

**list**

CEA/SACLAY  
DIRECTION DE LA RECHERCHE TECHNOLOGIQUE  
DEPARTEMENT INGENIERIE LOGICIELS ET SYSTEMES

cea

energie atomique - energies alternatives

## Département Ingénierie Logiciels et Systèmes

Saclay, le 17 mars 2011

REF. : DRT/LIST/DILS/2011-0069/FV

**Par**

**DRT/LIST/DILS/LMeASI (CEA)**



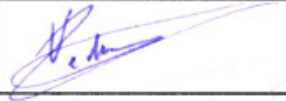


DIRECTION DE LA RECHERCHE TECHNOLOGIQUE  
LABORATOIRE D'INTEGRATION DES SYSTEMES ET DES TECHNOLOGIES  
DEPARTEMENT INGENIERIE LOGICIELS ET SYSTEMES  
CEA/SACLAY – 91191 GIF-SUR-YVETTE CEDEX  
TÉL . (33)1 69 08 55 32 - FAX (33)1 69 08 83 95 – E-MAIL : fabrice.derepas@cea.fr  
Etablissement Public à caractère Industriel et Commercial  
R.C.S. PARIS B 775 685 019

DIRECTION DE LA RECHERCHE TECHNOLOGIQUE  
LABORATOIRE D'INTEGRATION DES SYSTEMES ET DES TECHNOLOGIES  
DEPARTEMENT INGENIERIE LOGICIELS ET SYSTEMES  
CEA/SACLAY – 91191 GIF-SUR-YVETTE CEDEX  
TÉL . (33)1 69 08 55 32 - FAX (33)1 69 08 83 95 – E-MAIL : [fabrice.derepas@cea.fr](mailto:fabrice.derepas@cea.fr)  
Etablissement Public à caractère Industriel et Commercial  
R.C.S. PARIS B 775 685 019

**Ce document et les informations qu'il contient, sont la propriété exclusive du CEA. Ils ne peuvent pas être communiqués ou divulgués sans une autorisation préalable du CEA LIST**

Identification : Rapport LIST/DILS/2011-0069/FV  
 Titre : API – Static Analysis of Numerical Accuracy in SCADE™  
 Auteurs : F. Védrine, X. Fornari  
 Mot clés : Static Analysis, Numerical Properties, Design Tool, SCADE™, FLUCTUAT  
 Unité : DRT/LIST/DILS/LMeASI  
 Résumé :

Ce document est la propriété du CEA. Il ne peut être reproduit ou communiqué sans son autorisation.  
 This document is the property of the CEA. It can not be copied or disseminated without its authorization.

	REDACTEURS <i>WRITERS</i>	VERIFICATEUR <i>CONTROLLER</i>	CHEF DE DEPARTEMENT <i>HEAD of DIVISION</i>
NOM	F. Védrine	E. Gault	Fabrice DEREPA
DATE	13/05/2011	13/05/2011	13/05/2011
SIGNATURE			

**Suivi des modifications :**

<b>Date</b>	<b>Version</b>	<b>Rédacteur(s)</b>	<b>Description des évolutions</b>
17/03/2011	0.1	F. Védrine	Initial version
13/05/2011	1.0	F. Védrine	Final version

**Diffusion : Esterel Technologies**  
**LMeASI**  
**Interested Consortium**

## Table of Content

### Table of Content

Application Programming Interface Static Analysis of Numerical Accuracy .....	4
1 API for numerical accuracy analysis.....	5
1.1 Functions in the API.....	5
1.2 Binary executables attached to the numerical analysis .....	8
1.3 Library functions for numerical assertions.....	9
1.4 Format of the evolution files .....	11
1.5 Analysis parameters .....	11
2 Example: SCADE coupling .....	13
2.1 Setup.....	13
2.2 Analysis .....	14
2.3 Getting results .....	15
3 Références .....	17

# Application Programming Interface

## Static Analysis of Numerical Accuracy

This document describes the Application Programming Interface (API), used to launch a static analysis based on the Abstract Interpretation framework and targeted to bound the numerical errors – rounding errors and numerical noise –. This API exports some Abstract Interpretation facilities like precise ranges for the numerical domains and for the numerical errors or like the evolution graphs. The API is defined in the C language for portability reasons – different operating systems, different compilers.

The dynamic library attached to the API is proposed for SCADE<sup>TM</sup> users, thanks to an executable that orders the different calls to the functions in the API. The library is implemented by FLUCTUAT [1] functionalities. Loading a SCADE model in the static analyzer requires the model be available through files. Such files correspond to the set of preprocessed C source files issued from the SCADE model by the KCG translator. Loading the analysis parameters also requires the parameters be described in a file, based on the XML style.

The document is organized as follows. First we describe the functionalities provided by the API. Then we describe how is realized the coupling for its use in SCADE<sup>TM</sup>.

# 1 API for numerical accuracy analysis

The functions in the API enable to create an analysis project, then to launch and to control the static analysis of numerical properties and finally to view the analysis results in an analysis report.

The API is implemented with FLUCTUAT [1] functions. It is an extensible API. An interesting extension is to add analysis functions with arguments whose type is “call-back” functions. The objective of such an extension is to improve the interactions between the user and the current analysis. The user can exploit such functionalities to debug and to refine his current model.

A user of the API functions should call them in the following order:

```
Interface* interface = createInterface();
Variable variable;
setResourceFile(interface, "resource.rc");
setOutputDirectory(interface, "output");
addPreprocessedSourceFile(interface, "module_scade_1.c");
addPreprocessedSourceFile(interface, "module_scade_2.c");
addPreprocessedSourceFile(interface, "module_scade_3.c");

setLookAtLocalVariable(interface, "speed", 18, "module_scade_1.c");
setLookAtLocalVariable(interface, "height", 28, "module_scade_2.c");

analyze(interface, "main");

while (retrieveVariableAndResult(interface, &variable)) {
    printf("variable %s with ident %d in [%e, %e] with error[%e, %e]\n",
        variable.name, variable.ident, variable.minDomain, variable.maxDomain, variable.minError, variable.maxError);
    printf("\tvariable locations: declaration=%s,%d last_write=%s,%d\n",
        variable.fileDeclaration, variable.lineDeclaration, variable.fileLastAssign);
};
```

The executable `evol_viewer` can then display the existing evolution graphs attached to a variable. As an example,

```
evol_viewer output speed 2
```

displays the evolution of the interval containing the absolute numerical errors of the variable speed with respect to the simulation cycles. The evolution graphs exist for the variable speed if a “`print(speed)`” annotation is present in the file “`module_scade_1.c`”.

## 1.1 Functions in the API

```
Interface* createInterface();
```

The function creates an analysis project, as the result of the function call. The project is then able to read annotated C source files, to launch a static analysis and to expose the results. At the end, the function [freeInterface](#) free all the resources allocated by the project.

```
bool addPreprocessedSourceFile(Interface* interface, char* filename);
```

The function adds the file filename in the list of the preprocessed C source files to be analyzed by the project interface. The code in filename is likely to contain calls to library function to specify numerical assertions. The available library functions are listed in section 1.3.

This function should be called on the whole set of the source files before any call to the [analyze](#) function. If a functional call is present in any source file, then the body/definition of the called function should be present in the code of one and only one source file (link constraint).

The result of this function is true in case of success – the file has been added in the list of files to be analyzed. The result of this function is false in the other cases. A failure can come from the absence of the file filename or from a syntactic problem in the file filename.

```
bool setResourceFile(Interface* interface, char* filename);
```

The function loads the whole set of parameters for the analysis project interface. All the non-default analysis parameters should be present in the file filename (see the whole description in section 1.5).

This function is called at most one time before the [analyze](#) function. In the case the parameters are not defined, they are set to their default value.

The result of this function is true in case of success and false in case of failure. A failure can come from the absence of the file filename or from a syntactic problem in the file filename.

```
bool setLookAtVariable(Interface* interface, unsigned int* ident, char* variableName, int lineSource, char* fileSource);
```

The function puts variableName in the list of the variables/signal whose value should be present in the analysis report. By default, only the global variables and the variables of the main function are reported in the analysis report. This function potentially enables to flag all variables of the source code.

The analysis report has to give an over-approximation of all the values and of all the numerical errors for variableName at the line lineSource in the file fileSource. Note that fileSource should have been added by the function [addPreprocessedSourceFile](#).

The parameter ident is a unique identifier that identifies the variable and the position in the source files when the value appears in the report. At the end of the function, ident has been automatically incremented. This identifier will be returned by the function [retrieveVariableAndResult](#) after the end of the analysis.

The result of this function is true in case of success: the variable variableName is well defined at the line lineSource in the file fileSource. The result is false in the other cases.

```
bool setInterestingVariable(Interface* interface, char* variableName, char* functionName, void* node);
```

The function puts variableName in the list of the variables/signal whose value should be in the analysis report. The main difference with the function [setLookAtVariable](#) is the parameters. It is an optimization when the C source code is in the “Static Single

Assignment” syntax, which guarantees a unique assignment/definition for each variable. So the name of the function `functionName` is sufficient to define a valid location to look at the domain and at the numerical errors after the assignment of `variableName`.

The analysis report has to give an over-approximation of all the values and of all the accumulated numerical errors for `variableName` at the end of `functionName`.

The parameter `node` is a unique identifier that identifies the variable and the position in the source files when the value appears in the report. This identifier will be returned by the function [retrieveVariableAndResult](#) after the end of the analysis.

The result of this function is true in case of success: the variable `variableName` is well defined in the function `functionName`. The result is false in this other cases.

```
void setOutputDirectory(Interface* interface, char* name);
```

The function indicates that the intern analysis results are to be stored in the directory name. In particular this directory should contain the evolution files.

The evolution files are produced when the analysis interprets a call to the library function `__DAED_DOUBLE_PRINT` in the C preprocessed source files (see section 1.4). In this case, it updates the evolution files attached to the printed variable, files stored in the directory name. When the analysis is complete, the graphical interface can show the evolution files thanks to the executable `evol_viewer`.

```
bool analyze(Interface* interface, char* entryFunction);
```

The function launches the static analysis of numerical accuracy for the function named `entryFunction`. As soon as the analysis has ended, the numerical results are available in the project interface. The function [retrieveVariableAndResult](#) retrieves them to build the analysis report.

The function returns the value true if the analysis has been completed and the value false if the analysis has failed on a syntactic problem or an internal error.

The static analysis performed is a forward analysis by Abstract Interpretation. It infers the ranges and the relations of the domains and of the numerical errors for each computation. In this framework, the domains can be over-approximated to ensure the termination of the analysis. The relational information used by the analysis enables to obtain very precise ranges for the set of all possible values and for the set of the numerical errors.

When the domains are too much over-approximated, the relational information also enables to determine the precise origin of the over-approximation. An interaction with the end-user in the graphical user interface is so possible with this kind of information and with additional “call-back” parameters to this function.

```
bool retrieveVariableAndResult(Interface* interface, Variable* variable);
```

The function retrieves the next variable for which the analysis has delivered results. The numerical results are available in the fields of `variable`, different from `variable->name` that just refers to the name of the variable.

The function returns the value false when all the flagged variables have been covered. The function returns the value true in the other cases. In these cases, the fields of variable are updated with the results of the analysis.

The field `variable->name` contains the name of the variable.

The fields `variable->lineSource` and `variable->fileSource` correspond to the time (line number and file name) when the values and the numerical errors are collected for the variable `variable->name`. At these times, the set of all possible values is included in the interval [`variable->minDomain`, `variable->maxDomain`] and the set of all possible numerical errors is included in the interval [`variable->minError`, `variable->maxError`]. If the function [setLookAtVariable](#) has flagged the variable `variable->name` in the numerical report, then `variable->lineSource` and `variable->fileSource` are exactly the line and the source file parameters of the call to [setLookAtVariable](#).

The fields `variable->lineDeclaration` and `variable->fileDeclaration` correspond to the declaration of the variable `variable->name`. These fields define a unique identifier for the variable.

The fields `variable->lineLastAssign` and `variable->fileLastAssign` correspond to the time (line number and file name) when the variable `variable->name` has been assigned in the current simulation cycle. These fields have sense when the preprocessed source file uses the “Static Single Assignment” syntax. These fields are located between `variable->...Declaration` and `variable->...Source`. They enable to retrieve the data flow information of the analyzed code.

The field `variable->ident` is a unique identifier for the numerical result of the static analysis of variable, if it is greater than 0. If the function [setLookAtVariable](#) has flagged variable for the report, `variable->ident` is exactly the identifier parameter of the call.

The fields `variable->minDomain` and `variable->maxDomain` correspond to an over-approximated range for the set of all possible values that the variable `variable->name` can take at the times specified by `variable->...Source`.

The fields `variable->minError` and `variable->maxError` correspond to an over-approximated range for the set of all possible numerical errors that can perturb the variable `variable->name` at the times specified by `variable->...Source`.

## 1.2 Binary executables attached to the numerical analysis

```
evol viewer output_directory variable_name code
```

The executable [evol viewer](#) shows the evolution graph attached to a variable. This display is available at the end of the analysis [analyze](#).

The analysis should have built a part of the evolution graph each time it has encountered a library call to the function `__DAED_DOUBLE_PRINT` in the preprocessed C source files with `variable_name` as a parameter (see section 1.4). The evolution part concerns the interval containing the set of all possible values and the interval containing the set of all possible errors for `variable_name`. The sequence of these intervals defines the evolution graph, cycle by cycle. If subdivisions are applied, the evolution graph makes the correct distinction between the different subdivisions. The evolution graph is stored in the directory `output_directory` if the function [setOutputDirectory](#) was called with this name. The file format of the evolution graphs is defined in section 1.5.

code is a bitfield to choose the initial evolution graph to show for variable\_name:

- code & 1  $\neq$  0  $\Rightarrow$  displays the evolution graph of the interval of the floating point values.
- code & 2  $\neq$  0  $\Rightarrow$  displays the evolution graph of the interval of the numerical errors as absolute errors.
- code & 4  $\neq$  0  $\Rightarrow$  displays the evolution graph of the interval of the real values.
- code & 8  $\neq$  0  $\Rightarrow$  displays the evolution graph of the interval of the numerical errors as relative errors.

### 1.3 Library functions for numerical assertions

In order to collect all the possible executions, the user has to provide the domain for the input data of the analyzed function. The natural description for such domains is an interval based representation. Some library functions are available to enter hypotheses on the domains in the preprocessed source code. The more precise are the hypotheses, the more precise is the static analysis.

The list of the available library functions follows:

- `__BUILTIN_DAED_DBETWEEN(a, b)` specifies an input interval whose (64 bits) floating point values are in the interval [a, b]. The real value is the same value as the floating point value. In particular the function does not introduce any numerical error/noise.
- `__BUILTIN_DAED_FBETWEEN(a, b)` specifies an input interval whose (32 bits) floating point values are in the interval [a, b]. The real value is the same value as the floating point value. In particular the function does not introduce any numerical error/noise.
- `__BUILTIN_DAED_IBETWEEN(a, b)` specifies an input interval whose integer values are in the interval [a, b]. The real value is the same value as the integer value.
- `__BUILTIN_DAED_DBETWEEN_WITH_ULP(a, b)` specifies an input interval whose floating point values are in the interval [a, b]. This library function indicates that the floating point values come from ideal real values – with infinite accuracy – that are stored in 64 bits floating point registers. The rounding error is so limited to an ULP (Unit in the Last Place) with respect to the considered floating point value.
- `__BUILTIN_DAED_FBETWEEN_WITH_ULP(a, b)` specifies an input interval whose floating point values are in the interval [a, b]. This library function indicates that the floating point values come from ideal real values – with infinite accuracy – that are stored in 32 bits floating point registers. The rounding error is so limited to an ULP (Unit in the Last Place) with respect to the considered floating point value.
- `__BUILTIN_DAED_DOUBLE_WITH_ERROR(a, b, c, d)`,  
`__BUILTIN_DAED_FLOAT_WITH_ERROR(a, b, c, d)`,  
`__BUILTIN_DAED_INT_WITH_ERROR(a, b, c, d)`,  
specifies an input interval whose floating point values (64 bits, 32 bits) or integer values are in the interval [a, b]. This library function indicates that the numerical errors are in the interval [c, d]. So the ideal real values are asserted to be in the interval [a+c, b+d].
- `__BUILTIN_DAED_DOUBLE_WITH_REL_ERROR(a, b, c)`,  
`__BUILTIN_DAED_FLOAT_WITH_REL_ERROR(a, b, c)`  
specifies an input interval whose floating point values (64 bits, 32 bits) or integer values are in the interval [a, b]. This library function indicates that the numerical errors

are bound by a relative error of  $x\%$  with  $x = 100 \times c$ . This point asserts that the ideal real values are in the interval  $[(1 - \text{sign}(a) \times c) \times a, (1 + \text{sign}(b) \times c) \times b]$ .

- `__BUILTIN_DAED_DREAL_WITH_ERROR(a, b, c, d)`,  
`__BUILTIN_DAED_FREAL_WITH_ERROR(a, b, c, d)`,  
`__BUILTIN_DAED_IREAL_WITH_ERROR(a, b, c, d)`  
 specifies an input interval whose ideal real values are in the interval  $[a, b]$ . This library function indicates that the numerical errors are in the interval  $[c, d]$ . So the floating point values (64 bits, 32 bits) or integer values are asserted to be in the interval  $[a+c, b+d]$ .
- `__BUILTIN_DAED_LIMIT_DOUBLE(x1, a, b)`,  
`__BUILTIN_DAED_LIMIT_FLOAT(x1, a, b)`,  
`__BUILTIN_DAED_LIMIT_INT(x1, a, b)`,  
 filters the floating point values (64 bits, 32 bits) or the integer values carried by the variable  $x1$  by limiting them in the interval  $[a, b]$ . This user's assertion indicates there is no data test reaching this instruction with a value for the variable  $x1$  outside  $[a, b]$ . The numerical error is the original numerical error of  $x1$ .
- `__BUILTIN_DAED_LIMIT_DREAL(x1, a, b)`,  
`__BUILTIN_DAED_LIMIT_FREAL(x1, a, b)`,  
`__BUILTIN_DAED_LIMIT_IREAL(x1, a, b)`,  
 filters the ideal real values carried by the variable  $x1$  by limiting them in the interval  $[a, b]$ . This user's assertion indicates there is no data test reaching this instruction by a sequence of ideal computations with a real value for the variable  $x1$  outside  $[a, b]$ . The numerical error is the original numerical error of  $x1$ , which provides restrictions for the floating point values (64 bits, 32 bits) of integer values for  $x1$ .
- `__BUILTIN_DAED_LIMIT_DERROR(x1, a, b)`,  
`__BUILTIN_DAED_LIMIT_FERROR(x1, a, b)`,  
`__BUILTIN_DAED_LIMIT_IERROR(x1, a, b)`,  
 filters the numerical errors carried by the signal/variable  $x1$  by limiting them in the interval  $[a, b]$ . This user's assertion indicates there is no data test reaching this instruction with a rounding error for the signal  $x1$  outside  $[a, b]$ . The real ideal value is the value of  $x1$ , which provides restrictions for the floating point values (64 bits, 32 bits) of integer values for  $x1$ : the new implementation value of  $x1$  is in  $[\min(x1)+a, \max(x1)+b]$  (in the relational mode that is the default mode)
- `__BUILTIN_DAED_FGRADIENT(x0_min, x0_max, gradient_min, gradient_max, x_min, x_max, i, i_0)`  
 limits a signal behavior with respect to its value at the previous cycle. The initial value of the signal at the cycle  $i_0$  is in the interval  $[x0\_min, x0\_max]$ . The difference between two successive iterations from the cycle  $i_0$  is in the interval  $[\text{gradient\_min}, \text{gradient\_max}]$ . All the possible values for the iterations after  $i_0$  are in the interval  $[x\_min, x\_max]$ . The parameter  $i$  is the current cycle. This user's assertion is used to specify the external environment.
- `__BUILTIN_DAED_DPRINT(s)`  
`__BUILTIN_DAED_FPRINT(s)`  
`__BUILTIN_DAED_IPRINT(s)`  
 asks the analysis to generate the evolution graphs for the signal/variable  $s$  of type floating-point (64 bits, 32 bits) or integer. The files describing the evolution graphs are `s.evolval`, `s.evolerr`, `s.evolvalexacte`. These files are read by the command [evol\\_viewer](#) to display them graphically.

## 1.4 Format of the evolution files

The evolution files have a common format for describing the evolution of the implementation values (file variable.evoldal), the evolution of the exact values (file variable.evoldalexacte) or the evolution of the numerical errors (file variable.evoldalerr).

Each entry/line in those files defines a point in the evolution graph. An entry has 4 fields.

The first field defines the primary global subdivision. This field is 0 if no primary subdivision is specified.

The second field defines the secondary global subdivision. This field is 0 if no secondary subdivision is specified.

The third and fourth fields define the interval containing all the possible floating-point or integer values of the implementation (file variable.evoldal), the interval containing all the possible ideal real values (file variable.evoldalexacte) and the interval containing all the possible numerical errors (file variable.evoldalerr) for the variable « variable » in the current cycle.

## 1.5 Analysis parameters

This section gives a definition for the analysis parameters that are used for a static analysis of numerical accuracy of C source code. The parameters are provided thanks to a resource file to be supplied to the function [setResourceFile](#).

The resource file for the analysis' parameters contains the following information. For a more detailed description the reader can consult the User's Manual of FLUCTUAT [1].

- `analysiskind = 5`, to use a relational analysis for the domains and for the numerical errors, which is the default analysis.
- `entryname = main`, to specify the entry point of the static analysis. By default, this entry point is the main function. The static analysis so collects the numerical properties at each cycle to give an over-approximation of all the possible values and of all the possible numerical errors at the end of every cycle.
- `mpfrbits = 60`, to specify the number of MPFR [4] bits used for the intern computations.
- `nbexecbeforejoin = ...`, to specify the cycle number from which the analyzer starts to merge the numerical properties. This parameter is fixed to the value 10 by default. It strongly depends on the analyzed code.
- `nbrelationlevel = -1`, to specify an unbound number of nested loops, for which the analysis keeps a relational history.
- `unfoldloopnumber = ...`, to specify how many times the main loop is unrolled. This parameter is fixed to the value 10 by default. This parameter is important for components including numerical filters and in this case, it depends on the convergence properties of the filters.
- `keeprelations = 1`, to keep precise relational properties at the end of loops.
- `maxiter = ...`, to bound the maximum number of iterations.
- `wideningthreshold = ...`, to specify the cycle number from which the analyzer starts to use widening operators instead of merge operators for collecting the numerical properties.
- `narrowingthreshold = ...`, to specify the cycle number from which the analyzer starts to use narrowing operators after the widening steps have been applied.
- `progressivewidening = 1`, to use progressive widening operators.

- `infthreshold = ...`, to specify a threshold before triggering the widening operators.
- `calculexacterr = 0`, to specify the presence of errors for numerical constants if they are not exactly represented.
- `subdivide = 0`, to specify the absence of global subdivision.
- `anasubfile = ...`, to specify the source file in which the primary subdivision is defined.
- `linenumber = ...`, to specify the line in the source file containing the variable to primarily subdivide.
- `slices = ...`, to specify the number of slices contained in the primary subdivision.
- `anasubfile2 = ...`, to specify the source file in which the secondary subdivision is defined.
- `linenumber2 = ...`, to specify the line in the source file containing the variable to secondarily subdivide.
- `slices2 = ...`, to specify the number of slices contained in the secondary subdivision.

## 2 Example: SCADE coupling

The functions in the API are organized in 3 groups. The first group (section 2.1) enables the creation of analysis projects, taking the model and the analysis parameters into account. The second group (section 2.2) enables to launch and to control the static analysis. The third group (section 2.3) enables to view the analysis' results, in particular in the HTML report accessible in SCADE™.

The API is extensible. An interesting extension is to add functions in the second group that have “call-back” functions as arguments. The objective of such an extension is to improve the interactions between the end-user and the current analysis (see section 2.2). The user can exploit such functionalities to debug and to refine the current model.

### 2.1 Setup

An analysis project of a SCADE model starts with the call to the [createInterface](#) function. Then the function [addPreprocessedSourceFile](#) enables to load the SCADE model in the analysis project.

When the SCADE interface calls the function [addPreprocessedSourceFile](#), the newly added file comes from the SCADE model after a KCG translation into the C language syntax. The KCG translation is likely to introduce library calls as annotations/assertions that specify some numerical data. The available libraries functions are listed in section 1.3. [addPreprocessedSourceFile](#) should be called on the whole set of the files generated by KCG before any call to the [analyze](#) function.

As an example, the `__BUILTIN_DAED_DBETWEEN` library function is generated when the user specifies a range for the input data.

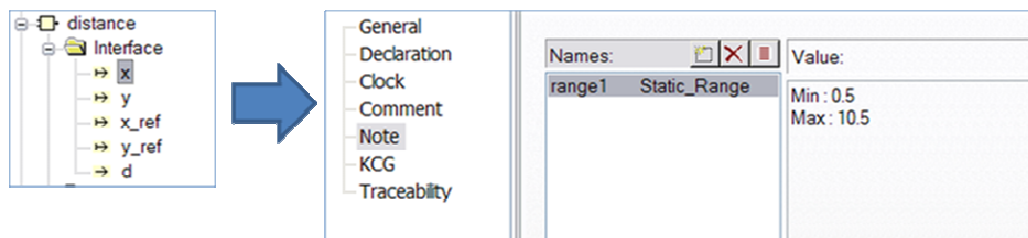


Figure 1: Generating an input interval in SCADE

The builtin library function `__BUILTIN_DAED_DOUBLE_WITH_ERROR` is generated with a specific SCADE operator (see Figure 2).

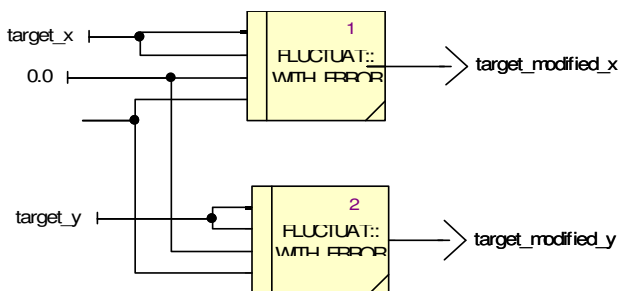


Figure 2: SCADE operator generating a numerical error

The function [setResourceFile](#) enable to set the analysis parameters. The parameters are provided thanks to a resource file generated by SCADE™ from the graphical form “Settings”.

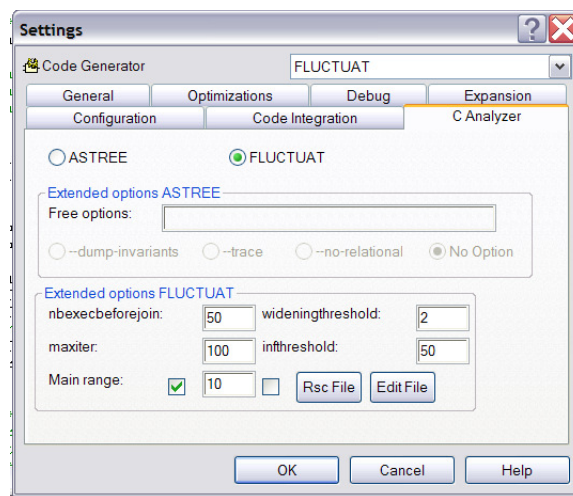


Figure 3: SCADE graphical form defining the analysis parameters

For the needs of the static analysis, KCG generates automatically a main function that calls the model. The model is then called in a bound or unbound loop with exit possibilities at every cycle. This main function is the `entryname` analysis parameter.

The entries of the HTML numerical analysis report can be specified thanks to the function [setInterestingVariable](#) of the API. By default, only the global variables and the variables of the main function are reported in the report. The function [setInterestingVariable](#) is fully applicable since SCADE is a dataflow language that KCG translates into C source files with a “Static Single Assignment” syntax.

The function [setOutputDirectory](#) of the API enables to define the place of the evolution graphs generated by the analysis. The graphical interface has to know this place to call the executable `evol_viewer` with the right arguments. The evolution files are produced when the analysis interprets the operators `PRINT` present in the SCADE model. KCG translates such operators into calls to the library function `__DAED_DOUBLE_PRINT` (see section 1.4). When the static analysis interprets such library calls, it updates the corresponding evolution files.

## 2.2 Analysis

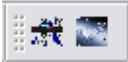
The static analysis of numerical accuracy is defined for this context in the Abstract Interpretation framework. To use it, SCADE™ calls the [analyze](#) function of the API. The analysis then covers the whole loaded SCADE model from the specified entry point. At the end the analysis results are available to build the HTML report and to display the evolution graphs.

The time of an analysis can vary from several seconds to few hours, depending on the size, on the complexity of the model and on the analysis parameters.

In the future, the function [analyze](#) is likely to accept some “call-back” functions as parameters, to react to events issued from the analysis. This point is interesting for long analyses and for analyses that fail in bounding the domains or the numerical errors. In these cases, the end-user

often needs additional information during the analysis. In some cases, he even wants to locally control the analysis to obtain more precise results more quickly.

### 2.3 Getting results

As soon as the analysis is complete, the function [retrieveVariableAndResult](#) provides the adequate data to build the analysis report. This report is available in SCADE after a click on . This report (see figure Figure 4) contains the following information for each signal/variable flagged by the user, at the time specified by the user and on the hypotheses specified by the user (see section 1.3):

- an interval containing the set of all possible values that the variable can take at the specified time
- an interval containing the set of all possible numerical errors (rounding errors), errors defined as the difference between the implementation of the computations and the ideal computations. This difference is considered as a perturbation/a noise for the variable at the specified time
- the line and the source file containing the declaration of the variable
- the line and the source file containing the unique assignment of the variable (in the simulation cycle). The uniqueness comes from the dataflow properties of the SCADE language. The line identifies a unique operator/computation in the SCADE model.
- the identifier supplied by the call of the function [setLookAtVariable](#) or [setInterestingVariable](#) to specify the variable and the time for which the values are collected.

#### The evaluation of the variable : outC.\_L41

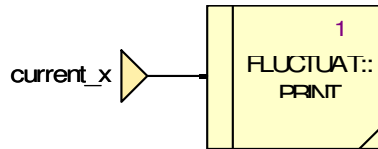
```
outC._L41 at : mainfunction\_static.c :: 74  
With domain in [9.638508e+000, 9.657804e+000]  
With error [-9.648156e-003, 9.648156e-003]  
Absolute precision 9.648156e-003
```

Figure 4: Numerical properties in the HTML report

Note that the name returned by the function [retrieveVariableAndResult](#) is the same as the name of the corresponding variable/signal of the SCADE model (KCG translation).

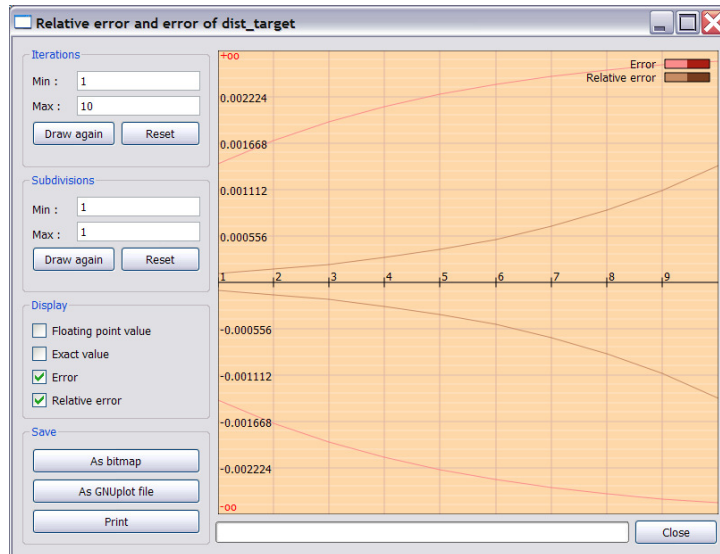
The reference of the C source code corresponds to the fields `variable->lineLastAssign` and `variable->fileLastAssign` = time (line number and file name) when the variable `variable->name` has been assigned in the current simulation cycle. These fields have sense only when the preprocessed source file comes from a dataflow language that uses Static Single Assignment that is the case of SCADE. These fields identify the unique operator/computation of the SCADE model that has a direct impact on `variable->name`.

The user can display the evolution graphs attached to a variable in SCADE thanks to the executable [evol viewer](#). The evolution graph is generated if, in the SCADE model, the variable/signal `variable_name` is connected to a PRINT operator (see Figure 5).



**Figure 5: SCAD operator generating an evolution graph**

From this connection, KCG generates a library call to the function `__DAED_DOUBLE_PRINT` in the preprocessed C source files (see section 1.4). During the static analysis and at each simulation cycle, *analyze* persists the numerical results for variable\_name. The resulting sequences of intervals are displayable with *evol viewer*.



**Figure 6: Example of evolution graphs**

### 3 Références

[1] FLUCTUAT User's Manual

[2] SCADE<sup>TM</sup> User's Manual

[3] IEEE 754 Standard for Floating-point Arithmetic. Technical report, 2004.

[4] G. Hanrot, V. Lefevre, Rouillier F., and P. Zimmermann. The MPFR library. Institut de Recherche en Informatique et Automatique, 2001.