software company we all know, based in Redmond, WA, USA. The firm has around 100 000 Full-Time Employees worldwide. There are more than 30 000 FTE's on the sole Redmond campus, which is thus the core of the company.

The main goal of Microsoft is to deliver quality software to enable easy, natural and efficient use of computers.

To achieve this goal, it is critical to stay at the leading edge of innovation and research is thus very important. That's why Microsoft features an important Research section. Microsoft Research brings together almost 1000 researches in various fields including programming languages, computer vision, hardware, servers, cryptography, theory, etc.

Microsoft Research funding is comfortable and allows for a pleasant and mellow research.

I was in the RiSE team (*Research in Software Engineering*), and more precisely in the "*Programming Languages and Analysis*" team, managed by Manuel Fähndrich
(http ://research.microsoft.com/en-us/groups/pla).

1.3 My curriculum

I have a Bachelor of Science and a Master of Science in Mathematics & Computer Science from the École Normale Supérieure in Paris, France. I also have the *agrégation* in Mathematics, minor Computer Science.

During this 2009-2010 year, I decided to delay the last year of my studies to do some research in abstract interpretation and this internship at Microsoft is the first step of this project.

2 Scientific contents

2.1 CodeContracts & Clousot

2.1.1 The workflow

CodeContracts is a project whose goal is to deliver the programmer with comprehensive tools to check his code both statically and dynamically.

The static analysis part is based upon abstract interpretation and applies to any compiled .NET binary. It means that the analysis can be achieved on any library or program coming from any one of the .NET languages (C#, Managed C++, VB .NET, ...).

Indeed, the .NET workflow is such that the source code can be written in any .NET language and then the compiler transforms that code into an intermediate language "MSIL" which can be seen as assembly code for the .NET virtual machine. One of the major advantages of MSIL over classic assembly code is that it keeps some interesting semantic informations such as class definitions, types, and so on.

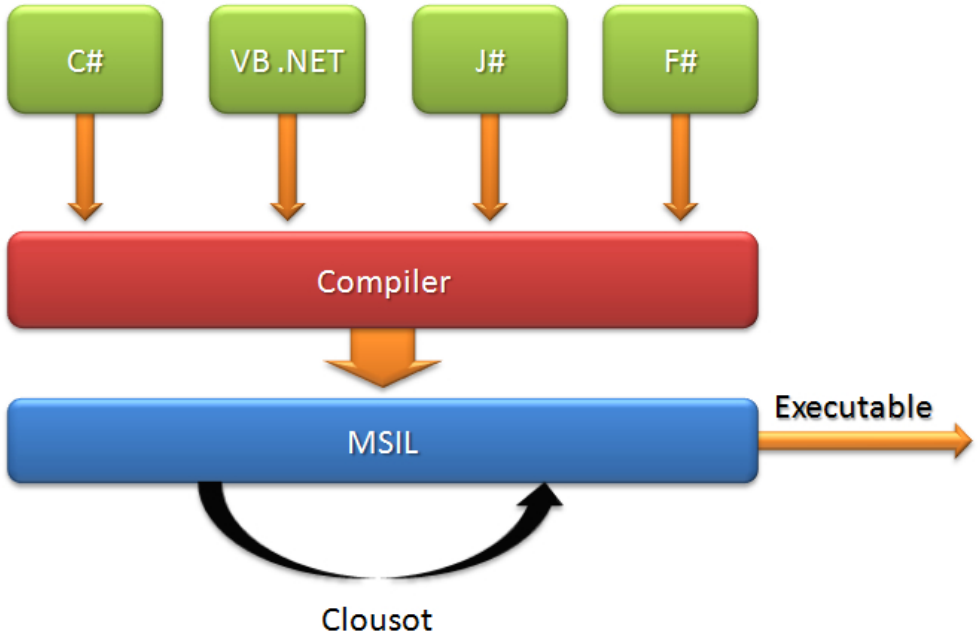

FIGURE 1 – The .NET Framework and Clousot analysis

That's why performing the analysis at the MSIL level is so interesting : it allows to treat any .NET language, and we have a post-compilation code (which frees us from verifying the compiler itself). Moreover, there are already some tools out to parse MSIL.

The analysis can be broken down into different pieces.

First, when the programmer writes some code, a .NET library allows the programmer to insert some annotations into the code (as genuine code, not comments, which are thus checked semantically) : pre-conditions and post-conditions on methods, object invariants, assertions, assumptions (see figure 5 for a quick example).

After compilation takes place (using the usual compiler for the target language) and if CodeContracts is switched on, then the annotations are propagated through the MSIL assembly to

allow for a later static and/or dynamic analysis.

Clousot, the static analyzer, then comes into play : it runs several analyzes on the code (non-null checks, non out-of-bounds array accesses, etc.) taking into account the annotations from the programmer and automatically inferring a whole lot of others. Then it outputs warnings (resp. errors) if bugs are possible (resp. present).

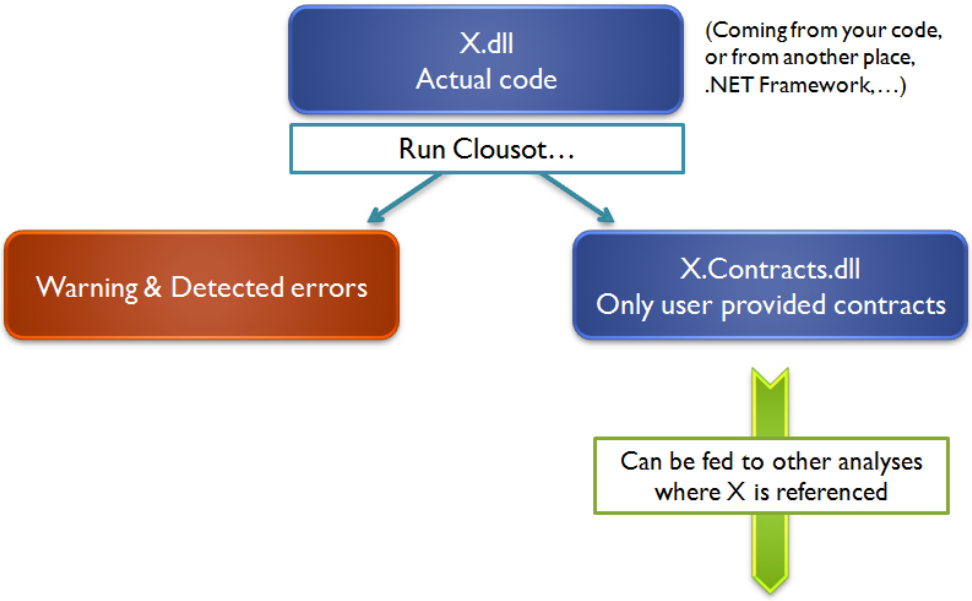

FIGURE 2 – Clousot Workflow, the static analyzer is the core of CodeContracts

The whole static analysis relies on abstract interpretation, we thus have soundness for granted (expect if there are human-made bugs in the analyzer) but it is possible to have some false alarms because of a lack of precision in the approximation during the inference.

If runtime analysis is switched on, then all the assertions, pre and post conditions are checked live at execution and throw an exception if they are not valid.

2.1.2 My work in there

I have done two projects during my internship. The first one aimed at making the contracts inferred during an analysis reusable, and the second one was about infering object invariants. Let's see that in more detail.

1st project : if an assembly A references another assembly B, one would like to be able to import B's contracts into A analysis. Before my internship, the only contracts you could import in such a way were those explicitly written by the programmer in B's source code because all the inferred one lived only in memory during B's analysis.

The goal was thus to make the contracts *inferred during B's analysis* available to A. As we may expect, the automatic inference is far more verbose than what could be written by hand. Such a feature could especially be very useful to generate contracts for standard system assemblies : one can launch an analysis of those system DLLs and then reuse the results on all future assemblies that depend on them.

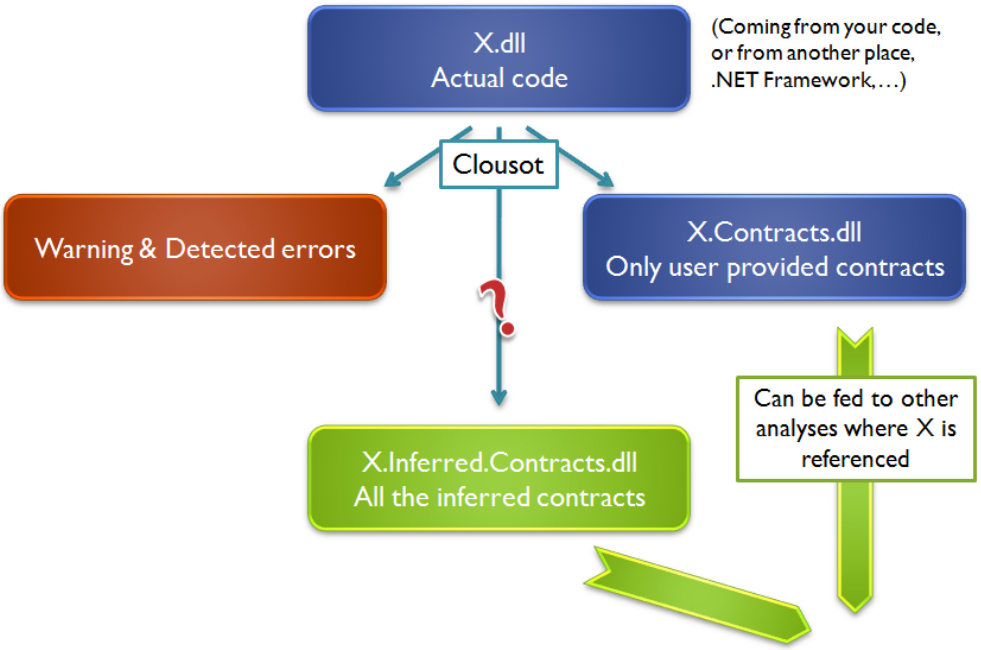

FIGURE 3 – My first project : Clousot Workflow & we would like to use inferred contracts outside B too

Moreover, abstract interpretation allows the user to choose carefully the analysis precision, so it could be possible to launch some very precise and time-consuming analyses on the system DLLs that are often referenced, and then save the results once and for all.

2nd project : the goal was to add class analyses to Clousot so as to be able to infer object invariants in addition to method contracts (until then, Clousot only inferred method contracts).

2.1.3 Implementation

The entire CodeContracts project is written in C#. We will only cover the abstract interpreter here in this report.

Reading MSIL code is achieved though the Microsoft CCI library (available on the internet). The code structure is then abstracted for the analyzer to treat higher level objects.

Method analyses, which are independant and each use their own abstract domain, are sequentially executed. The analysis order is dictated by the method call graph constraints.

The project is separated into different modules independant from each other, and limitates logic dependencies between the orthogonal parts of the implementation.

Let's give us a quick overview of the scale of the project. The CodeContracts project is several hundred of thousands lines of code, and Clousot alone is between 50 000 and 100 000 lines.

2.2 Inferred contracts

2.2.1 The setting

When he writes code, the programmer can write contracts by himself (such as what you can see in Figure 5 further down). It is expected from him to give non trivial contracts to the

analyzer, ie. those too complex to be automatically inferred. Then the analyzer takes on the task of automatically deducing as much contracts as possible to prove as much assertions as possible (and thus avoid unnecessary warnings).

To get a better grasp of the order of magnitude we are dealing with, the number of inferred contracts on a big-scale DLL (such as System.dll from the .NET framework) is as high as several dozens of thousands (see exact figures in Figure 6).

In this setting, we asked ourselves several questions on the inferred contracts :

1. How to store them for reuse in another analysis ? That's what we already discussed in Figure 3.
2. Are they all useful ? In case some of them are not, we should delete them in order to speed up the analysis.
3. Are they all valid ? Of course, we don't want to infer false statements.

Let's see how we deal with these interrogations one after another.

2.2.2 Storing inferred contracts

Storing inferred contracts is not as easy as it may first seem. We don't want to just store them by themselves (as a string for example), we also want to capture their semantic meanings so as not to have to re-analyze them when we re-import them.

Principle

The solution we chose is to get back to C# code : for each analyzed assembly, we generate all the C# code of the assembly except that the body of each method only contains inferred contracts (see Figure 5).

Once we have this code, it is just a matter of compiling it with the usual compiler and you get a DLL containing all the inferred contracts for free. Now, you can feed any analysis depending on this specific assembly with this DLL and the contracts will be properly imported (see Figure 4).

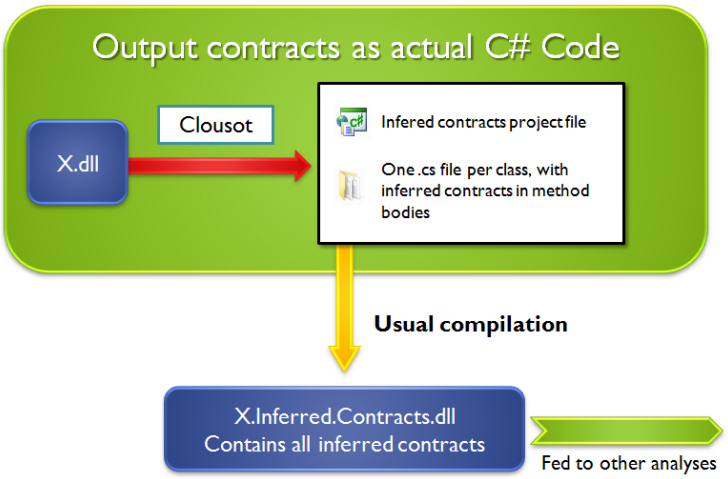

FIGURE 4 – Storing inferred contracts as C# code

Advantages and achievements

This approach, in addition to solving the initial problem, brings with it many other advantages that are worth exploring.

First of all, the produced code is human readable and allows for graceful hand-editing. It proves to be very useful, for instance in the aim of generating "*contracts libraries*" for system DLLs, which is essential to the effectiveness of CodeContracts. Until then, those contracts libraries had to be written by hand as C# code. Needless to say it was a long and cumbersome task. Now, the skeleton of those libraries are already at hand and you only have to add the non-inferred contracts you want, as shown on Figure 5.

```
public class ProtectedProviderSettings : ConfigurationElement
{
  #region Object Invariants
  [ContractInvariantMethod]
  protected void InferredObjectInvariant()
  {
    Contract.Invariant(this._propProviders != null);
  }
  #endregion

  #region Methods and constructors
  public ProtectedProviderSettings ()
  {
      Contract.Ensures(Contract.Result<System.Collections.Specialized.NameValue
      Contract.Ensures(this._PropertyNameCollection != null);
      Contract.Ensures(Contract.Result<System.Collections.Specialized.NameValue
  }
  #endregion

  Properties and indexers

  Fields
}
```

FIGURE 5 – Typical output of the *OutputPrettyC#* tool : the assembly skeleton including all the contracts (and object invariants, cf. project 2) inferred during the analysis. Here we show a sample coming from the analysis of System.Configuration.dll

Moreover, you get a semantic verification of the contracts basically for free (because it is done by the C# compiler). It proved to be very useful because there are so many reasons why a contract would not compile (violating some little forgotten C# rules is surprisingly easy) that you really have to take care to get it to actually compile. Thus, a compiling contract has high probability of being genuinely wanted. That is what we experienced.

Last but not least, it allows iteration of the analysis. When you have the inferred contracts stored somewhere, you can run the analysis again with that new information as input. However, because of small problems in the Clousot implementation at the time of my internship we couldn't measure the efficiency of such an approach.

My work

To realize this project, we had to intercept the inferred contracts inside Clousot and "clean" them. Independantly, I have coded a library allowing for valid and pretty C# output. In particular, it was of course used in this setting to output the C# code we wanted.

At the end of the day, everything has worked well. The biggest DLLs in the .NET framework lead to a fully compiling generated code, without error, containing all the inferred contracts of interest.

2.2.3 Contracts utility

As mentioned earlier, this approach is packed with a free semantic checking. This verification was in fact the first "stress-test" for the contracts that Clousot generates. Until then, contracts were used inside the analyzer and just output on the screen. There was no real way to check if they really "meant" something.

We learned a lot from this approach.

.NET programming is definitely object-oriented. However, we found that the analyzer was not always object-oriented in the way it handled some particular situations and we have thus worked towards better using this fact.

The first thing that came out of the compilation tests was that the analysis inferred a good proportion of useless contracts (which lead to compilation errors).

Typically, one could find in the body of some method a contract referring to the private field of another class, which was inaccessible from there. Such a contract cannot be of any use because this field will never be used in the method and thus it can be safely removed. Care has still to be taken though because this contract could be of use in the internal analysis of the other class, so it should not be *totally* removed.

This leads to a subtle distinction between contracts : some are "interface contracts", ie. they concern public fields and give information for other people wanting to use the object ; and some others are "internal contracts" which should remain private to the declaring class and just allow for more interface contracts inference.

The compilation of our generated C# code also allowed us to detect contracts which made no sense simply because of typing errors. Mainly we ran through `signed`/`unsigned` issues and more importantly arithmetic operations between booleans and integers (booleans being just 0 and 1 values in the MSIL at some places) or between `enum` and integers. These contracts did not make sense or needed some little adjustments which have made the overall analysis more robust.

The compilation also enabled us to detect contracts that were not false but just trivial (a field address is not null... That's true but also always true, it is safe to skip it).

After all those improvements and filtering were implemented in Clousot, the inferred contracts count dropped dramatically, leading to a substantial speedup in the analysis, as witness the following figures :

	System. Configuration.dll		System.dll		System. Deployment.dll		System. Design.dll	
	Avant	**Après**	**Avant**	**Après**	**Avant**	**Après**	**Avant**	**Après**
Temps d'analyse	2'50"	1'30"	20'40"	18'30"	2'50"	1'10"	14'20"	11'20"
Post-conditions	3397	1469	27149	13866	3287	1596	19458	10372
Pré-conditions	849	642	8349	7883	1323	924	11132	10603
Nb. de métodes	1921		15497		2212		12922	

FIGURE 6 – Results : useless contracts are gone, the analysis is faster

Of course, the big deal is not to lower the contracts count, but to do so without losing deducing power. And indeed, we could see that the number of proven assertions did not change after our modifications.

2.2.4 Contracts soundness

As already mentioned, the static analyzer is based upon the abstract interpretation theory. This theory guarantees that inferred contracts are *sound*. We might miss some because of a lack of precision, but all what is inferred is fortunately valid.

However, practice is a little bit different : the analyzer implementation is human-made and thus fatally has some bugs. How to detect them ?

These errors often only appear in very particular cases (because most common cases are validated by the programmer when he does his testing). But the paradox is that on huge DLLs where we could actually see those failures, there are too many contracts for anyone to eyeball them.

Our solution of recompiling contracts has proved to be useful for detecting those errors. As mentioned earlier, there are so many reasons for a contract not to compile that an error in the analyzer is likely to lead to an uncompilable contract, which is thus automatically pin-pointed. Of course, this is no rock-solid argument, but our approach really made us aware of some bugs in the analyzer that we have then fixed.

The bugs were mainly about wrongly over-generalized preconditions and about wrong control flow construction is some cases leading to some false contradictions.

2.3 Object invariants

The second part of my internship was about implementing some object analysis in Clousot.

Until then, Clousot only inferred contracts about methods and no object invariants. My task was to add this functionality to the Clousot framework. I then used this environment to implement a particular (and very basic) analysis.

2.3.1 Getting object invariants inference in Clousot

I will not get into the details of adding the object invariant facility into Clousot design, but it is still interesting to mention an issue we ran into that we did not suspect would be that important at first glance.

Memory footprint problem

To make an object analysis, you have to keep all the method analysis results available for each method of the class you are considering.

If several analyses are simultaneously run, you thus have to keep in memory several abstract states for each method until you launch the class analysis. The abstract states (one for each used abstract domain) can prove to require big amounts of memory, even for reasonably large methods.

It is clearly not feasible to run all the class analyses after all the methods have been examined, because it would require to store all the results of each analysis on each method during the whole execution – too much.

So we can think of an "on-the-fly" strategy : after the last method of a particular class has been analyzed, then the class analysis can be run (and all the informations about those methods can then be discarded).

But the method call graph put some constraints on the order of analysis. In practice, if you don't pay particular attention to the issue, the methods from a class can be run at unnecessary distant times, forcing you to keep their analysis results longer than optimal.

On big DLLs containing thousands of classes, we measured that the number of open class analyses at one time could go as high as 800 (and lead to *Out Of Memory* exceptions).

Solution

The analysis order had thus to be reorganized. A majority of methods don't have mutual constraints and there is thus plenty of room to do some improvements, especially by packing up methods coming from the same class together.

To do so, starting with the method call graph $\mathcal{G}_m$ we build a class call graph $\mathcal{G}_c$ where nodes are classes and an edge between c_1 and c_2 means that a method of c_1 calls a method of c_2 (which is no more than $\mathcal{G}_m$ quotiented by the relation $m_1 \sim m_2 \iff class(m_1) = class(m_2)$).

Until then, the methods were analyzed according to the topological order of the strong components of $\mathcal{G}_m$, noted $<_m$. But the order inside those strong components was not defined.

Here, we make use of $\mathcal{G}_c$ to pack methods from the same class together when $\mathcal{G}_m$ allows it.

More precisely, we first look at the strong components of $\mathcal{G}_c$ in the topological order, and then we use $<_m$ inside each of them to compute the analysis order. Thus, the time when we need to store information about methods is dramatically reduced, and it is easily seen that the constraints dictated by $\mathcal{G}_m$ are well respected.

Results

This approach does not have any perceptible impact on the execution time (we barely ask more than the construction of $\mathcal{G}_c$ along with a traversal and then some quick sorts according to $<_m$).

But the memory problem is now gone : on the same DLLs we experienced *Out Of Memory* exceptions before, the maximal number of simultaneous class analyses has now dropped to 25 which has a reasonable memory footprint.

2.3.2 `readonly` fields analysis

The object invariants analysis that I implemented was based on the fact that, for .NET languages supporting it (including C#), the `readonly` fields of a class can only be modified in constructors.

Principle

Let $\mathcal{C}$ be a class and let $(f_i)_i$ denote its `readonly` fields (it is assumed that $\mathcal{C}$ has some, otherwise the analysis doesn't say anything). Let's also assume that we are running a single analysis which uses its personal abstract domain.

Now, let $(C_j)_j$ be the constructors of $\mathcal{C}$ and $\widehat{f}_{i,j}$ be the abstract state of field f_i at the exit point of C_j. Then we can deduce a class invariant regarding f_i :

$$\widehat{\mathsf{Inv}}_i \quad = \quad \bigsqcup_j \widehat{f}_{i,j}$$

Thus, we have a set $(\widehat{\mathsf{Inv}}_i)_i$ of object invariants for $\mathcal{C}$.

This principle can then be applied independantly to the other analyses with their own abstract domain.

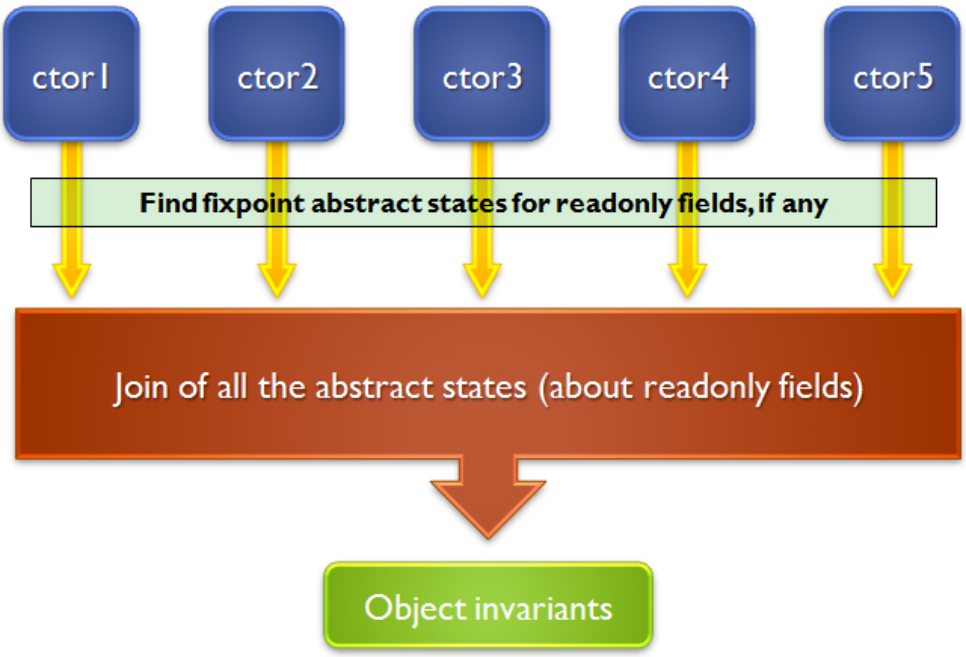

FIGURE 7 – Object invariants inference concerning `readonly` fields

Note here that the only informations we need concern constructors of the classes, so we can spare memory space (we don't even keep the informations about non-constructors and launch the analysis right after all constructors of a class have been analyzed).

Results

The performance of this analysis are unsurprisingly good once the analysis order optimization has been carried out, because it just asks for a few *join* operations.

However, we found out in practice that it does not infer that much invariants. Even on big DLLs with thousands of classes, it barely infers more than 10 invariants or so.

There could be several explanations for that matter of fact, among which we think the most likely are : the `readonly` field facility is not used a lot in system DLLs or the analyses run here were not precise enough to infer things about them and/or the idioms used in conjunction with `readonly` fields were too complicated to be analyzed in the first place.

One way to significantly improve the performance of this analysis would be to actually *infer* the `readonly` property on fields instead of waiting for the programmer to provide it.

We did not really look deep into that. The interesting part really was to *have* an object invariant analysis implemented in Clousot. The particular analysis we used here is just the first step towards the usage of this new framework.

3 Discussions and conclusion

We had a quick look at the two projects I made during my internship at MSR.

The first one aimed at recompiling inferred contracts so as to reuse them and it lead to a very valuable semantic checking on them. It actually was the first time that contracts could be really *examined*.

There are undoutbly still some minor adjustments to make about C# outputting from MSIL when new corner cases appear at usage. But the developed tool works properly even on the biggest system DLLs and provides the user with *out-of-the-box* compiling C# code.

My second project was about introducing object invariants analysis in Clousot. After that was done, I implemented a particular object analysis in this new framework.

The next step is now to implement some more sophisticated analyses in this framework which will lead to better results and will improve the overall deductive power of Clousot.