Chair of
Software Engineering

IMPROVING RELEVANCY OF DYNAMICALLY-INFERRED CONTRACTS IN EIFFEL

Master Thesis

By: Flaviu Roman
Supervised by: Nadia Polikarpova
Prof. Dr. Bertrand Meyer

Student Number: 08-906-489

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

inf | Informatik
Computer Science

Contents

1. Introduction

Professional software development has the goal to create computer programs that fulfill given requirements. This means firstly the existence of a specification (more or less formal) of what the software is required to do, and secondly a method that is able to validate the results according to the given specifications or requirements list. There is a long standing history in the software development for creating specifications and tools for both of these two aspects, nevertheless there are few satisfactory methods and tools available as yet. One of the difficulties rests in achieving a balance between human-produced high-level specifications, and machine-generated specifications that have larger precision.

With the advent of the object-oriented paradigm, a global system-level specification is decomposed by semi-formal methods (e.g. UML) into specifications for components, e.g. software classes. It is assumed that classes by their design have responsibilities to fulfill certain sub-goals mapped by such decomposition. In Eiffel, these responsibilities are described via interface protocols as formal contracts. Design by Contract™ is the technique which attempts to tighten together, at a smaller structural level, the software components and their intended behavior, thus ensuring independent correctness of lower-level system elements.

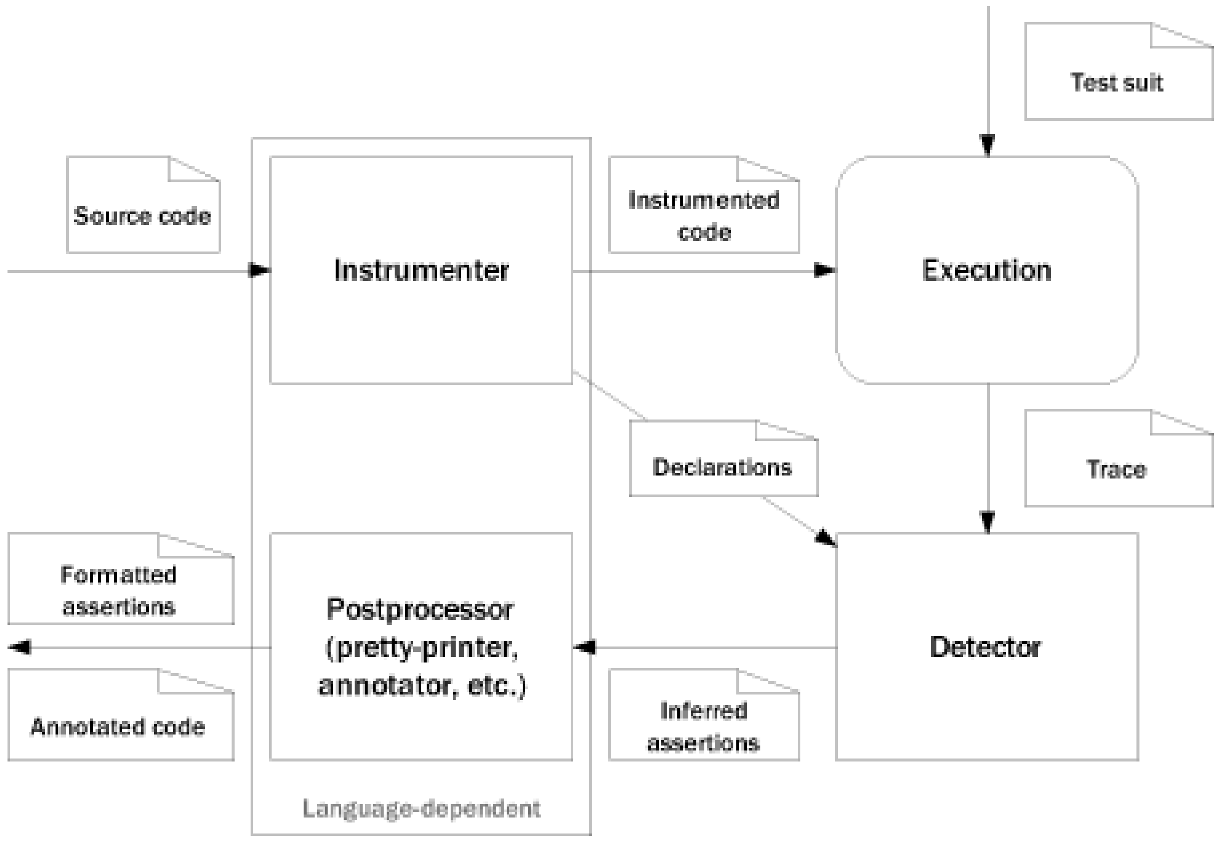
Adding contracts into programming improves the system, as long as contracts reflect the specifications and requirements both at component and system levels; this means they have to be both useful and correct. Writing contracts is specific only to a few languages and software development paradigms. For a long period of time it has been an assumption that most programmers are reluctant to produce the contracts themselves. Effort is made nowadays to aid the programmer by automatically inferring contracts from the written code so this could reduce contract-development time. This procedure can not replace programmer written contracts totally, but it can increase the quality. **Daikon** is one of such a tool that generates code invariants that can be inserted in code as contracts.

However the number of irrelevant inferred assertions is high. In the intention to overcome this deficit, as part of improving CITADEL, the Eiffel front-end for Daikon, the project aims to categorize, analyze and eliminate irrelevant assertions using heuristics.

2. Daikon

Daikon is a dynamic assertion detector that determines conditions holding throughout a software system execution. After these conditions have been found developers can assume that they also hold for all other executions and therefore form a specification (a set of contracts) of the system. To infer program properties Daikon observes values of certain variables at specific program points during program executions.

An overview of an automatic inference system is presented below.



Daikon can infer assertions over up to three variables. These can be either scalars or sequences.

CITADEL[CIT], the Daikon front-end for Eiffel, currently does not support sequences. The invariants inferred by Daikon are transformed into assertions that are included in the initial source code, in Eiffel language form, and are marked with special tags so they become easily recognized by searches or automatic recall engines.

The program points where CITADEL seeks for assertions are routine pre- and post-conditions, class invariants and loop invariants.

3. Loop invariants

A loop invariant is an invariant used to prove properties of loops. Specifically in Floyd-Hoare logic, the partial correctness of a while loop is governed by the following rule of inference:

$$\frac{\{C \wedge I\}\, body\, \{I\}}{\{I\}\, while(C)\, body\, \{\neg C \wedge I\}}$$

which means:

- a while loop does not have the side effect of falsifying *I*: if the loop body does not change an invariant *I* from true to false given the condition *C*, then *I* will still be true after the loop has run as long as it was true before;
- *while(C)*... runs as long as the condition *C* is true after the loop has run, if it terminates, *C* is false.

A theoretical approach[Fut93] which respects Hoare logic defines loop invariants as the weakest precondition on the induced topology *wp(DO, R)* of the sets *P(DO, R)* where *P* is a precondition, *DO* is the loop body, and *R* the postcondition. The authors further discuss finding invariants by means of generalizing the preconditions. Utility of programming with invariants is expressed by Paige[Pai86]. Standard procedures and strategies for loop invariants elaboration are also discussed by Gries[Gri82], while Havel[Hav79] concentrates on loops from a finite automata point of view.

The loop invariants are extremely useful for several operations. The literature presents studies of loop invariants checking for performing several types of improvements or verifications. The notion of loop invariants has been defined long ago, in conjunction to verification, beginning with Hoare[Hoa69] and Floyd[Flo67]. The theme is theoretically debated also by Huth and Ryan[Hut04]. Liu et al[Liu01] discuss methods of strengthening invariants for improving computation efficiency through incremental computation, while Weide et al[Wei08] assess loop invariants in the context of benchmarks for types of loops that deal with repeated computing operations and with collections.

An important study[Ngu04] deals with improving method tests and dynamic type checking upon loops, by eliminating redundant operations on types of objects that are loop invariant. A significant research[Nar05] has been made in reducing time and space of detecting software errors inside loops when employing loop invariant checks in testing embedded systems for scalability. In this case, loop invariant checking is performed instead of code duplication techniques, 26 recalling 62% of the errors caught by duplication while reducing time and memory demands with 80% and 4% respectively.

However there are no publicly established standard benchmarks for loop invariant inference. Loop invariants inference tests are usually employed by everyone using their own techniques on loops where the techniques performs quite well[Rod07]. Many have focused inference techniques (e.g. loops with a particular structure or invariants that are polynomial equalities) hence tests are usually performed on loop structures that match the technique.

4. Comparative study

The initial study[Pol08] has been made on the first version of the implementation of CITADEL. Several classes have been tested, taken from different environments, like industrial-grade Eiffel libraries, student-written code, and code used for teaching introductory programming.

Each class has been tested with small and large test suites. The test suites have been created with random generation of data for test. The results have been annotated using different classes for the inferred assertions, depending on their relation to the utility and correctness (i.e. correct and useful (not written by programmer), deducible (also written by programmer), false, uninteresting, other relevant, other irrelevant). The programmer written assertions may have also been annotated if they are not expressible in the automatic tool or if they are expressible but not inferred. Areas with no programmer written assertions but which contain automatically inferred ones have also been annotated.

Testing the results involved comparing the number of annotations for each class and discussing features that may have affected results in some way.

Results show that

- the large test suite provides more relevant and correct results, as expected for a dynamic verification software
- the automatic tool infers 5 times more useful contracts than programmers write. This is the main result of the work that strongly proves its utility and validates the need for future development of the tool. It is though an expected result, because it is evident that it is impossible for programmers to assert everything because of the complexity of the relations among objects
- the automatic tool recalls only half of the programmer written assertions. This result shows that programmers however are able to specialize better than computers in what relates to the software logic and software requirements. Therefore it is absolutely evident that programmer-written contracts cannot disappear, and such an automatic tool can never totally replace programmers in contract assertions
- the automatic tool has a 90% rate of inferring correct assertions. This is a great result that shows the exactness of the tool, especially due to the fact that using so few types of comparisons and being able to test only a few primitives still represents a sufficiently performing system that is able to supply correctness. The automatic tool has a 64% rate of inferring useful assertions. This result was expected to be lower than the correctness one, because of the impossibility of the tool to predict usefulness, as opposed to the simplicity of checking correctness. However, this result can be improved, by improving the tool. There are a number of theoretical cases where the tool cannot infer anything useful, therefore all these cases can be eliminated totally. This is also my main project issue, finding more and new methods of eliminating uninteresting assertions, and implementing them so this result can be improved.

5. Problem statement

As future developments for the initial study, the team has identified a number of possible improvements for the system, among which:

- filtering out uninteresting assertions (suppressions);
- push-button interface and precondition reduction;
- one-argument functions in unfolding;
- program point inheritance.

The current project concentrates on the first improvement, to deal with filtering out uninteresting assertions in loops (regarding unmodified variables) and in functions postconditions (assuming Eiffel functions are pure, assertions are only meaningful if they relate to the Result of that function). The other types of suppressions are currently unable to be implemented in CITADEL and are discussed as future developments.

The main expected result was to reduce uninterestness as much as possible as to be able to improve relevancy of dynamically inferred contracts for a programmer, who needs to deal with as strong but briefly-written assertions as possible.

6. Design procedures. Decisions and implementation

We decided to implement the filters through Daikon's suppression mechanism. The basic operation for loops involves creating a surrogate program point containing all the visible variables that are not modified inside the loop. For functions postconditions, the surrogate program point contains all variables except for *Result*. A variable is considered modified if it is the target of one of the following operations: assignment, assignment attempt, and creation instruction.

Daikon suppresses the invariants containing variables from these surrogate program points, therefore we obtain only assertions regarding modified variables for loops, and *Result* for functions postconditions.

In order to improve CITADEL to support suppressions, several modifications had to be done, both as design and as implementation.

The most significant were:

1. Since the suppression mechanism is quite a large part of CITADEL's construction, there was a need for the creation of a suppression cluster, useful for renamings. Renamings are necessary because the variables in the program point may have different names as the variables at the suppressee program point. The suppression cluster is presented in Figure 1, and provides the basis for implementing other types of suppressions as well.

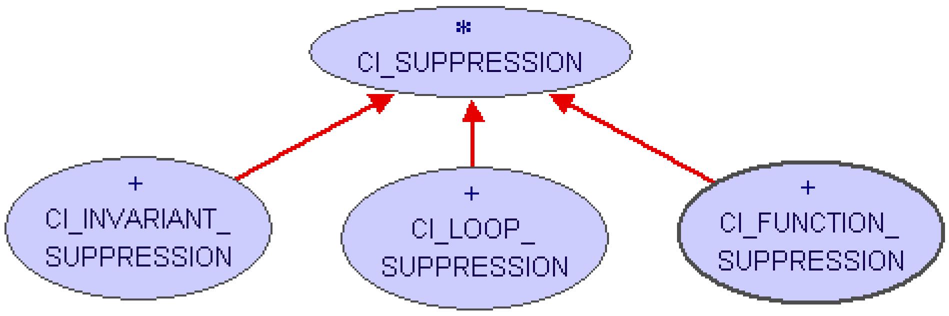


Figure 1. Suppression Cluster

2. Creation of surrogate program points for function exit and loop suppressions. Daikon supports suppressions if they are passed as "parent" program points to the current program point. A parent program point is processed just as a normal one, but the information yielded is instead used for the current program point and Daikon does not report any invariant detected in the current program point if it is also found in the parent point. Here also the removal of certain classes from the program points hierarchy, as they are no longer needed

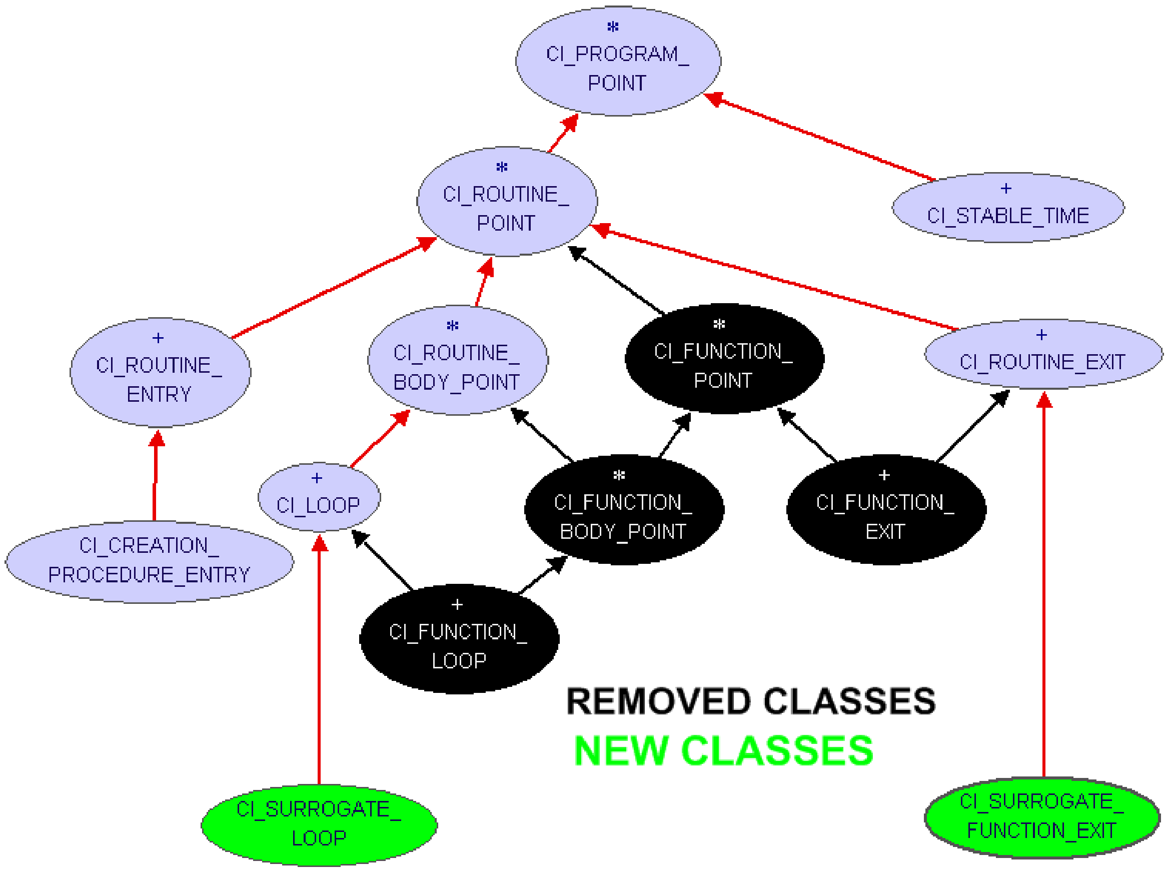


Figure 2. Changes in Program Point

3. Parsing the loops and function exits with suppressors and inserting the possibility to run with or without suppressions (for testing and comparative purposes)

```
create surrogate_loop.make (modified_entities, loop_point
  )
create suppression.make (surrogate_loop)
loop_point.suppressions.extend (suppression)
process_program_point (loop_point)
process_program_point (surrogate_loop)
```

7. Tests, results and comparisons

Testing procedures involved finding loops and expressing the theoretical invariants for them, then running the automatic tool and comparing the results with the one expected from the theoretical invariants. We searched for some loops in the domains of

- primitive types operations (division and remainder of naturals);
- arrays operations (maximum, sequence search);
- linked data structures (list reversal, nondestructive search in a stack with another stack).

Each loop was created inside its own class, to avoid interferences from a feature to another. Then we created test suites for each of the classes, executing a thousand calls to the feature involving the loop concerned. The running times for the execution of the instrumented system varied from a few seconds to a maximum of about 40 seconds.

7.1. Division and remainder of naturals:

```
divide_and_reminder (y, z: INTEGER_32)
     from
          r := y
          q := 0
          w := z
     until
          w > y
     loop
          w := 2 * w
     end
     from
     until
          w <= z
     loop
```

```
            q := 2 * q
            w := w // 2
            if w <= r then
                 r := r - w
                 q := q + 1
            end
       end
```

Loop invariants for loop 1:

- $z \le w$
- $w \le 2y$

Loop invariants for loop 2:

- $qw + r = y$
- $r < w$
- $w \ge z$

Item	Without Suppressions	With Suppressions
Deducible assertions from formal invariants	7	7
Uninteresting assertions	13	5

We observe from this results that the uninteresting assertions are reduced by more than half, while there is no deducible invariant that is lost in the process, which means the tool performs very well under primitives. A single type of invariant could not be detected because Daikon does not currently support it: the invariants over linear expressions with multiplications.

7.2. Maximum of an array

```
max (a: ARRAY [INTEGER_32]): INTEGER_32
     from
          i := a.lower + 1
          Result := a [a.lower]
     until
          i > a.upper
     loop
          if a [i] > Result then
               Result := a [i]
          end
          i := i + 1
     end
```

Loop invariants:

- $m = max_{j \in [a.lower..i-1]} a[j]$

- $i \geq a.lower + 1$
- $i \leq a.upper + 1$

Item	Without Suppressions	With Suppressions
Deducible assertions from formal invariants	2	2
Uninteresting assertions	19	0
False assertions	1	0

7.3. Sequential search in an array

```
index_of (a: ARRAY [G]; v: G): INTEGER_32
     from
          i := a.lower
     until
          i > a.upper or found
     loop
          if a [i] = v then
               Result := i
               found := True
          else
               i := i + 1
          end
     end
```

Loop invariants:

- $found \Rightarrow a[p] = v$
- $\neg found \Rightarrow \forall\, j \in [1 .. a.upper] : a[j] \neq v$
- $i \geq a.lower$
- $i \leq a.upper + 1$

Item	Without Suppressions	With Suppressions
Deducible assertions from formal invariants	2	2
Uninteresting assertions	17	0
False assertions	3	0

Results on arrayed collections eliminated all uninteresting assertions. Most of the inferred assertions without suppressions are related to the container features or traversal elements which, in the best case, should belong to the invariant of the container type class. CITADEL was unable to recall assertions in the form of universal and existence expressions.

7.4. Linked list reversal

```
reverse
     from
          tail := first
          first := Void
     until
          tail = Void
     loop
          first := tail
          tail := tail.right
          first.put_right (previous)
          previous := first
     end
```

Loop invariants:

- $seq(tail)^R + seq(first) = old\ seq(first)^R$

Item	Without Suppressions	With Suppressions
Deducible assertions from formal invariants	0	0
Uninteresting assertions	1	0
False assertions	2	0

7.5. Linked stack search

```
search (s: LINKED_STACK [G]; v: G)
     from
          found := False
          create temp.make
     until
          s.is_empty or found
     loop
          cv := s.item
          s.remove
          temp.put (cv)
          if cv = v then
               found := True
          end
     end
     from
     until
          temp.is_empty
     loop
          cv := temp.item
          temp.remove
```

```
                s.put (cv)
            end
```

Loop invariants for loop 1:

- $seq(s)+seq(temp)^R = old\ seq(s)$
- $v \in seq(temp) \Leftrightarrow found$

Loop invariants for loop 2:

- $seq(s)+seq(t)^R = old\ seq(s)$

Item	Without Suppressions	With Suppressions
Deducible assertions from formal invariants	0	0
Uninteresting assertions	10	0
False assertions	1	0

Results on linked structures show that CITADEL is yet unavailable to recall complicated assertions which have to be modeled probably with model-based contracts. Even so, the uninteresting assertions are eliminated using the suppressions procedures.

8. Overall relevancy improvement

We decided to use the following formula which reflects the relative reduction of the irrelevant assertions with suppressions with respect to without suppressions:

$$\frac{(n-r)-(n'-r')}{n}$$

where

- n = total number of inferred assertions without suppressions;
- n' = total number of inferred assertions with suppressions;
- r = total number of relevant inferred assertions without suppressions;
- r' = total number of relevant inferred assertions with suppressions.

Item	Relevance Improvement
Naturals Division	40.00%
Array Maximum	90.90%
Array Search	90.90%
Linked List Reversal	100.00%
Linked Stack Search	100.00%

Average	84.36%

9. Conclusions and future developments

The improvements work well and the project has proved to be successful. However, testing on only 5 classes is just a start and a possible future work may involve gathering more significant information by extending this study with more examples.

Improvements in filtering uninteresting assertions can be done by extending the suppression hierarchy. More precisely, some other classes of suppressions can be implemented as soon as CITADEL will support their intended operation. Among these there may be:

- supplier invariant: an assertion in a client may be suppressed by the invariant of the supplier. The higher depths of unfolding will CITADEL support, the more improvements this idea will bring
- supplier postcondition: same as above but with functions having arguments. CITADEL currently has no unfolding of such functions
- ancestor – descendant: improves coherence of assertions in ancestor classes based on their variants in the descendants. This will most likely bring significant improvement, but can only be implemented when inheritance support will be added to CITADEL

Improvements and future developments in CITADEL are[CITPRJ]: variable comparability, program point inheritance, unfolding functions with at least one argument, precondition reduction, etc.

An important development issue would be to integrate CITADEL into an IDE (EiffelStudio) and to perform its operations in a graphical user interface. The results of eliminating or reducing uninterestness obtained by these improvements to the system prove to be extremely useful for such a purpose.

10. References

[Fut93] Gerald Futschek. Algebraic properties of loop invariants. Formal Methods in Programming and Their Applications, 735:57–66, 1993.

[Pai86] Robert Paige. Programming with invariants. IEEE Software, 3(1):56–69, 1986.

[Gri82] David Gries. A note on a standard strategy for developing loop invariants and loops. Sci. Comput. Program., 2(3):207–214, 1982.

[Hav79] Ivan M. Havel. On two types of loops. In Mathematical Foundations of Computer Science, volume 74/1979 of Lecture Notes in Computer Science, pages 89–107. Springer Berlin / Heidelberg, 1979.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. Communications of the ACM, 12(10):576 – 580, 1969.

[Flo67] R. W. Floyd. Assigning meanings to programs. In American Mathematical Society Symposia on Applied Mathematics, volume 19, pages 19–31, 1967.

[Hut04] Michael Huth and Mark Ryan. Logic in Computer Science: Modelling and Reasoning about Systems. Cambridge University Press, New York, NY, USA, 2004.

[Liu01] Yanhong A. Liu, Scott D. Stoller, and Tim Teitelbaum. Strengthening invariants for efficient computation. Science of Computer Programming, 41(2):139–172, October 2001.

[Wei08] Bruce W. Weide, Murali Sitaraman, Heather K. Harton, Bruce Adcock, Paolo Bucci, Derek Bronish, Wayne D. Heym, Jason Kirschenbaum, and David Frazier. Incremental benchmarks for software verification tools and techniques. In Springer-Verlag, editor, VSTTE 2008 (Verified Software: Theories, Tools, and Experiments), volume LNCS, pages 84–98, October 2008.

[Ngu04] Phung Hua Nguyen and Jingling Xue. Strength reduction for loop-invariant types. In 27th Australasian Computer Science Conference (ACSC'04), pages 213 – 222, Dunedin, New Zealand, 2004.

[Nar05] Sri Hari Krishna Narayanan, Seung Woo Son, Mahmut Kandemir, and Feihui Li. Using loop invariants to fight soft errors in data caches. In 10th Asia and South Pacific Design Automation Conference, pages 1317–1320, January 2005.

[Rod07] E. Rodriguez-Carbonell and D. Kapur. Generating all polynomial invariants in simple loops. J. Symb. Comput., 42(4):443–476, 2007.

[Pol08] Nadia Polikarpova, Ilinca Ciupa, and Bertrand Meyer. A comparative study of programmer written and automatically inferred contracts. Unpublished. 2008

[Dai07] Daikon Invariant Detector User Manual, 2007.

[CIT] CITADEL. http://se.inf.ethz.ch/people/polikarpova/citadel.html

[CITPRJ] CITADEL connected projects. http://se.ethz.ch/projects/index.html