CITADEL User Manual
Version 1.1

Contents

1 What is *CITADEL*?

CITADEL (Contract Inference Tool Applying *Daikon* to the *Eiffel* language) is an *Eiffel* [1, 2] front-end for the *Daikon* assertion detector [3, 4].

1.1 *Daikon*

A dynamic assertion detector is a tool 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. Interesting program points can be, for instance, routine entries and exits. Variables are different expressions which make sense at a specific program point, such as the currently executing object, routine arguments, the return value of a function, attributes of other variables, etc.

Daikon maintains a list of assertion templates which it instantiates using such program variables at specified program points, and checks if they hold for the executions of the program through a given set of test cases. As soon as an assertion does not hold for an execution, it is eliminated and not checked again for further executions.

Daikon's dynamic contract inference system consists of several components, as shown in figure 1. The main steps involved in the contract inference process are the following:

1. An instrumenter modifies the program source so that, at certain program points, it saves the values of the variables in scope to a data trace file. The instrumenter also produces program point declarations (static information about program points and variables).

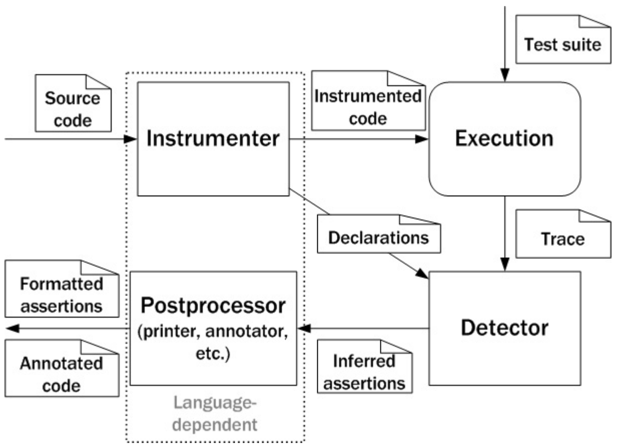

Figure 1: Dynamic contract inference system.

2. The instrumented program is exercised through a test suite. Each run of the program results in a data trace file.
3. *Daikon* instantiates assertion templates from its list using variables of appropriate types. This results in a list of potential assertions, which are then checked against the variable values recorded in the data trace files.
4. The inferred assertions can be post-processed, for instance by a pretty-printer, and inserted into the original source code as annotations.

Out of these components, only the instrumenter and the postprocessor depend on the programming language in which the original system is written. These two components form a *front-end* that allows the universal assertion detector to work for software systems written in different languages (and even with data that was generated through other means than during program execution). *CITADEL* is one of such front-ends that allows *Daikon* to work with software systems written in *Eiffel*. Other examples include *Chicory* (for *Java*), *Kvasir* and *Mangel-Wurzel* for *C/C++*, *dfepl* for *Perl*. Their description can be found in [4].

1.2 Contract inference in *Eiffel*

Contract inference can be used in *Eiffel* to strengthen programmer-written specifications (mainly, feature postconditions and class invariants). Sometimes it can also be used to correct existing specifications (strengthening preconditions).

Assertions produced by *CITADEL* are intended to become part of regular *Eiffel* contracts, hence the choice of program points and variables is constrained by language rules for assertions:

- Program points of interest in *CITADEL* correspond one-to-one to *Eiffel* assertions:
  - routine *entry* points correspond to preconditions;
  - routine *exit* points correspond to postconditions;
  - *stable time* points correspond to class invariants;
  - *loop* points correspond to loop invariants.
- The basic set of variables visible at each program point consists of entities that can be accessed from the corresponding assertions:

  - `Current` for stable times;
  - formals for creation procedure entry;
  - `Current` and formals for regular routine entry or procedure exit;
  - `Current`, formals and `Result` for function exit;
  - `Current`, formals and locals for a loop inside a procedure;
  - `Current`, formals, locals and `Result` for a loop inside a function.

- The full set of variables at each program point is obtained from the basic set by *unfolding* each of its variables, i.e. calling suitable queries on them. A query is suitable for unfolding if it is exported to both the surrounding class of the inferred assertion and to its guarantor (the class that should guarantee assertion fulfillment). Unfolding is performed to a fixed depth (e.g. in `Result.query1.query2` depth is 2), usually equal to 1.
- Due to the *uniform access principle* not only attributes are used in unfolding, but also zero-argument functions. Functions with arguments could also be used, but this feature is currently not implemented in *CITADEL*. In *Eiffel* it should be safe to use functions in unfolding, because they should always be pure (according to the *command-query separation principle*).
- When a function is used in unfolding it might be called in a state when its value cannot be calculated. Thus before calculating a function value, the instrumented system checks the programmer-written precondition for this function. If the precondition doesn't hold, the instrumented system outputs the standard *Daikon* value "nonsensical" instead of the real function value. This value is also used for queries called on a void target.
- For each variable from the full (unfolded) set the instrumented system will output its *identity*. What this identity actually is, depends on the *Eiffel* type of the variable:
  - If a variable has a type that corresponds to one of the types that *Daikon* understands (boolean, integer, real or string), then we take the variable value as an identity. We call such variables *printable*.
  - If a variable is not printable and has reference type, then its address is taken as an identity.
  - If a variable is not printable and has expanded type, some function of its fields' identities is taken as an identity.

  Daikon compares variables by their identities. Thus the notion of identity corresponds to equality semantics, that is if the identities of two variables are equal (for *Daikon*), then these variables should be equal (in the *Eiffel* sense). This is not true for strings, because they are compared by reference in *Eiffel* and by value in *Daikon*. *CITADEL* distinguishes between string-references and string-values, treating the former as reference variables and the latter as printable ones. String-references are used to infer equalities of strings and string-values are used for other assertions.
- The actual kind of identity for a variable is generally determined during runtime, based on its dynamic type, whereas the set of queries that are used in variable unfolding is determined by its static type. The precondition that should be checked for each query is also determined by the static type. Variables of generic types (containing formal parameters) are treated as if they had a base type (i.e. the constraining type).
- If once functions are used in unfolding they can be called in a different context that in the original system and thus yield a different value and spoil the execution. We don't know the perfect solution to this problem, that is why *CITADEL* offers several strategies for using once functions in unfolding:
  - don't use them at all;
  - use them like normal functions with a risk of changing system behavior;
  - before calling a once function in unfolding check whether it has already been called in the original context; if not, output the value as "nonsensical".

1.3 Design and Implementation of *CITADEL*

CITADEL consists of two main components: the *instrumenter* and the *annotator*. Besides, there exists a *CITADEL*-specific version of *Daikon* that can output inferred assertions in *Eiffel* format.

The instrumenter is the most complex part of the front-end. An input to the instrumenter consists of an *Eiffel* software system (a set of classes with a ".ecf" configuration file) and a set of clusters and classes to instrument. The instrumenter performs the following steps:

1. Syntax and type analysis of the original system.
2. Finding program points of interest in the abstract syntax tree (AST) of the original system.
3. Finding the basic set of visible variables at each program point of interest.
4. Unfolding variables.
5. Creating a declarations file, which contains — for each program point — its name, the names and types of its variables and other compile-time information.
6. Finding in the the AST right places to insert instructions that will output variable values to the trace file.
7. Generating these instructions (including the checks for each variable that it can be evaluated, i.e. checks for void targets and checks of preconditions).
8. Generating code that calculates the variable identity during runtime (in particular, that distinguishes between variables of printable, expanded and reference types).
9. Generating code that keeps track of the status of each once function (if this option is enabled).

The instrumenter's output consists of a declarations file and an *instrumented system*: instrumented classes, auxiliary classes (that work with the trace file, identities and once functions status) and a new configuration file.

After that *CITADEL* compiles and runs the instrumented system. This execution results in a trace file, which, together with the declarations file, is given as input to the *Daikon* detector. *Daikon* was modified in such a way that it produces textual form of assertions already in *Eiffel* format.

These assertions together with the original software system are then given to the annotator, which finds in the AST of the original system places where inferred assertions should be inserted. Besides that it does some postprocessing of assertions (to be removed in future versions of *CITADEL*). The annotator's output is the set of classes from the original system, which were chosen for processing, with inferred assertions added to their contracts.

For syntax and type analysis *CITADEL* uses the *Gobo Tools* library. To compile the instrumented system it uses the *ISE Eiffel* compiler.

Current limitations of *CITADEL* include:

- It is unable to instrument deferred and external features.
- Functions with arguments are not used in unfolding.
- It is not compatible with the new version of the *Gobo* library, which comes with *EiffelStudio* 6.2.
- It doesn't use the program point hierarchy that allows assertions from one program point to suppress corresponding assertions from the others.
- It doesn't implement variable comparability analysis that would allow to eliminate many uninteresting assertions (mainly, comparisons between unrelated variables).
- It doesn't support the online inference mode, when the execution trace is not written to a file, but is directly forwarded to *Daikon*.
- It doesn't support concurrent programs.

2 Compiling *CITADEL*

To compile *CITADEL* you need:

- *CITADEL* sources available from http://se.inf.ethz.ch/polikarpova/citadel;
- *EiffelStudio* 6.1 available from www.eiffel.com;
- *Erl-G* library version 1.3.1 (e.g. revision 740) available from http://se.ethz.ch/people/leitner/erl_g;
- *Autotest* sources version 1.3.2 (e.g. revision 740) available from http://se.ethz.ch/research/autotest;
- *EiffelStudio* sources revision 70887 available from http://eiffelstudio.origo.ethz.ch.

The *CITADEL* configuration file refers to the home directories of *EiffelStudio* sources, *Erl-G* and *Autotest* as `EIFFEL_SRC`, `ERL_G` and `AUTO_TEST` correspondingly. You should set these environment variables to wherever you put the libraries mentioned above.

The main configuration file for *CITADEL* is `src/citadel/citadel.ecf`.

3 Using CITADEL

To use *CITADEL* you additionally need the *CITADEL*-specific version of *Daikon* (both sources and compiled version are available from http://se.inf.ethz.ch/polikarpova/citadel). If you don't want to modify the sources you only need the compiled version, which consists of a single file `daikon.jar`. Set the environment variable `DAIKONDIR` to wherever you put this file. *Daikon* uses *Java*, so you should have *JRE* or *JDK* installed and one of the environment variables `JREDIR` or `JDKDIR` set to wherever *JRE* or *JDK* is.

You will usually run *CITADEL* using the following syntax:

```
citadel ecf_filename {-l cluster_to_instrument}* {-c class_to_instrument}*
```

If no clusters are specified, then all clusters from the system are chosen, and if no classes are specified, then all classes from the chosen clusters are chosen. So, for example, to instrument a single class `TESTER` from cluster `example` in system `example`, write

```
citadel example.ecf -l example -c TESTER
```

(if you write just `citadel example.ecf -c TESTER`, it will choose `TESTER` from `example` plus all classes from all other clusters in the system).

The full list of *CITADEL* options can be found in the table below.

Full name	Short name	Argument	Description
`help`	`h`		Output help message and exit
`version`	`v`		Output version information and exit
`quiet`	`q`		Don't output progress information
`target`	`t`	string	Target that should be used for contract inference (if the original system has multiple targets)
`cluster`	`l`	string	Cluster to instrument
`class`	`c`	string	Class to instrument
`output`	`o`	string	Output directory for instrumented and annotated systems and auxiliary files
`start`	`s`	integer	Step to start with (1 = instrument original system; 2 = compile instrumented system; 3 = execute instrumented system; 4 = infer assertions; 5 = annotate original system). Default value is 1
`end`	`e`	integer	Step to end with. Default value is 5

...

...

`use-old-decl-format`	`m`		Use the old version of the daikon declaration file format
`unfolding-depth`	`u`	integer	Depth to which variables are unfolded. Default value is 1
`impure`	`g`	string	Name of impure function that should not be used in unfolding in format `CLASS_NAME.function_name`
`only-attributes`	`a`		Don't use functions in unfolding (only attributes)
`unfold-printable`	`p`		Unfold variables of printable types (significantly enlarges output, but may give some useful information about variables of printable types)
`any-queries`	`y`		Use queries inherited from class `ANY` in unfolding (may significantly enlarge output)
`obsolete-queries`	`b`		Use obsolete queries in unfolding
`inapplicable-queries`	`z`		Use inapplicable queries in unfolding
`once-status`	`n`	integer	Once functions status tracking mode (0 = once functions are not used in unfolding; 1 = only once functions from instrumented classes are used in unfolding; 2 = all once functions are used, but requires modification of the whole system, not just instrumented classes; 3 = once functions are used without watching their status (can lead to system behavior alteration). Default value is 1
`finalize`	`f`		Finalize instrumented system before running
`instrumented-args`	`i`	string	Arguments to pass to instrumented system
`daikon-args`	`d`		Arguments to pass to *Daikon*
`java-memory`	`j`	integer	Memory size for JVM when running Daikon and PrintInvariants
`disable-suppression`	`r`		Disable suppression of pre- and postconditions by class invariants (always enabled with -m)
`disable-nonres-suppression`			Disable suppression of pure function postconditions not involving `Result` (always disabled with -m)
`disable-loop-suppression`			Disable suppression of loop invariants not involving variables modified in the loop body (always disabled with -m)
`disable-supplier-suppression`			Disable suppression of clauses by class invariants of the involved variables (always disabled with -m)
`keep-original`	`k`	string	What should *CITADEL* do with original contracts in the annotated system (0 = remove; 1 = keep; 2 = flatten). Default value is 2
Full name	Short name	Argument	Description

4 Tests

To try *CITADEL* out you may use the set of tests available from the website. These tests exercise 25 classes from widely-used libraries (*EiffelBase*, *Gobo*, *MML*), *Traffic* application and student projects.

All the tests are joined into a single system `tests.ecf`. To compile this system except for *EiffelStudio* 6.1 you will additionally need:

- *MML* library available from https://svn.origo.ethz.ch/eiffelstudio/branches/eth/ballet/mml;
- *Traffic* application (version 3.3.1079) available from http://traffic.origo.ethz.ch.

All the tests are unit tests of single classes. To test a particular class uncomment the creation of a corresponding tester class in the root class `TESTER`. Then run the testing system under *CITADEL*, instrumenting the class under test. For example, to obtain inference results for class `TRAFFIC_BUILDING` you may run *CITADEL* with the following arguments:

```
citadel <tests_dir>\tests.ecf -o <output_dir> -d "--config
    <citadel_dir>\daikon_config" -l traffic/building -c TRAFFIC_BUILDING
```

References

[1] *B. Meyer* Object-Oriented Software Construction, 2nd Edition. Prentice Hall, 1997.

[2] *Standard ECMA-367. Eiffel*: Analysis, Design and Programming Language. ECMA International, 2006. http://www.ecma-international.org/publications/standards/Ecma-367.htm

[3] *M. D. Ernst* Dynamically Discovering Likely Invariants. PhD Dissertation. University of Washington, 2000.

[4] Daikon *Invariant Detector User Manual*, 2007.