SWEET manual

Table of Contents

1. Introduction

Compiled by Jan Gustafsson, Mälardalen University, Sweden

Box 883, 721 23 Västerås Sweden

Version: $Revision: 8028 $ $Date: 2019-10-15 14:45:47 +0200 (ti, 15 okt 2019) $.

SWEET

SWEET is a research prototype tool for flow analysis and BCET and WCET calculation for time-critical real-time programs. BCET/WCET stands for Best and Worst Case Execution Time Analysis, respectively. SWEET has been developed by the WCET research team in Västerås, Sweden since 2001 and is available upon request.

The name "SWEET" stands for SWEdish Execution Time analysis tool.

SWEET analyses programs in ALF format. ALF stands for Artist Flow Analysis language; it is a general intermediate program language format, especially developed for flow analysis. ALF code can be generated from different sources, like C code and assembler code, and a number of translators are available.

The main function of SWEET is flow analysis. The result of flow analysis is flow facts, i.e., information about loop bounds and infeasible paths in the program. Flow facts are necessary for finding a safe and tight WCET for the analysed program. Assuming a timing models exists, SWEET can also produce BCET/WCET estimates directly, avoiding the need for calculating flow facts. SWEET also performs other analyses (as described in this manual).

SWEET analyses programs for single core processors; there is no support for multi-core processors.

This document is a user manual and describes the basic functions and uses of SWEET. Running examples are provided. The manual assumes some basic knowledge about WCET analysis. The section Scientific Articles contains references to scientific articles which describe WCET theory and the analysis techniques used in SWEET which can be used as introduction.

The main functions of SWEET are described in Section Main Features of SWEET.

Contact

If you want to contact SWEET developers, use the mailing list <wcet@list.mdh.se> to the WCET project/research group at Mälardalen University. We have also created a mailing list for SWEET <SWEET-USERS@LIST.MDH.SE> to be used for SWEET users to discuss issues with SWEET.

We have quite a few users now, so we need a forum. The idea is to create an open discussion list where SWEET users can exchange experiences etc. Also, we who work with the maintenance and development of SWEET are also members of the list to learn and possibly react to the discussions. Jan Gustafsson is currently the owner of the list and will act as a moderator.

To subscribe, send an email to <listserv@list.mdh.se> with empty topic and the following contents:

subscribe SWEET-USERS

To unsubscribe send:

unsubscribe SWEET-USERS

SWEET manual history

- 2012: first version
- 2012 - 2013: a number of additions and bug fixes
- 2013-03-14:

  Added a section on AlfBackend installation.

  All ALF files are available for the examples.

  The scripts c_to_alf and c_to_alf_map are updated.

  More detailed description of manual annotations.

  A number of smaller additions and bug fixes.
- 2013-03-22:

  Explanation on why to avoid merging when calculating infeasible paths.

  A number of smaller additions and bug fixes.
- 2013-03-23:

  Sections "Support for C" and "Limitations of SWEET" added.
- 2013-04-11:

  Section "Running AlfBackend" added.

  A number of smaller additions and bug fixes.

  Distribution license text added.
- 2013-05-02:

  New debug command "Show" added.

  A number of smaller additions and bug fixes.
- 2013-05-08:

  New parameters to DOT print.
- 2013-05-30:

  AlfBackend is updated. Update your local installation! All ALF files in the manual are re-generated.

  A number of smaller additions and bug fixes.
- 2013-11-20:

  New types of input annotations, using dereferencing, are added.

  Corrections in the description of input annotations.

  A new version of AlfBackend is available.

  Info about SWEET user's forum.

  Version check is removed.

  Offsets for ALF labels are not used.

  The files std_hll.alf and libc.alf may differ between computers with different data layout (e.g., pointer size). An explanation is added how to handle this issue.

- 2015-12-17

  A new domain, CLP, is added.

  A script for optimal merging is added.

  A number of smaller additions and bug fixes.

- 2016-06-27

  The installation instructions for AlfBackend now point to the GitHub version of AlfBackend, which is based upon LLVM 3.4.

  The script `c_to_alf_map` is updated to give the flag `-gcolumn-info` to **clang**.

  Various minor corrections have been made throughout the manual.

- 2018-01-05

  The URL to the SWEET Subversion repository has been updated.

- 2018-04-06

  A new parameter, `inv`, to `--slice` has been added.

  The polyhedral domains have been removed.

1.1. SWEET and Supporting Tools

SWEET is used together with a number of supporting tools.

The **ALF tools** handles translation from the actual programming language to the ALF format. There are a number of available translators:

- C to ALF translators:
  - **AlfBackend**, a C to ALF translator based on LLVM. Developed by Benedikt Huber, TU Vienna, Austria.
  - **melmac**, a C to ALF translator based om SATiRE. Developed by Gergö Barany, TU Vienna, Austria, for the ALL-TIMES project. http://www.complang.tuwien.ac.at/gergo/melmac/ . Outdated and replaced by the AlfBackend translator.
- Assembler to ALF translators:
  - **CRL2ALF***, a PowerPC assembler to ALF translator, based on CRL2. Developed by Jens Björnhager, MSc student, Västerås, Sweden.
- **SWEET** can translate NIC code to ALF format. NIC (New Intermediate Code) is a format that was used by earlier versions of SWEET. NIC is replaced by ALF.

- (more translators are under development and will be added to the list).

SWEET, the flow (high level) analyzer, takes ALF files as input and produces flow fact files, to be used by a low level analyzer. SWEET can produce flow facts in three formats: for low-sweet, for aiT and for Rapita.

The **low-level and calculation** tools takes a flow fact file and a timing model as input, and produces BCET/WCET estimates. The **'low-sweet'** tool, developed at Uppsala University and Mälardalen University, Sweden, is a research prototype that can be used in combo with SWEET. Currently, low-sweet includes timing models for the NEC850 and ARM9 processors. The commercial tools **aiT** (http://www.absint.com/ait/) and **RapiTime** (http://www.rapitasystems.com/rapitime) can also take flow facts files produced by SWEET as input.

AlfBackend and SWEET are accessible free of charge. See the sections "Install AlfBackend" and "Install SWEET". The tool marked with * is available upon request, by contacting the SWEET developers.

1.2. SWEET contributors

SWEET has been developed since 2001. The following persons have been engaged in the development of SWEET in one way or another (our apologies if we have missed someone):

Andreas Ermedahl, Jan Gustafsson, Christer Sandberg, Linus Källberg, Martin Skogevall, Björn Lisper, Stefan Bygde, Linus Sjöberg, and Nerina Bermudo.

1.3. Running SWEET

If you have built SWEET in a command line environment (see section "Command line environment" below), it is run as a UNIX/Linux-like command (sweet) in a command line environment in PC/Windows (using e.g., Cygwin, MinGW), UNIX systems, and in Linux or Mac (using Terminal). It uses input files, and creates logs and result files. All files are of text type. The SWEET executable is found in

```
(my_path)/build/cmakedir/src/
```

when the installation instructions found in the section "Command line environment" are followed.

If SWEET is built using a GUI environment (see section "GUI environments" below) it can be run inside your build system of choice, like Visual Studio (Windows) or Xcode (Mac).

Another way to run SWEET is to use the SWEET web page [http://www.mrtc.mdh.se/projects/wcet/sweet/online/content/index.php]. The web page offers a convenient way of testing SWEET, without the need for installation. This web service was developed during an MSc project at Mälardalen University.

1.4. Installing SWEET

The following describes the steps needed to install SWEET in a Windows, Mac OS X or Linux environment.

The SWEET source code is distributed under the following license:

Note: The SWEET contributors are listed under section SWEET contributors.

1.4.1. Initial preparations

To compile SWEET, the following software packages are required:

- CMake (http://www.cmake.org). The compilation process is based on the program CMake, which can generate build files for a variety of platforms and build environments.
- GNU Multiple Precision Arithmetic library (GMP) (http://gmplib.org).

Below follows instructions on how to install these packages on the most common platforms.

1.4.1.1. Windows + Visual Studio

CMake can be downloaded from https://cmake.org. For GMP we recommend the Windows port, called MPIR, which is available from http://www.mpir.org.

1.4.1.2. Windows + Cygwin

Download and run the Cygwin setup file from http://www.cygwin.org. Include the following packages in the installation:

- cmake
- libgmp-devel (gmp, gmpxx, libgmp, libgmpxx)
- make
- gcc and g++ 4.xx

Uninstall gcc and g++ 3.xx or make sure that the 4.xx version is invoked. In principle, SWEET can be compiled using 3.xx, but it can take a very long time.

Note that users have reported problems when installing SWEET in a 64-bit version of Cygwin. The 32-bit version is preferred.

1.4.1.3. GNU/Linux

All of the above software packages are included in most GNU/Linux distributions, so they are easiest installed using the standard package management tools. For example, on Debian-based distributions, use:

```
sudo apt-get install cmake libgmp-dev
```

1.4.1.4. Mac OSX prior to Lion

CMake can be downloaded from https://cmake.org.

Install gcc 42 using MacPorts (this also installs GMP) with:

```
sudo port install gcc42
```

1.4.1.5. Mac OSX Lion (or later)

CMake can be downloaded from https://cmake.org.

Download GMP from http://gmplib.org. Install it with the following command:

```
./configure --enable-cxx --disable-static && make && make check &&
sudo make install
```

1.4.2. Building SWEET

Basically, there are two ways of building SWEET; either using a command line environment or GUI (Graphical User Interface) environments. ***Note***: in Windows, do not mix Cygwin and GUI environments; if you choose Cygwin, use it for all installations.

In what follows, it is assumed that the SWEET source code is located in the directory `sweet/`.

1.4.2.1. Command line environment

If you are using a command line environment, e.g., UNIX/Linux, Terminal (Mac OS X), or Cygwin (Windows), you can proceed using the following steps:

- Create an empty directory to build SWEET in, for example under `sweet/build/`:

  ```
  cd sweet/build
  
  mkdir cmakedir
  ```

- Go to the new directory and run CMake to generate the build system. CMake needs the path to the SWEET root directory (where the file CMakeLists.txt is located) given as an argument (in this example, the path is `../..`). CMake also needs a `-G` argument to specify the build system type, so the commands may look like:

  ```
  cd cmakedir
  
  cmake ../.. -G "Unix Makefiles"
  ```

  If CMake complains that it cannot find the paths for the GMP library, you must set them manually by running CMake again with one or both of the arguments

  ```
  -DGMP_INCLUDE_DIR=X -DGMP_LIBRARY=Y
  ```

  where X is the path to the GMP include directory and Y is the path to the GMP library file (the actual file, not its containing directory).

- SWEET can then be built by running "`make`" in the same directory.

  The command "`make test`" builds SWEET and runs a number of tests.

- Add SWEET to your path:

  ```
  export PATH=(my_path)/sweet/build/cmakedir/src/:${PATH}
  ```

CMake can also generate configurations for other build systems, for example Visual Studio (Windows), Xcode (Mac), and Code::Blocks; see the documentation on http://www.cmake.org/ for a complete list. This is controlled by the option `-G "X"` to CMake, where X is the desired build system. So to create e.g. a Visual Studio 2010 solution, follow the same steps as above but give `-G "Visual Studio 10"` to CMake instead of `-G "Unix Makefiles"`.

1.4.2.2. GUI environments

You can build SWEET using GUI environments as an alternative to command line environments.

Start the GUI version of the CMake application and fill in the fields in the CMake window according to the following:

- Source code: Root directory for SWEET source code, e.g., `sweet/`

- The binaries: Path to the empty directory where the build system should be, e.g., `sweet/build/cmakedir/`

- The variable window in CMake contains build variables which controls the build. These variables are described above and below.

Click ”Configure” and then choose the generator of choice, .e.g., UNIX Makefiles, Xcode or Visual Studio. If any important variables are missing, CMake will issue an error message.

When the configure step is done without errors, continue with ”Generate”. If the generate step is completed without errors, continue to build the SWEET target in your build system of choice.

1.4.2.3. Optional build settings

Parser source file generation

By default, no build rules are created by CMake to regenerate parser source files. This means that they are not regenerated even if the parser specification files are changed.

Note that all the generated parser sources are already included in the source tree, so these rules are not needed just to compile SWEET, only to be able to change the parsers. To enable these build rules, give the option

```
-DGENERATE_FLEXBISON_RULES=true
```

to CMake to create Flex/Bison rules. Of course, these build rules require that Flex and Bison are installed.

1.4.2.4. Auxiliary scripts and files

During the use of SWEET, a number of auxiliary scripts and files are used. They are mentioned and described in the manual, but for convenience they are also listed here.

- An ALF file (std_hll.alf) with standard definitions of the NULL pointer and absolute memory in ALF for C. The file contents is dependent on the pointer size of the computer (32 or 64 bits).

  std_hll.alf [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/32/std_hll.alf] for 32 bit computers.

  std_hll.alf [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/64/std_hll.alf] for 64 bit computers.

- Standard C library (libc) functions has to be handled specially, since the C code normally is not available as C code. Instead, these functions are available in ALF form in the libc.alf [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/libc.alf] file.

- The UNIX-script optimal_merge [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/optimal_merge] performs optimal merge.

- Scripts related to optimal merge:

run_sweet_merge [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/run_sweet_merge]: Analysis of a certain merge option (help script).

test_optimal_merge [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/test_optimal_merge]: Run 'optimal_merge' for the test program ”MergeExample” for all types of loop bounds (NOTE: takes appr. 10 hours to execute).

1.5. Installing AlfBackend

AlfBackend is available on GitHub: https://github.com/visq/ALF-llvm. Please follow the installation instructions on the GitHub page.

1.6. Running AlfBackend

You run AlfBackend in a command line environment. Download the script "c_to_alf" by right-clicking the following link: c_to_alf [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/c_to_alf] and run it in your terminal window:

```
source c_to_alf <program>
```

where `<program>` is the name of the C file (no extension). The script assumes that the path to the llvm bin directory is included in PATH.

There is also a script "c_to_alf_map" to be used to create `.map` files using a special command. This script can be downloaded by right-clicking the following link: c_to_alf_map [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/c_to_alf_map]. For more information about the use of `.map` files, see the Section Perform AE of a program in ALF format and calculate C code flow facts.

Note that the AlfBackend generates code using the data layout of your computer. This means that ALF code generated in an 32 bit computer will differ from ALF code generated in an 64 bit computer. A consequence is that ALF code generated in different computers cannot be linked, if the computers has different data layout. This also means that the standard files "std_hll.alf" and "libc.alf" are unique for computers with different data layout.

2. Main Features of SWEET

This section describes the most important features of SWEET.

2.1. Flow analysis

A program can be executed in several different ways, i.e. through different paths. To identify these paths is a task for flow analysis. The purpose of the flow analysis is to derive bounds on the dynamic execution behaviour of the program. This includes information on which functions get called, loop bounds (maximum number of times loops are iterated), dependencies between conditions or branches, paths that are feasible through the program, and execution frequencies of code parts. SWEET generates such flow information in the form of flow facts.

To find exact flow information is in general undecidable, since a perfect flow analysis would solve the halting problem. Thus, any flow analysis must be approximate. To ensure a safe WCET estimate, the flow information must include all feasible program paths.

The goal of SWEET's flow analysis is to calculate flow information as automatically as possible. SWEET offers powerful loop bound analysis, since upper bounds on the number of loop iterations must be known in order to derive WCET estimates. The flow analysis of SWEET can also identify infeasible paths, i.e., paths which are executable according to the control flow graph structure, but not feasible when considering the semantics of the program and possible input data values. In contrast to loop bounds, infeasible path information is not required to find a WCET estimate, but may tighten the resulting WCET estimate.

2.2. Abstract domains

The flow analysis is performed on an abstract model of the program, using an abstract domain and abstract operations. The purpose of this abstraction is to find a safe (over-)estimation of the possible values of the variables in the program, and, using these values, calculate safe flow information. SWEET offers several abstract domains:

- Intervals for integers and floating point values (default). Abstract values are expressed as [low..high] which represents the set of all values from low to high.
- An abstraction of pointers. Abstract pointers are expressed as a set of pairs (x, o), where x is a frame id and o is an offset interval. An example of a value of an abstract pointer p is {”a”, [1..3], ”c”, [0..0]}, meaning that p can point to either "a" with offset [1..3] or to "c". Given an environment E, the actual reference to memory is calculated as E(x)+o. For accesses of array elements a[i] the offset is calculated as s · i where s is the size in bytes of the elements.

  Intervals and the pointer abstraction is described, e.g., in the paper

  http://www.mrtc.mdh.se/index.php?choice=publications&id=0972
- Unfortunately, the interval domain suffers from an inability to abstract sparse value sets with reasonable accuracy. Such value sets commonly result from analyzing, e.g., array accesses, where the possible array offsets are separated by multiples of the width of the elements. To improve the accuracy of SWEET's analyses in such situations, we extended SWEET with a variant of Sen and Srikant's domain of Circular Linear Progressions (CLP), which keeps stride information together with the interval bounds. The CLP domain is described in the paper

  Linus Källberg: Circular Linear Progressions in SWEET [http://www.es.mdh.se/pdf_publications/3813.pdf], Technical Report, MDH, 2014.

2.3. Abstract execution

The main flow analysis technique used in SWEET is abstract execution (AE), which is a form of symbolic execution based on an abstract interpretation (AI) framework. Rather than using traditional fixed-point AI iteration, the AE executes the program in the abstract domain, with abstract values for the program variables and abstract versions of the operators in the language. For instance, the abstract domain can be the domain of intervals: each numeric variable will then hold an interval rather than a number, and each assignment will calculate a new interval from the current intervals held by the variables. As usual in AI, the abstract value held by a variable, at some point, represents a set containing the actual concrete values that the variable can hold at that point. An abstract state is a collection of abstract values for all variables at a point.

2.3.1. Abstract input annotations

Annotations in SWEET provide a way to assign abstract values to variables at certain program points. Annotations can for example represent all possible values for an input variable or a state variable, and thus catch all possible initial states for the program before analysis.

If SWEET analyses a program without using abstract input annotations, the analysis will be a **single path** analysis. In this case, SWEET works very much like an interpreter. When analyses a program with abstract input annotations representing sets or intervals of data, the analysis will be a **multi-path** analysis. SWEET will analyse all possible paths for this particular set or interval of data in one run.

2.3.2. Specification of merge points

When using abstract values, conditionals cannot always be decided. In these cases, AE must then execute both branches separately in two different abstract states. This means that AE may have to handle many abstract states, representing different possible execution paths, concurrently. The number of possible abstract states may grow exponentially with the length of these paths. In order to curb the growing number of paths, merging of abstract states for different paths can take place at certain program points (so called merge points). If the states are merged using the least upper bound operator on the abstract domain of states, then the result is one abstract state safely representing all possible concrete states. In this way a single-path abstract execution, representing the execution of the different paths, can continue from the merge point. Thus, if SWEET has problem finishing its flow analysis within given time limits, one or more merge points should preferably be used.

SWEET allows the user to specify the placement of merge points using the `-ae merge=<mergepoint>` parameter. The possible values to the merge parameter are described in Section The abstract execution option. In principle, using a lot of merging (more often) leads to quicker termination of the AE, but might lead to less precise flow analysis results. The mentioned section contains tips how to set the merge parameter for optimal results.

Merging is most interesting to use when SWEET analyses large programs with abstract input value annotations which contains large intervals.

2.3.3. Scope graph

Scope graphs can be used as "holders" of the flow analysis result of a program. A control graph is partitioned into scopes in a scope graph. A scope is a loop or a function which can be repeated. For example, a loop nest will be represented by a chain of scopes, where the scopes for inner loops are below scopes for outer loops. The purpose of the use of scopes is to structure the flow analysis of the program and also to structure the generated flow constraints in such a way that the execution of repeating constructs can be analyzed and constrained.

2.3.4. Flow facts

Flow information for scopes can be expressed as flow facts. Flow facts are used to constrain virtual execution counters for the nodes in CFG (control flow graph) in a scope. Each time a scope is entered from above in a scope hierarchy the counter #N is initialized to zero for every node N and is incremented at each execution of the node.

Flow facts have the format scope : context : linear constraint. Here context can have two possibilities. It can be a "for all" context [range] which specifies that the linear constrain should hold for all iterations of the scope and for a specific range of iterations. It can also be a "for each" context <range> which specifies that the linear constraint should hold for each individual iteration of scope. The constraint should hold for all possible iterations, if range is left out.

SWEET expresses the flow analysis (loop bounds and infeasible path constraints) results as flow facts. See Section SWEET's Flow Fact Language for details.

2.3.5. BCET/WCET calculation based on abstract execution

We have integrated time estimation in our AE method for calculating program flow constraints. This method is in principle a very detailed value analysis. As it computes intervals bounding variable values, it bounds both the BCET and the WCET. In addition, it derives the explicit execution paths through the program which correspond to the calculated BCET and WCET bounds.

The BCET/WCET calculation can be based on both basic blocks (see Section Generate BCET and WCET estimates using basic block and edges cost table) and program constructs (see Section Generate BCET and WCET estimates using ALF construct costs).

2.4. Value analysis

SWEET offers a value analysis based on classical Abstract Interpretation as presented in the paper below.

Patrick Cousot and Radia Cousot: *Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints*. In Proc. 4th ACM Symposium on Principles of Programming Languages, 1977.

The value analysis is controlled by a number of options, and the results are presented in text files and/or in DOT graphs. See Section The value analysis option for details. The analysis domain is controlled by the domain option. See Section The domain option for further information.

2.5. Generate flow hypothesis from traces

This feature is a result from the cooperation with Rapita Systems Inc. http://www.rapitasystems.com/ in the ALL-TIMES project. SWEET is able to read traces, produced by the RapiTime tool, from a file and run abstract execution according to the traces. SWEET will generate a set of flow hypothesis based on the traces. The format of these hypotheses is exactly the same as for the flow facts. However, flow hypothesis are different from flow facts since they are not guaranteed to be safe estimates on the program's possible executions. Instead, the flow hypothesis are only valid for the given traces.

2.6. Do various program analyses, like def-use analysis, reaching definitions analysis, program slicing, and generate program statistics

SWEET includes a number of program analyses which are used internally during flow analysis. These analyses are also available externally. See Section Do various program analyses, like def-use analysis, reaching definitions analysis, program slicing, and generate program statistics for examples and further information.

2.7. Support for C

C programs are translated to ALF code using AlfBackend before being analysed by SWEET. AlfBackend supports ANSI-C with the following limitations:

- Variable-length argument lists are not supported.

There is often a need for an ALF file (std_hll.alf) with standard definitions of the NULL pointer and absolute memory in ALF for C. The file contents is dependent on the pointer size of the computer (32 or 64 bits). Get this file by right-hand clicking and saving one of the following links:

std_hll.alf [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/32/std_hll.alf] for 32 bit computers.

std_hll.alf [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/64/std_hll.alf] for 64 bit computers.

Standard C library (libc) functions has to be handled specially, since the C code normally is not available as C code. Instead, these functions are available in ALF form in the `libc.alf` file. This file is developed by and available from the SWEET developers (see below). The file should be linked to the analysed program. The file contains ALF versions of the standard C functions, based on open C implementations of libc like http://git.uclibc.org/uClibc. The functions try to return a correct result, but if that is not possible, they will return TOP (any value).

Note 1: the `libc.alf` file is currently not complete. If you find that some standard C function is missing, please contact the SWEET developers so we can provide an updated version. Some libc functions, like `printf` and `malloc`, will not be supported. See next section.

Note 2: The time behaviour for the standard library functions in `libc.alf` are not guaranteed to be correct. This is because the code can differ between the ALF code and the code in the target computer. To get this right would require to generate ALF code from the binary code for the standard library functions. This may pose a problem, since we don't have support for so many processors (see section SWEET and Supporting Tools). However, the time behaviour problem is reduced by the fact that most standard library functions are short.

Note 3: The libc.alf contents is dependent on the data layout, e.g., pointer size of the computer (32 or 64 bits). Therefore, the file has to be generated in the computer where it is to be used. Download the libc.c file here: `libc.c` [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/libc.c] and generate the libc.alf file by running this script [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/libc_script]. (There is a possibility to "cross-compile" to other computers using AlfBackend, but this feature is not documented in this manual. Contact the SWEET developers if you are interested in using this option).

Note 4: There is a problem if the analysed C program contains a variable with the same name as a standard C function, if libc.alf is linked to the program during analysis. Such variable names should be avoided, but they sometimes occur (for example in statemate.c in the Mälardalen benchmarks). The problem can be solved by removing the function from libc.c and then re-generated using the mentioned script.

2.8. Limitations of SWEET

There are a number of limitations of SWEET which the user should be aware of. The main reason for not including the following features is that they are normally not used in time-critical real-time systems.

- Recursion is not supported.
- Dynamic allocation of memory (`malloc` etc. in C) is not supported.
- Some libc functions, like `printf`, are not supported.

The whole program, including called functions, has to be available as ALF code to get the best results. If some functions are missing, "do not check for unresolved external references" `-c extref=off` and "process undefined functions" `-ae pu` must be used (see sections The check statically option and The abstract execution option).

We currently assume IEEE 754-2008 floats, i.e. 32-bits floats are assumed to have 8 bits exp and 23 bits frac, while 64-bits floats are assumed to have 11 bits exp and 52 bits frac. If the target computer uses the same standard, the setting

```
-do floats=est
```

will give correct result for floating point calculations.

If the target computer uses another floating point standard, it is wise to use the setting

```
-do floats=top
```

when analysing the code. Doing so, the floating point calculations will always be safe (yield TOP_FLOAT), but will in a sense be meaningless. If floating point values are used in conditions however, this may lead to infinite loops during analysis. Using the setting

```
-do floats=est
```

will remedy this situation and yield approximately correct results, but SWEET results, like flow facts, may be wrong, and possibly unsafe. See section The Domain Option for more details.

3. Scientific Articles

This section presents some of the most important scientific papers regarding SWEET. It contains references to scientific articles which describe WCET theory and the analysis techniques used in

SWEET. These papers are mentioned and referred to in this manual. There are many more papers published by the WCET group. The WCET papers are accessed through the link

http://www.mrtc.mdh.se/index.php?choice=publications&year=any&project=0017.

- *SWEET – a Tool for WCET Flow Analysis*. Björn Lisper. 6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA 2014).

  An important part of Worst-Case Execution Time (WCET) analysis is to find constraints on the possible program flows. Upper bounds on the number of loop iterations are necessary to bound the execution time at all, and other kinds of constraints can help increasing the precision of the WCET analysis significantly by ruling out certain paths. Thus, methods and tools to find such constraints are sorely needed. Without such tools, such constraints must be provided manually by the user: this can be tedious and error-prone, and the soundness of the WCET analysis may be compromised if unsound program flow constraints are specified. SWEET (SWEdish Execution Time tool) is a tool that derives such program flow constraints, encoded as arithmetic constraints on execution counters (so-called "Flow Facts"), automatically. SWEET can compute a variety of Flow Facts, from simple loop iteration bounds to complex infeasible path constraints. It can analyze a variety of code formats through translation into an intermediate format.

  http://www.es.mdh.se/publications/3693-SWEET_____a_Tool_for_WCET_Flow_Analysis

No nosso trabalho não estamos preocupados com os Flow Facts propriamente ditos, e sim com o uso de search e abstract execution / Interpretation Para encontrar o caminho explicito do WCET!!!

Tem muito poder nossa ferramenta.

- *Deriving WCET Bounds by Abstract Execution*. Andreas Ermedahl, Jan Gustafsson, Björn Lisper, Proc. 11th International Workshop on Worst-Case Execution Time (WCET) Analysis (WCET 2011), Austrian Computer Society (OCG), Porto, Portugal, Editor(s): Chris Healy, July, 2011.

  We have integrated the time estimation in our Abstract Execution method for calculating program flow constraints. This method is in principle a very detailed value analysis. As it computes intervals bounding variable values, it bounds both the BCET and the WCET. In addition, it derives the explicit execution paths through the program which correspond to the calculated BCET and WCET bounds.

  http://www.mrtc.mdh.se/index.php?choice=publications&id=2594

- *The Mälardalen WCET Benchmarks - Past, Present and Future*. Jan Gustafsson, Adam Betts, Andreas Ermedahl, Björn Lisper. Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis, July, 2010. ✓ ✓

  Leituras sobre os benchmarks.

  Modelling of real-time systems requires accurate and tight estimates of the Worst-Case Execution Time (WCET) of each task scheduled to run. In the past two decades, two main paradigms have emerged within the field of WCET analysis: static analysis and hybrid measurement-based analysis. These techniques have been successfully implemented in prototype and commercial tool sets. Yet, comparison among the WCET estimates derived by such tools remains somewhat elusive as it requires a common set of benchmarks which serve a multitude of needs. The Mälardalen WCET research group maintains a large number of WCET benchmark programs for this purpose. This paper describes properties of the existing benchmarks, including their relative strengths and weaknesses.

  http://www.mrtc.mdh.se/index.php?choice=publications&id=2284

- *Towards a Flow Analysis for Embedded System C Programs*. Jan Gustafsson, Andreas Ermedahl, Björn Lisper. The 10th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS'05), Sedona, USA.

  Reliable program Worst-Case Execution Time (WCET) estimates are a key component when designing and verifying real-time systems. One way to derive such estimates is by static WCET analysis methods, relying on mathematical models of the software and hardware involved. This paper describes an approach to static flow analysis for deriving information on the possible execution paths of C programs. This includes upper bounds for loops, execution dependencies between different code parts and safe determination of possible pointer values. The method builds upon abstract interpretation, a classical program analysis technique, which is adopted to calculate flow information and to handle the specific properties of the C programming language.

  Is it that simple?

http://www.mrtc.mdh.se/index.php?choice=publications&id=0972

- *Fully Bounded Polyhedral Analysis of Integers with Wrapping*. Stefan Bygde, Björn Lisper, Niklas Holsti (Tidorum LTD) International Workshop on Numerical and Symbolic Abstract Domains, 2011, Venice, Italy.

  Analysis of convex polyhedra using abstract interpretation is a common and powerful program analysis technique to discover linear relationships among variables in a program. However, the classical way of performing polyhedral analysis does not model the fact that values typically are stored as fixed-size binary strings and usually have a wrap-around semantics in the case of overflows. In embedded systems where 16-bit or even 8-bit processors are used, wrapping behaviour may even be used intentionally. Thus, to accurately and correctly analyse such systems, the wrapping has to be modelled. We present an approach to polyhedral analysis which derives polyhedra that are bounded in all dimensions and thus provides polyhedra that contain a finite number of integer points. Our approach uses a previously suggested wrapping technique for polyhedra but combines it in a novel way with limited widening, a suitable placement of widening points and restrictions on unbounded variables. We show how our method has the potential to significantly increase the precision compared to the previously suggested wrapping method.

  http://www.mrtc.mdh.se/index.php?choice=publications&id=2595

  (The presented analysis was partially implemented in an earlier version of SWEET.)

4. Examples of Using SWEET (Use Cases)

This section contains examples of uses of SWEET. Each type of use is called a use case. The idea with these examples is to show typical command lines that invokes useful functions in SWEET.

The examples assume that there is a script "c_to_alf" that invokes the AlfBackend, and of course, that AlfBackend is installed. This script can be downloaded by right-clicking the following link: c_to_alf [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/c_to_alf].

The notion <x> denotes a selectable value.

4.1. Show the version of SWEET.

The following command will show the date the actual SWEET binary was built:

```
sweet --version
```

will give the following console text:

```
Executing command 'version'.
```

```
This executable was built 20xx-xx-xx from SVN revision yyyy (20zz-
zz-zz)
```

If, for some reason, the SVN revision and date could not be identified during the compilation of SWEET, "?" will be display.

4.2. Show help texts.

On-line help is available for SWEET. The command:

```
sweet --help
```

writes help texts on the console. There are also specific help texts on various topics:

```
--help topic=annot
```

shows help on SWEET's input annotation language. This help text is also available (in formatted and extended form) in the Section Abstract Input Annotations.

```
--help topic=ffg
```

shows help on flow fact generation using AE. Also gives an introduction to SWEET's context-sensitive valid-at-entry-of flow fact format. This help text is also available (in formatted and extended form) in the Section SWEET's Flow Fact Language.

```
--help topic=trace
```

shows help on flow hypotheses generation using traces. This help text is also available (in formatted form) in the Section Flow Hypotheses Generation Using Traces.

4.3. Perform static check of ALF code.

Assume that we have a C file example_1.c [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/example_1.c]. We now want to translate the C file to ALF and then do a static check to ensure that the ALF file is syntactically correct. This requires the following steps:

1. Translate the file to ALF format:

   ```
   source c_to_alf example_1
   ```

   or, use this ALF file example_1.alf [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/example_1.alf].

2. Perform static check

   ```
   sweet --input-files=example_1.alf --check-statically
   ```

the output from SWEET

```
SWEET version 20xx-xx-xx

Executing command 'input-files'.

Parsing ALF file 'example_1.alf'.

Executing command 'check-statically'.
```

shows that the ALF file is syntactically correct since no errors were reported. Note that the static check always is invoked when another option is run, i.e., you don't have to give the --check-statically option when an analysis is made.

4.4. Generate DOT graphs for a program.

Assume that we have a C program consisting of many files.

1. Translate the files to ALF format:

   ```
   source c_to_alf file_1

   source c_to_alf file_2
   ```

   ...

   ```
   source c_to_alf file_n
   ```

2. Generate DOT graphs for the program:

```
sweet --input-files=file1.alf,file2.alf,...,filen.alf --dot-print
file=<file-name> g=cfg,cg,rsg,fsg,sgh
```

where `file=<file-name>` specifies the base name of the file where to print the graph(s). A string indicating the format of the file will be appended + a '.dot' suffix. For example, if file=pgm, the name of the control flow graph will be pgm.cfg.dot. If not given, the program file base name will be used. For example, if the (single) input file name is `pgm.alf`, the name of the control flow graph will be `pgm.cfg.dot`.

The graph types that are generated are:

cfg - Control flow graph.

cg - Call graph.

rsg - Reduced scope graph where the cfg nodes are basic blocks.

fsg - Full scope graph where the cfg nodes are statements.

sgh - Scope graph hierarchy where no cfg nodes are shown.

The DOT files can be viewed with a DOT viewer, for example Graphviz, which can be downloaded from www.graphviz.org [http://www.graphviz.org].

The graphs are colour-coded, so it is easy to find, e.g., headers and back-edges. The coding scheme is:

- Blue node = header
- Red edge = backedge
- Green edge = exit/entry edge
- Blue edge = recursive call edge

As an example, let us generate a graph for the C program example_1.c [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/example_1.c].

1. Translate the C file to ALF format:

   ```
   source c_to_alf example_1
   ```

   or, use this ALF file example_1.alf [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/example_1.alf].

2. Generate all graph types for the program:

   ```
   sweet --input-files=example_1.alf --dot-print g=cfg,cg,rsg,fsg,sgh
   ```

The resulting graphs are accessible using the following links: example_1.rsg.dot [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/example_1.rsg.pdf].

4.5. Link ALF files.

SWEET requires that all code is available when doing its analyses. If some functions are missing, the "process undefined functions" feature `pu` must be used (see section The abstract execution option). To handle the case when a the program code consists of many files, SWEET has an ALF linking facility.

As a first example, we look at the case when we have an application completely written in C. To be able to analyse the program, all (called) parts of the program must be available as ALF code. This ALF code can be created from the C source code using AlfBackend. If standard C library functions (libc) are used, there is an ALF file available (file libc.alf). There is also a standard ALF file (std_hll.alf) with standard definitions of the NULL pointer and absolute memory in ALF for C. Get these files (libc.alf and std_hll.alf) by right-hand clicking and saving the following links: libc.alf [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/libc.alf] and std_hll.alf [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/std_hll.alf].

SWEET offers the possibility to generate one ALF file from the linked files. This is convenient since the translation to ALF and the ALF linking can be eliminated in subsequent processing. The linker checks that all necessary code is included.

1. Translate the files to ALF format:

   ```
   source c_to_alf file_1
   ```

   ```
   source c_to_alf file_2
   ```

   ...

   ```
   source c_to_alf file_n
   ```

2. Link the ALF files and generate one ALF file for the program:

   ```
   sweet --input-files=file1.alf,file2.alf,...,filen.alf --function-information=x p=alf
   ```

where x = start function for the generated ALF code. This code includes all functions reachable from the given function (default = main). The name of the generated ALF file will be x.alf.

The command in 2. can also be used when we have an application consisting of several ALF files which are generated from, e.g., assembler code.

Let us look at an example with a C program that calls a libc function. We use the st.c [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/st.c] program from Mälardalen benchmarks, which is a collection of 30+ useful WCET benchmarks, written in C (see http://www.mrtc.mdh.se/index.php?choice=publications&id=2284).

1. Translate the files to ALF format:

   ```
   source c_to_alf st.c
   ```

   or, use this ALF file st.alf [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/st.alf].

2. Link the ALF files and generate one ALF file for the program:

   ```
   sweet --input-files=st.alf,libc.alf,std_hll.alf func=main --function-information=main p=alf
   ```

The func=main parameter was given since this program code contains more than one starting point for an analysis.

4.6. Perform AE in single path mode and calculate flow facts

Abstract execution is a method based on abstract abstraction, but where all loop iterations are executed separately instead of always merging states at program join point after loop iterations. The abstract execution can be performed in single path mode (one execution) or multi path mode (all possible executions).

Abstract execution is the most developed analysis in SWEET, and there are many parameters to this option (see Section The abstract execution option).

As a first example, let us analyse a C program in single path mode. Let us again use the C file example_1.c [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/example_1.c].

1. Translate the C file to ALF format:

   ```
   source c_to_alf example_1
   ```

   or, use this ALF file example_1.alf [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/example_1.alf].

2. Perform abstract execution and calculate flow facts:

   ```
   sweet --input-files=example_1.alf,std_hll.alf --dot-print g=rsg --
   abstract-execution ffg=unsp ft=n --flow-facts lang=ais,ff
   ```

The parameter ffg defines the flow fact generators (ffgs) to use. In this example, we use unsp (upper bounds for nodes in program context) since we are interested in the loop bound on the inner triangular loop. For more information on selection of ffgs, see Section The flow facts option.

The parameter ft=n means: do not continue with the fall-through state after switch statements (new ALF standard since late 2011). See Section The abstract execution option [aeo] for more information.

The parameter lang defines the format of the produced flow facts. Here we use ff (SWEET's default flow fact format), and ais (the AIS format). For more information on the ff format, see Section Flow Fact Language.

The results of the analysis are flow facts in two formats, and a scope graph for reference purpose. Let us have a look at the generated files.

SWEET's default flow fact format ff [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/example_1.ff] contains the upper bound of the loops. Of special interest is the bound for the inner loop shown on line 5:

```
: main : [] : main::bb5 <= 5050 ;
```

Please note that the generated flow facts refer to entities in the ALF file. The node "main::bb5" can be seen in the scope graph [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/example_1.rsg.pdf] for the program (where the label is main__bb5) and it is a node in the body of the inner loop (called "main_L1_L1"). Thus, the flow fact shows that the body of the inner loop iterates at most 5050 times, not 100 * 100 = 10000 which could be the first guess. This is because of the dependency of the inner loop on the outer.

The file in **AIS format** ais [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/example_1.ais] shows the same information in its own format on line 8:

```
FLOW SUM ("main::bb5") <= 5050 ;
```

Again, the generated flow facts refer to entities in the ALF file. With ALF files generated using a special option to AlfBackend, a map file could be generated that allowed the AIS and Rapita formats to refer to C entities; see Section Perform AE of a program in ALF format and calculate C code flow facts.

4.7. Perform AE in multipath mode and calculate flow facts

As the next example, let us analyse a C program in multipath mode. Let us use the C file example_2.c [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/example_2.c]. The input annotation file example_2.ann [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/example_2.ann] sets the variable "i" to the interval [0..10] at the entry to the function foo. Note that the ALF variable name "%i" has to be used.

1. Translate the C file to ALF format:

   ```
   source c_to_alf example_2
   ```

   or, use this ALF file example_2.alf [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/example_2.alf].

2. Perform abstract execution and calculate flow facts:

   ```
   sweet --input-files=example_2.alf,std_hll.alf annot=example_2.ann
   --dot-print g=rsg --abstract-execution ffg=lhss,uhss,unps ft=n --
   flow-facts lang=ais,rapita,ff
   ```

The flow fact generators (ffgs) we use in this example are lhss (lower header node bounds in scope context), uhss, (upper header node bounds in scope context), and unps (upper bounds for node pairs in scope-context) since we are interested in the loop bounds and an infeasible path involving two nodes. The parameter lang defines the format of the produced flow facts. Here we use ff (SWEET's default flow fact format), rapita (the RAPITA format), and ais (the AIS format). The results of the analysis are flow facts in three formats, and a scope graph for reference purpose. Let us have a look at the generated files.

SWEET's default flow fact format example_2.ff [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/example_2.ff] contains:

- lower and upper bounds of the loop "main_foo1_L1" (see the scope graph [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/example_2.rsg.pdf] where the loop and its loop header are visible):

  ```
  : (foo, foo::bb2) : [] : foo::bb2 >= 91 ;
  ```

  ```
  : (foo, foo::bb2) : [] : foo::bb2 <= 101 ;
  ```

  We see that we get the loop bound as an interval [91..101], since the loop bound depends on the variable i, which is an interval. Note that when SWEET gives loop bounds on headers (in this case foo::bb2), it will be one more iteration than the body for a loop of this type (a for-loop). The first part of the flow fact (foo, foo::bb2) means that the flow fact are valid for the loop in the scope foo with the header foo::bb2. As earlier, the generated flow facts refer to entities in the ALF file.

- information on an infeasible path:

  ```
  : (foo, foo::bb2) : [] : foo::bb5 + foo::bb11 <= 101 ;
  ```

  which means that the sum of the number of executions of foo::bb5 and foo::bb11 are exactly equal to the loop bound, i.e., the are mutually exclusive. This is due to the exclusive conditions if (x > 5) and if (x < 0) in the program. There are also a large number of other flow facts of unps-type in the file.

The file in **AIS format** example_2.ais [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/example_1.ais] shows the lower and upper bounds of the loop "main_foo1_L1" on line 4:

```
LOOP "foo::bb2" MIN 90 MAX 100 BEGIN ;
```

The AIS format refers to the bounds of the loop body.

The file in **Rapita [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/example_2.rapita]** format example_2.rapita [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/example_2.rapita] shows the lower and upper bounds of the loop "main_foo1_L1" on line 4 and 5:

```
#pragma RPT [foo::bb2] loop_max_iter(100) ;
```

```
// ** pragma RPT [foo::bb2] loop_min_iter(90) ;
```

The Rapita format also refers to the bounds of the loop body. (The lower bound is commented away in the current implementation).

Note that neither the AIS nor the Rapita formats represent the infeasible paths; therefore output of this information to these formats is not implemented in SWEET. To get this information, you should use the .ff format.

Again, the generated flow facts refer to entities in the ALF file. With ALF files generated using a special option to AlfBackend, a map file could be generated that allowed the AIS and Rapita formats to refer to C entities; see Section Perform AE of a program in ALF format and calculate C code flow facts.

4.8. Perform AE in multipath mode using merging, and calculate flow facts

Sometimes, abstract execution may take a long time. This can be the case, e.g., for programs with many paths and large number of variables. One way of shorten the analysis time can be to use merging. This option will cause SWEET to merge states at certain program points during analysis, and thus curb the number of states.

Merging is described in Section Specification of merge points.

As an example, we will do abstract execution of bsort100.c [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/bsort100.c], which is a program that performs the bubble sort algorithm from the Mälardalen benchmarks. With the annotation file bsort100.ann [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/bsort100.ann], we set all values in the Array variable to TOP_INT (any integer value) at the entry to BubbleSort (thus overwriting the assigned values in the function Initialize in the program):

```
FUNC_ENTRY BubbleSort ASSIGN Array 32 32 100 TOP_INT;
```

After converting the file to ALF (see above), yielding the ALF file bsort100.alf [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/bsort100.alf], we first try the abstract execution without merging (`merge=none` is default):

```
sweet --input-files=bsort100.alf,std_hll.alf annot=bsort100.ann --
abstract-execution ffg=lhss,uhss ft=n --flow-facts lang=ff
```

and note that the analysis does not terminate within reasonable time. If we instead try with:

```
sweet --input-files=bsort100.alf,std_hll.alf annot=bsort100.ann --
abstract-execution merge=all ffg=lhss,uhss ft=n --flow-facts lang=ff
```

we get the resulting flow facts in bsort100.ff [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/bsort100.ff] within seconds.

What merging strategy to use depends on the properties of the analysed program. A general rule could be to use as little merging as possible while still getting results within time and memory limits. This is a trade-off situation. Our WCET 2011 paper Deriving wcet bounds by abstract execution [http://www.mrtc.mdh.se/index.php?choice=publications&id=2594] contains examples of this time vs. precision situation. As shown in Table 2 and 3 in the paper, sometime we lose precision by merging, sometimes we don't.

In rare cases, analysis does not terminate when merging is used. This may be the case when merging introduces overestimations which cause loops conditions to be both taken and not taken.

4.9. Perform calculation of loop bounds using AE with optimal merging

It is often the case that the optimal merging strategy is not known (see the discussion in the previous section). The UNIX-script optimal_merge, described in this section, solves this problem (the script and its help script can be downloaded at the end of this section). The scripts will perform calculations of upper or lower loop bounds for a given program for **all** combinations of merge options, and then calculate the tightest bounds from the result.

```
optimal_merge p timeout options fftype
```

Command line arguments:

The argument

```
p
```

is the base name ”p” of the program in ALF format (p.alf). An annotation file (p.ann) is assumed to be present in the folder (since merging is meaningless for a single-path program), as well as the std_hll.alf file.

The argument

```
timeout
```

defines the upper time limit for the SWEET analysis, in seconds. If SWEET does not finish with in this time, it will be aborted, and the missing flow facts will be ignored in the result. Note that the result will always be safe anyway.

The argument

```
options
```

can be used to define extra SWEET options, if wanted.

The argument

```
fftype
```

defines the flow fact type that will be calculated. Only one type is allowed for each run of the script. The type can be selected from the following lists:

For upper loop bound calculation:

uhss uhsf uhsp uhpf uhpp unss unsf unsp unsp unpf unpp uess uesf uesp ucsf ucsp ubns

For lower loop bound calculation:

lhss lhsf lhsp lhpf lhpp lnss lnsf lnsp lnps lnpf lnpp less lesf lesp lcsf lcsp lbns

For explanation of the types, please see Section The abstract execution option.

The result will be a file with the name <p>_<fftype> with the calculated loop bounds.

Example:

```
bash optimal_merge MergeExample 100 func=main uhss
```

The example is based on the C file MergeExample.c [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/MergeExample.c] and uses the files MergeExample.alf [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/MergeExample.alf] and MergeExample.ann [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/MergeExample.ann]. The result is stored in the file MergeExample_uhss [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/MergeExample_uhss]. In his file, we can see the tightest limits found from all (terminating) analyses with different merge options (uhss = upper header bounds in scope context).

Scripts that are used:

optimal_merge [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/optimal_merge]: Main script.

run_sweet_merge [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/run_sweet_merge]: Analysis of a certain merge option (help script).

Optional example script:

test_optimal_merge [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/test_optimal_merge]: Run 'optimal_merge' for the test program ”MergeExample” for all types of loop bounds (NOTE: takes appr. 10 hours to execute).

4.10. Perform AE and calculate C code flow facts

Flow facts in .ff format relate directly to the ALF format, as shown in the previous use-cases. A more readable and robust form of flow facts would be if they related to positions in the source code. These flow facts are valid even if the ALF files are re-generated (assuming that the position in the source code is not changed). This possibility of "C code flow facts" is now available. These C code flow facts can be used as input to the aiT and RapiTime tools.

The map file

C code flow facts can be created for ALF code which is generated using a special option to AlfBackend as described below. When this option is used, a "map" file is generated. Each line in this file has the following general format:

```
label_id;file;line;col
```

where `label_id` is the ALF label, `file` is the file name of the C file (including relative or absolute path), and `line` and `col` denote the placement (line, column) in the source code file of the corresponding C construct. An example line from the map file expint.map [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/expint/expint.map] for the file expint.c [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/expint/expint.c] is:

```
alf_label_26;expint.c;67;12
```

Generation of map files

AlfBackend can create .map files using a special command which is invoked using the c_to_alf_map script. This script can be downloaded by right-clicking the following link: c_to_alf_map [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/c_to_alf_map]. The expint.map file is generated using the command:

```
source c_to_alf_map expint
```

The ALF code generated is expint.alf [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/expint/expint.alf].

If the files `libc.alf` and/or `std_hll.alf` files are used, the corresponding .map files are needed by SWEET. Get these files by right-hand clicking and saving the following links: libc.map [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/libc.map] and std_hll.map [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/std_hll.map].

Generation of aiT flow facts

The map file is used when generating flow facts to connect these to C lines. For example, the command:

```
                            ?
sweet -i=expint.alf -ae pu -m rcs -d g=cfg --flow-facts lang=ais?
                                ?
```

will generate the following aiT flow facts in the file expint.ais [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/expint/expint.ais]:

```
# Loop bounds

LOOP "expint.c_67_17" MAX 100 BEGIN ;
LOOP "expint.c_75_27" MAX 49 BEGIN ;
```

which give the loop bounds for the loops that start on the line 67 and 75 in the expint.c file, respectively.

Generation of RapiTime flow facts

Similar flow facts can be generated for the RapiTime tool using the SWEET command:

```
sweet -i=expint.alf -ae pu -m rcs -d g=cfg --flow-facts lang=rapita
```

It will generate the following aiT flow facts in the file expint.rapita [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/expint/expint.rapita]:

```
// Loop body bounds

#pragma RPT [expint@L2] loop_max_iter(100) ;

#pragma RPT [expint@L3] loop_max_iter(49) ;
```

4.11. Perform value analysis based on AE

Value analysis is based on abstract interpretation. Let us as an example analyse the program bs.c [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/bs.c] with the corresponding ALF file bs.alf [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/bs.alf]. The annotation file bs.ann [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/bs.ann] assigns TOP_INT to "%x" to force the code in bs to look inside and outside all key positions. The value analysis is invoked by:

```
sweet -i=bs.alf -va d=r p=r
```

The result is printed to the text file _AlfValueAnalysis.txt [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/_AlfValueAnalysis.txt] and the DOT graph file _AlfValueAnalysis.dot [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/_AlfValueAnalysis.pdf].

4.12. Perform AE using the CLP domain

To improve the accuracy of SWEET's analysis for certain types of values, SWEET is extended with a variant of Sen and Srikant's domain "Circular Linear Progressions" (CLPs), which keeps stride information together with the interval bounds. The example test1.c [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/test1.c] shows the power of the CLP-domain in two stages.:

```
volatile int32_t top;
int32_t main() {
  int32_t arr[] = { 110, 140, 100, 130, 120 };
  int32_t i = top, ret;
  if (0 <= i && i < sizeof(arr) / sizeof(arr[0]))
   ret = arr[i]; // i \in [0..4], ret \in { 100, 110, ..., 140 }
  else
   ret = 90;
  return ret; // ret \in { 90, 100, ..., 140 }
}
```

In the true branch, the index 'i' is first multiplied with 4 to become a offset in the array (each element is 4 bytes). With the CLP domain you get stride = 4, while the interval domain also includes offsets that are not evenly dividable by 4. When the values are fetched from the array (all values are retrieved since 'i' can point to any element), the resulting set {100, 110, ..., 140} is represented exact with a CLP, but not with an interval. The second stage plays of course no role for the intervals, because they get problems already at the indexing.

The output annotations from runs with -do = clp respectively -do = int show the value of the variable 'ret' at the end of program. The result with -do = clp is as exact as it can be (merged from the two branches of the if statement):

```
STMT_ENTRY "main" "main::if.end" 0 ASSIGN "%ret.0" 0 32 INT 90 140
10 ;
```

while we only get junk with -do = int:

```
STMT_ENTRY  "main"  "main::if.end"  0  ASSIGN  "%ret.0"  0  32  INT
-2147483648 9175040 OR INT 1677721600 2147483647 ;
```

The analysis is invoked by:

```
sweet  -i=test1.alf,std_hll.alf  outannot=test1.oas  -ae  vola=t  -do
type=clp
```

The analysis uses the following files: test1.alf [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/test1.alf] and test1.oas [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/test1.oas].

4.13. Generate BCET/WCET estimates using basic block and edges cost table

SWEET has a function where BCET and WCET are estimated using abstract execution. As SWEET computes intervals bounding variable values, it also bounds both the BCET and the WCET, based on timing information for the basic blocks. In addition, it derives the explicit execution paths through the program which correspond to the calculated BCET and WCET bounds. This feature of SWEET is described in detail in the paper:

http://www.mrtc.mdh.se/index.php?choice=publications&id=2594

It must be noted, that this function currently is based on some features provided by the NIC intermediate format, an older version of SWEET and of low_sweet. These are used to generate the timing model (.tdb) necessary for the calculations. It should also be noted that even though SWEET is capable of calculating the BCET, this calculation was excluded from the paper since the low_sweet analysis currently does not derive safe lower timing bounds.

However, the necessary .tdb file could be provided from any method or tool that can measure or calculate the timing for basic blocks. Even the pipeline effects can be provided (as negative #cycles on edges).

As an example, let us use the Mälardalen benchmark program janne_complex.c [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/janne_complex.c] with its ALF file janne_complex.alf [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/janne_complex.alf]. The annotation file is janne_complex.ann [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/janne_complex.ann]. The timing file generated from NIC intermediate format, an older version of SWEET ("nic-sweet") and of low_sweet is found in janne_complex.tdb [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/janne_complex.tdb]. It was generated using the commands:

```
nic-sweet -i janne_complex.nic -ae
```

```
low_sweet --mode pipe janne_complex --cpu ARM9 --printTDB
```

```
mv janne_complex.tsdb janne_complex.tdb
```

Each line in the .tdb file is a (basic block, #cycles) pair. The timing is calculated for the ARM9 processor.

The ALF file is generated from the NIC intermediate format with the command:

```
nic-sweet -i janne_complex.nic -palf
```

The reason for generating the ALF code from the NIC code is that the ALF code and the timing file must refer to the same basic blocks.

Finally, the BCET and WCET is calculated with the command:

```
sweet -i=janne_complex.alf annot=janne_complex.ann -ae
bbc=janne_complex.tdb merge=none
```

The result on the command line looks like this:

```
BCET estimate based on BB cost lookup table: 7

WCET estimate based on BB cost lookup table: 614
```

Again, note that the BCET is not safe in this case (see above).

4.14. Generate BCET/WCET estimates using ALF construct costs

This use case is similar to the use case above, but instead of calculating time based on timing information for basic blocks, we instead use times for program constructs. This allows SWEET to generate BCET and WCET estimates on a higher level (like source code or intermediate code level) and thus do an earlier analysis of the code than what is possible in "traditional" WCET analysis. The timing for the constructs are found, e.g., by model identification. The method is described in the paper:

http://www.mrtc.mdh.se/index.php?choice=publications&id=2592

In the paper, and in the following example, we have chosen the ALF code level as the level for the timing information.

As an example, let us analyse the program insertsort.c [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/insertsort.c] with the corresponding ALF file insertsort.alf [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/insertsort.alf]. The annotations are found in insertsort.ann [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/insertsort.ann]. The timing information is stored in the file insertsort.clt [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/insertsort.clt]. Each line in the file has the format (ALF construct, #cycles).

The following command calculates the BCET and WCET estimates:

```
sweet -i=insertsort.alf,std_hll.alf annot=insertsort.ann -ae
aac=insertsort.clt tc=st,op merge=all
```

The following results are shown on the command line:

```
BCET estimate based on ALF AST cost lookup table: 500

WCET estimate based on ALF AST cost lookup table: 4145
```

4.15. Read a trace from file and run abstract execution according to the trace

what is this?

See Section Flow Hypotheses Generation Using Traces for instructions and examples.

4.16. Generate annotation templates

SWEET offers a function to generate abstract annotation templates for each imported and referenced identifier (i.e., references to undefined entities). The annotation value will be set to TOP in case the type is known, else the value is left to fill in manually. This is useful, e.g., for programs that use global variables that can be changed by other programs. This is common in embedded systems (where these programs often execute as tasks). The limits of these variables must be filled in to get as tight flow facts as possible.

Assume that we have two cooperating programs. The first program, annot_ex_loop.c [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/annot_ex_loop.c] with the corresponding ALF file annot_ex_loop.alf [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/annot_ex_loop.alf], defines a global variable, GlobalVariable, and uses the value of this variable to control the number of iterations in a loop. The second program, annot_ex_incr.c [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/annot_ex_incr.c] with the corresponding ALF file annot_ex_incr.alf [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/annot_ex_incr.alf], declares GlobalVariable (i.e., imports it), increments it and resets it to -9 when a limit of 9 is reached.

It is obvious that an analysis of the first program in isolation will not give a correct loop bound, since global variables in C are set to 0. Using SWEET to analyse the program this way is done with:

```
sweet --input-files=annot_ex_loop.alf,std_hll.alf --abstract-execution ffg=lhss,uhss ft=n --flow-facts lang=ais
```

The resulting loop bounds will be:

```
LOOP "FunctionWithLoop::bb1" MIN 13 MAX 13 END ;
```

To solve this problem, we use the function to generate abstract annotation templates. We run the following command for the second program where GlobalVariable was imported:

```
sweet -i=annot_ex_incr.alf,std_hll.alf --check-statically extref=off --annotation-templates=annot_ex_incr.ann
```

and get an annotation file annot_ex_incr.ann [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/annot_ex_incr.ann] with the following line:

```
VOLATILE ASSIGN GlobalVariable <type> <val> ;
```

This line is edited to:

```
PROG_ENTRY ASSIGN GlobalVariable INT -9 9 ;
```

(VOLATILE is not used) and the file is renamed to annot_ex.ann [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/annot_ex.ann]. Now, SWEET can calculate the correct loop bounds by:

```
sweet --input-files=annot_ex_loop.alf,std_hll.alf annot=annot_ex.ann --abstract-execution ffg=lhss,uhss ft=n --flow-facts lang=ais
```

and get the correct loop bound:

```
LOOP "FunctionWithLoop::bb1" MIN 1 MAX 22 END ;
```

For more information see Section The annotation templates option. Actually, the same functionality can be achieved by using option --function-information with parameter p=a. See Section The function information option.

4.17. Do various program analyses, like def-use analysis, reaching definitions analysis, program slicing, and generate program statistics

4.17.1. The def-use analysis

TBA

4.17.2. The reaching definitions analysis

TBA

4.17.3. Program slicing

Program slicing is a technique to reduce/specialize a program for a specific analysis. For example, if only the looping behaviour of a program is of interest, the variables and statements that do not affect the iterations of the loops can be removed. The remaining program is called a "slice".

SWEET offers several types of slicing, as shown in Section The program slicing option. We will show two types of slicing in this section; slicing on loop conditions and slicing on all conditions.

1. **Slicing on loop conditions**. Consider the program slicing.c [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/slicing.c]. We are interested to see which variables that affect the iteration of the loops. These are the variables that appears in loop conditions. Since the analysis is made on the ALF code, we first translate the C code to ALF, and get the file slicing.alf [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/slicing.alf]. If we look into the ALF code (line 22 - 28) we see that the variables in the program are:

   ```
   "%j.0" "%i.0" "%k.0" "%tmp4" "%tmp7" "%k.1" "%tmp10"
   ```

   Then we run the command:

   ```
   sweet -i=slicing.alf --slice ent=l p=l d
   ```

   meaning "create a program slice with only the local variables that affect the loops" (we just print the local variables p=l since there are no global variables in the program). If we look in the resulting list of variables in the file _ALFSlicing_loops_backward.txt [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/_ALFSlicing_loops_backward.txt] we get:

   ```
   "%i.0" "%tmp10"
   ```

   If we look at the slicing.c and slicing.alf files we see that the variable "i" is stored in these ALF variables, and it is also only this variable that controls the loop. We can conclude that the variables

   ```
   "%j.0" "%k.0" "%tmp4" "%tmp7" "%k.1"
   ```

   have been removed. Looking in the generated PDG [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/_ALFSlicing_loops_backward.pdf] (program dependency graph), that was generated due to the parameter d, we can see the removed PDG nodes (in gray) and the remaining PDG nodes (in black).

2. **Slicing on conditions**. If we are interested to see all variables that affect the control flow of the program slicing.c, we run the command:

   ```
   sweet -i=slicing.alf --slice ent=c p=l d
   ```

   meaning "create a program slice with only the local variables that affect conditions". The resulting list of variables in _ALFSlicing_conds_backward.txt [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/_ALFSlicing_conds_backward.txt] contains the following variables:

   ```
   %i.0 %k.0 %tmp7 %k.1 %tmp10
   ```

   These are the variables affecting all conditions, including loops. Comparing to the list of remaining variables in example 1. above, and looking at the slicing.c and slicing.alf files, we can conclude that the variable "k" is added to the list since it is controlling the if-statement at line 14 in the program.

   Looking in the generated PDG [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/_ALFSlicing_conds_backward.pdf] (program dependency graph), that was generated due to the parameter d, we can see the removed PDG nodes (in gray) and the remaining PDG nodes (in black).

4.17.4. Generate program statistics

TBA

4.17.5. Using the debug function to find infinite loops in SWEET analysis

The debug function in SWEET allows you to monitor the analysis. Especially, if SWEET "hangs" in an infinite loop in the AE, you can find the place of the loop in the ALF code (and the corresponding C code, if relevant). Proceed as follows:

1. Print the call graph: sweet –i=<file.alf> -d g=cg

2. (Optional step). Find the level in the call graph where which the loop is:

   ```
   sweet –i=<file.alf> -ae debug=inter

   sth 'enter'

   'ctrl' c

   q 'enter'
   ```

   After `'ctrl' c`, you get (for example):

   ```
   Maximum height: 2, minimum height: 2
   ```

   which means that the loop appears on call level 2 in the call graph. Main is level 1.

3. Set breakpoints for all "switch" statements in the ALF code. Both if statements and loop condition statements are always translated to "switch" statements by AlfBackend.

   ```
   sweet –i=pgm.alf -ae debug=inter

   bp switch 'enter'

   r 'enter'
   ```

   You will get (for example):

   ```
   Reached a breakpoint switch on 191:8, ("pgm::bb10",0) in state "40575"
   ```

   Here you get the line number in the ALF program. Repeat running the program (with `r 'enter'`) to find the control flow pattern. Example: 155 191 255 330 351 454 191 255 330 351 454 191 etc., points to line 191 in the ALF file as the start line of the infinite loop.

   Another way is to find that the surrounding loops hangs. Example:

   ```
   sweet –i=pgm.alf -ae debug=inter

   bp 155 'enter'

   r 'enter'
   ```

   hangs, but

   ```
   sweet –i=pgm.alf -ae debug=inter

   bp 191 'enter'

   r 'enter'
   ```

   keeps repeating line 191, which points to line 191 in the ALF file as the start line of the infinite loop.

4. (For ALF code generated from C). Find the corresponding loop in the C file.

Generate the C to ALF map file:

```
source c_to_alf_map pgm
```

Locate the loop statement in the C file:

Line 191 -> (pgm.alf) -> "pgm::bb10" -> (pgm.map) -> pgm.c line 34: `while(n > 0)`

In the same way we find that lines 255, 330, 351, and 454 are all if statements inside the inner while loop. So now we see that the infinite loop is the while loop starting at line 34 in the C code.

5. SWEET Command Options

The SWEET tool is invoked on the command line and is controlled by options and parameters with optional values. The general format for this is:

```
sweet  -option1  parameter11  parameter12=value  ...  -option2=value
parameter 21 ...
```

There is a short and a long (more explanatory) form of most options. The short form is preceded by "-", the long form by "--".

The options can appear in any order, as well as the parameters to the options. The parameters to an option must however appear before the next option appears on the command line.

Some options depend on others. This will be mentioned in the documentation. This is also controlled by SWEET, and an error message will appear if not all the depending options are given.

Some options and parameters have default values, i.e., values that will be used if no value is given.

5.1. The input files option

The input option

`-i, --input-files`

defines the input files to be read by SWEET. The format is:

`-i=<file-name(s)>`

The value "file-name(s)" specifies the path to the code to be analyzed. The value can contain one file name, or many. In the case of multiple files, these should be separated by "," (no space). Multiple ALF files will be linked together.

The parameter

`lang=<X>`

defines the language of the code. <X> can be one of:

`auto` - (default). This choice makes assumptions based on the file suffix (alf or cfg, see below).

`alf` One or more ALF file(s).

`cfg` File containing set of control-flow graphs. Useful for trace-based flow hypothesis generation (see -ae rtf).

The parameter

`func=<function-name>`

defines the start function for all analyses. By default the root function is the start function.

The parameter

annot=<file-name>

loads a file with abstract input annotations. The format of the annotation file is described in section "Abstract Input Annotations".

The parameter

unmangle

will unmangle identifiers in ALF code generated by melmac, by removing the ::<number> suffix which melmac appends to all identifiers. Useful to make linking of several melmac ALF files possible.

The parameter

rde

removes duplicate exports. When having functions that are exported from several ALF files, all but one definition of the function are removed. Which definition to keep is unspecified, since SWEET assumes that all definitions are identical. Default off.

The parameter

merged_output

controls how the values in the .out file are represented.

hmm

what is this option?

merged_output=no merges the output, i.e., each variable is represented as one value which includes all possible values the variable can have at the program point.

Although I just want the one input.

merged_output=yes

does not merge the output, i.e., each variable is represented as a list of several values (separated by "OR") which are possible at the program point.

5.2. The check statically option

The option:

-c,--check-statically

checks the parsed program for consistency. Parameters:

size=<X>

checks if the sizes of expressions are consistent, and if there are unsupported constructs in the program. <X> can be one of:

`on` - (default) Turns on this check

`off` - Turns off this check

connected=<X> Checks if the call graph is connected, i.e. if there is an unambiguous start point for the analysis. <X> can be one of:

`on` - (default) Turns on this check

`off` - Turns off this check

extref=<X> Checks if there are any unresolved external references in the program (like calls to external functions or accesses to external global variables). <X> can be one of:

`on` - (default) Turns on this check

`off` - Turns off this check

fcall=<X> Checks that (non-function-pointer) function calls are made with right amount of parameters, that each function has at least one return statement, that all return statements has the same amount of parameters, and that calls and returns are consistent in terms of parameters. <X> can be one of:

`on` - (default) Turns on this check

`off` - Turns off this check

If no parameters are given, all checks will be performed.

5.3. The load map option

The option:

-m,--map-file

loads a file mapping ALF statements to corresponding C-source code.

Parameters:

file=<file-name> specifies the file name of the map file. If this parameter is not given, the base name of the program file with the suffix ".map" will be used. In case more than one program file is given to -i, the corresponding .map files are assumed to be named as the base names of the program files, with ".map" appended. This parameter should then not be given.

rcs Read C sources - open C files specified in the map file and read C-lines from them to internal mappings.

c Check map file content. Controls that each basic block has a corresponding C source reference in the map file.

pal=<file-name> Print .map file after linking.

5.4. The domain option

The option:

-do,--domain

configures the abstract domain to be used by SWEET.

Parameters:

type=<X> The domain to be used for integers and offsets in pointers. <X> can be one of:

`int` - Intervals (default)

`clp` - Circular Linear Progressions

floats=<X> How to handle floats. <X> can be one of:

`top` - (default) All calculations of floating point values result in TOP (safe).

`est` - A best effort analysis that tries to estimate floating-point values rather than returning TOP, even though the analysis may be unsound (due to different floating point value representation on the host and on the target).

`reuse=<X>` Specifies that reuse (i.e. 'copy-on-write') of internal parts of abstract states should be used. Instead of always making deep copies of internal objects the states are allowed to share these objects. The actual copying takes place when a state wants to update its particular object. Saves memory and execution time for both VA and AE. If one or more options except 'all' or 'none' is given other one options will be turned off. Alternatives marked with '*' only concerns the AE. <X> can be one or more:

`all` - (default) All available reuse is on

`none` - No reuse

`m` - Memory

`c` - Call stack

`r` - Recorders*

`h` - Recorder holders*

`edo` Only read/write at offsets evenly divisible by the size of previously read/written values.

`noglobinit` Skips the global init section. Initial values will be top or restricted by annotations.

5.5. The abstract execution option

The option

`-ae,--abstract-execution`

runs abstract execution to produce flow facts. Parameters:

Flow facts generation.

`ffg=<X>` The flow fact generators (ffgs) to use. Given using four dimensions (as a four letter string):

1. bounds derived: upper (u), lower and upper (l), infeasible (i).

2. entities kept track of: headers (h), nodes (n), edges (e), call-edges (c), loop-body-begin-edges (b).

3. combination of entities: single (s), pairs (p), paths (n).

4. flow fact context: each iteration (e), all iterations (a), scope (s), function (f), program (p).

`<X>` can be one or more of :

`uhss` - (default) Upper header bounds in scope context (default on, but must be given if ffg is used). (u)

`lhss` - Lower header node bounds in scope context. (l)

`uhsf` - Upper header node bounds in function context. (u)

`lhsf` - Lower header node bounds in function context. (l)

`uhsp` - Upper header node bounds in program context. (u)

`lhsp` - Lower header node bounds in program context. (l)

`uhpf` - Upper bounds for header pairs in function context. (u)

`lhpf` - Lower bounds for header pairs in function context. (l) uhpp - Upper bounds for header pairs in program context. (u)

lhpp - Lower bounds for header pairs in program context. (l)

unss - Upper bounds for nodes in scope context. (u)

lnss - Lower bounds for nodes in scope context. (l)

unsf - Upper bounds for nodes in function context. (u)

lnsf - Lower bounds for nodes in function context. (l)

unsp - Upper bounds for nodes in program context. (u)

lnsp - Lower bounds for nodes in program context. (l)

inse - Nodes never taken for individual iterations of scope. (u)

insa - Nodes never taken for all iterations of scope. (u)

unps - Upper bounds for node pairs in scope-context. (u)

lnps - Lower bounds for node pairs in scope context. (l)

unpf - Upper bounds for node pairs in function-context. (u)

lnpf - Lower bounds for node pairs in function context. (l)

unpp - Upper bounds for node pairs in program-context. (u)

lnpp - Lower bounds for node pairs in program context. (l)

inpa - Excluding node pairs for all iterations of scope (ie. nodes never executed together for any iteration) (u). If no merging is used also reports on node pairs that must be taken together (ie. each iteration either none or both nodes must be taken).

uess - Upper bounds for edges in scope context. (u)

less - Lower bounds for edges in scope context. (l)

uesf - Upper bounds for edges in function context. (u)

lesf - Lower bounds for edges in function context. (l)

uesp - Upper bounds for edges in program context. (u)

lesp - Lower bounds for edges in program context. (l)

ucsf - Upper bounds for call edges in function context. (u)

lcsf - Lower bounds for call edges in function context. (l)

ucsp - Upper bounds for call edges in program context. (u)

lcsp - Lower bounds for call edges in program context. (l)

ubns - Upper bounds for sum of loop body begin edges in scope context (i.e. an upper bound on the number of times the loop body can be taken for each entry of the loop). (u)

lbns - Lower bounds for sum of loop body begin edges in scope context (i.e. bounds on the number of times the loop body can be taken for each entry of the loop). (l)

all - All ffgs should be used, i.e. both (u) and (l) ffgs allnoinse - All ffgs except inse (since inse may give problems in low-sweet)

ub - Upper bounds ffgs should be used. Includes ffgs marked with (u).

lb - Lower bounds ffgs should be used. Includes ffgs marked with (l).

merge=<X> The merge points to be used. See section Specification of merge points for more information. <X> can be one or more:

all - (default) All merge points (fe+fr+le+be+je).

none - No merging is used.

fe - Each function entry point.

fr - Each function return point.

le - Each loop exit edge.

be - Each back-edge.

je - Each point of joining edges.

SWEET is run with merging at all merge points as default. This leads to quicker termination of the AE, but might lead to less precise flow analysis results. So, a smart strategy to get optimal results is to first run SWEET without any merging at all with merge=none. If the analysis does not finish fast enough, try different merge points and see which gives the best results. (This strategy is automated in the 'optimal_merge' script; see Section "Perform calculation of loop bounds using AE with optimal merging".

Please note that infeasible path information is lost when some types of merging are used. For example, merging at each point of joining edges (merge=je) will impair the results for infeasible node pairs (e.g., fg=unpp). Therefore, avoid merging when infeasible paths are calculated.

df Execute in a depth first manner rather than breadth first manner.

usm Use unordered state merge strategy.

op Optimized single path execution.

debug=<X> Turns on debugging. <X> can be one of:

inter - Interactive debugger

What is this?

trace - Prints a trace to the file debug_msgs.txt

See Section SWEET's Debug Facilities for details.

tc=<X> Type counting. Reports for each type its number of occurrences during the execution. <X> can be one or more of:

all - Use all type countings, i.e., gn+op+st+sp.

gn - Generic nodes, i.e. nodes in the ALF syntax tree.

op - Operators, i.e. add, sub, ...

st - Statements, i.e. store, call, ...

sp - Statement pairs, i.e. <store,store>, <store,call>, ...

scp - Simple count printout, prints counts as integers on a single line (is actually no counter report, just a printout directive).

css Check single state, i.e., throw a run-time error if more than one state is generated during AE. Mostly for debugging purposes.

ene Extended handling of equal and not equal condition.

ft=<X> How to handle the fall-through state in switch statements that do not contain a default case. <X> can be one of:

`c` - (default) Continue with the fall-through state if it is not bottom (C standard, used by melmac).

`n` - Do not continue with the fall-through state (new ALF standard since late 2011).

`w` - Do not continue with the fall-through state and warn if it is not bottom - this may be due to overestimations.

vola=<X> How to handle volatiles. If volatiles occur in the program and AE is run this flag must be set. <X> can be one of:

`i` - Ignore volatile keyword in frame initializations. Treat volatile declared frames as normal frames.

`t` - Set all volatile frames to top values. All writes to frame will be ignored.

isi Ignore stores to integer addresses. Stores where the address is a concrete integer value are skipped, and therefore have no effect on the program state. Normally, such addresses are interpreted as TOP addresses, which results in that the stored value is stored to all locations in all memory frames, after first being LUBed with the value already stored at each location. This option is useful when analyzing code that contains stores to absolute memory addresses, for example, when memory-mapped I/O is used. With this option turned on, the analysis makes the unsafe assumption that these memory locations are disjoint from the memory locations used by regular variables in the program. Note that loads from such addresses are not skipped and will return TOP values, but that would happen in the normal case as well.

pu Process undefined functions, which means that calls to undefined functions will simply not be made. Further, all undefined global variables will initially be set to TOP, but become updated according to writes made during the AE analysis. The option must be set if AE should be run on code containing imports to functions that are not provided. Normally used together with the feature "do not check for unresolved external references" `-c extref=off` (see section The check statically option).

?? **gtf=<file-name>** Generate trace file, i.e., print the names of the CFG nodes taken during execution. Throws, similar to ccs, a run-time error if more than one state is generated during AE.

rtf=<file-name> Read trace from file and runs AE according to trace. A trace file can consist of several traces. A trace is a space-separated list of node names. Each trace ends with a semicolon.

Opa, da para rodar o AE seguindo um path???
that's what we need.

aac=<file-name> Generate BCET and WCET estimate using ALF AST construct costs. The input file holds the ALF AST construct costs. Requires the tc parameter to be set. Only cost for code constructs set by tc will be used in the BCET/WCET calculation.

aacpaths If set the aac option will also generate a BCET and WCET path based on the ALF AST construct costs. The paths will be printed to screen and are lists of basic block names. Can only be used together with aac.

aacprof If set the aac option will also generate profiling information based on the ALF AST construct costs. The profiles show the cost distributions across functions for the BCET and WCET paths, and are written to files named _ALFCostCalc_BC.txt and _ALFCostCalc_WC.txt, respectively. Can only be used together with aac.

oaac=<file-name> Old version of aac. Has same tc requirement.

bbc=<file-name> Generate BCET and WCET estimate using basic block and edges cost table. The input file holds the cost lookup table for basic blocks and edges. Table rows should be on form

`BBID COST` (node with single cost) or

`BBID COST1 COST2` (node with lower and upper cost) or

`BBID1 BBID2 COST` (edge with single cost) or

`BBID1 BBID2 COST1 COST2` (edge with lower and upper cost).

where BBID is a string not starting with a number or a minus, and COST is a (potentially negative) integer.

`bbcpaths` If set the bbc option will also generate a BCET and WCET path based on the BB cost lookup table. The paths will be printed to screen and are lists of basic block names. Can only be used together with bbc.

5.6. The timing database option

The option:

`-td,--timing-database`

creates a timing database (tdb) for the basic blocks of an ALF program based on cost lookup table (clt) holding timing costs for ALF code constructs.

Parameters:

`i=<clt-file-name>` Specifies the name of the input clt file to read timing for ALF code constructs from. If no file name is given default timing values will be used.

`o=<tdb-file-name>` Specifies the name of an output tdb file to which the result will be written to. If no file name is given the result will be written to stdout.

5.7. The value analysis option

what is the use of that?

The option:

`-va,--value-analysis`

runs value analysis (on nodes in scope graph).

Parameters:

`a=<X>` The analysis processing algorithm to use. <X> can be one of:

`n` - (default) Use worklist and process in optimal node order.

`l` - Use worklist but do not derive/use node ordering.

`j` - Use chaotic (Jacobi) fixpoint analysis.

`fp=<X>` The fixpoint passes to use. <X> can be one of:

`wn` - (default) Widening followed by narrowing.

`w` - Widening only.

`n` - Narrowing only.

`c` - Classic fixpoint analysis (no narrowing or widening).

`wnp=<X>` The widening and narrowing points to use. <X> can be one of:

`l` - (default) Do widening and narrowing before last stmt in loop header.

`b` - Do widening and narrowing at back edges.

`f` - Do widening and narrowing before first stmt in loop header.

`ft=<X>` How to handle the fall-through state. <X> can be one of:

`c` - (default) Continue with the fall-through state if it is not bottom (C standard, used by melmac).

`n` - Do not continue with the fall-through state (new ALF standard since late 2011).

`w` - Do not continue with the fall-through state and warn if it is not bottom - this may be due to overestimations.

`p=<X>` Print value analysis to text files. Generated files will be named as _AlfValueAnalysis_<f>_<i>.txt where i is iteration and f is fixpoint pass. <X> can be one of:

`r` - Only the final result is printed (least detailed).

`f` - The result after each fixpoint pass is printed.

`i` - The result of each iteration is printed (most detailed).

`d=<X>` Draw value analysis to DOT files. Generated files will be named as _AlfValueAnalysis_<f>_<i>.dot where i is iteration and f is fixpoint pass. <X> can be one of:

`r` - Only the final result is printed (least detailed).

`f` - The result after each fixpoint pass is printed.

`i` - The result of each iteration is printed (most detailed).

5.8. The def-use analysis option

The option

`-du,--def-use-analysis`

derives defines and uses of statements. Parameters:

`p` - Print DU analysis result to txt files. Generated files will be named _ALFDUAnalysis.txt.

5.9. The reaching definitions analysis option

The option

`-rd,--reaching-definitions-analysis` runs the reaching definitions analysis. Parameters:

`a=<X>` The analysis processing algorithm to use. <X> can be one of:

`n` - (default) Use worklist and process in optimal node order.

`l` - Use worklist but do not derive/use node ordering.

`j` - Use chaotic (Jacobi) fixpoint analysis.

`p=<X>` Print RD analysis to txt files. Generated files will be named as _ALFRDAnalysis_<i>.txt where i is iteration. <X> can be one of:

`r` - Only the final result is printed.

`i` - The result of each iteration is printed.

d=<X> Draw RD analysis to dot files. Generated files will be named as _ALFRDAnalysis_<i>.dot where i is iteration. <X> can be one of:

`r` - Only the final result is printed.

`i` - The result of each iteration is printed.

5.10. The program dependency graph option

The option

-pdg,--program-dependency-graph

derives a program dependency graph (PDG). Also derives other intermediate graphs, such as program control-flow graph (PCFG), program control-dependency graph (PCDG), program data-dependency graph (PDDG), as well as a def-use analysis (DU). See below for how to draw (-d) and print (-p) the graphs.

5.11. The program slicing option

The option

-sl,--slice

derives a slice based on a program dependency graph (PDG). Parameters:

ent=<X> The code entity/entities to start the slicing upon. Can not be used together with lab or var. <X> can be one of:

`c` - Conditions (includes loop exit conditions).

`l` - Loop exit conditions.

`g` - Global variables.

`i` - Call graph root function input variables.

`r` - Call graph root function return statements.

lab=<label(s)> Labels of code entities to start the slice upon. A comma separated list of strings. Can be statement label(s) or function name(s). If function name is selected, all statements in the function will be in the start slice. Can not be used together with ent or vars.

var=<variables(s)> Variables to start the slice upon. If several variables have the same name (due to local and global scoping) only one of them will be selected. If `dir=f` all code and data potentially affected by var(s) are derived. This is made by first deriving all statements which may use the var(s) and then do a forward from these statements. If `dir=b` all code and data potentially affecting var(s) are derived. This is made by first deriving all statements where var(s) may be defined. We then do a backward slice upon these nodes. Can not be used together with ent or vars.

mul=<X> How to slice upon multiple entities. <X> can be one of:

`t` - (default) One single slice together on all entities.

`i` - Individual slices on each selected entity.

dir=<X> The slicing direction to use. <X> can be one of:

`b` - (default) Backward, derives all statements and variables which may affect the selected entity.

`f` - Forward, derives all statements and variables which may be affected by the selected entity.

p=<X> Print slice result(s) to txt file(s). Generated file(s) will be named _ALFSlicing_<ent/lab>_<dir>.txt. <X> can be one or more:

`all` - Print all (stmts, funcs, global, local, and scoped vars).

`s` - Print labels of the statements.

`f` - Print names of the functions.

`g` - Print the global variables.

`l` - Print the local variables (using just their names).

`v` - Print the local variables with scope info (using func.var or func.scope.var naming).

inv Invert the slice before printing it, i.e., report what parts were removed from the slice as opposed to what parts remain in the slice. This is the recommended setting when using source code mappings (with --map-file), unless the provided map files are guaranteed to include mappings for all statements in the source code. Otherwise, if some source code statements do not have a mapping, it will appear as though they were sliced away since they are not included in the printed slice.

d Draw slice result(s) to dot file(s) as PDG graph. Generated file(s) will be named _ALFSlicing_<ent/lab/var>_<dir>.dot.

5.12. The flow facts option

The option

`-f,--flow-facts`

should I look into that?

I did and it was alright.

generates flow fact file(s). The generated flow facts depend on the flow fact generation algorithms used in the AE. Requires that the AE has been run using a fully context-sensitive scope graph. Parameters:

csl=<X> Specifies the largest call string length that should be used for generated flow information. Value should be either the keyword 'full' or a decimal integer >= 0. If csl value is not given zero context-sensitivity will be used.

lang=<X> The format of the produced flow facts. <X> can be one or more:

`ff` - (default) SWEET's flow fact format, see Section Flow Fact Language.

`sg` - Scope graph - includes both flow facts and a CFG.

`rapita` - The RAPITA format.

`ais` - The AIS format.

`tcd` - Produces a dummy tcd-file (containing only basic blocks, no statements).

o=<file-name> Specifies the flow fact file base name. By default the base name of the input file will be used.

co The result is printed to the console rather than to a disk file.

5.13. The print option

The option

`-p,--print-textually`

prints data in a textual format. Parameters:

`g=<X>` The data structures(s) to be printed. <X> can be one or more of:

`cfg` - (default) Control flow graph

`cfgd` - Control flow graph with debug info

`pa` - Pointer analysis

`cg` - Call graph

`sg` - Scope graph

`sym` - Symbol table

`ast` - Abstract syntax tree of the program

`astl` - Abstract syntax tree of the program after linkage

`expl` - Exported symbols after linkage.

`impl` - Imported symbols after linkage.

`o=<file-name>` specifies the graph file base name. The file name ending will indicate the file content. If not set content will be printed to stdout.

`co` The result is printed to the console rather than to a file.

5.14. The DOT graphs option

The option

`-d,--dot-print`

prints program graph(s) to dot file(s). Parameters:

`g=<X>` The graph(s) to be printed. <X> can be one or more of:

`cfg` - (default) Control flow graph

`cfgc` - Control flow graph, nodes annotated with C-code if mappings are available. Depends on optional loaded C-code mappings.

`cg` - Call graph

`rsg` - Reduced scope graph - cfg nodes are basic blocks

`fsg` - Full scope graph - cfg nodes are statements

`sgh` - Scope graph hierarchy - no cfg nodes are shown.

`pdg` - Program dependency graph (requires -pdg)

`pcfg` - Program control-flow graph (requires -pdg)

`pcdg` - Program control dependency graph (requires -pdg)

`pddg` - Program data dependency graph (requires -pdg)

`file=<file-name>` Specifies the base name of the file where to print the graph(s). A string indicating the format of the file will be appended + a '.dot' suffix. If not given, the program file base name will be used.

dag Prints a scope graph generated in DAG form rather than as a context sensitive graph. This parameter only applies to the printing of scope graphs.

tt The nodes in the graph will be labelled using integers, and the node names will appear as tool tips. This will make the graph more dense. Currently only notified by the scope hierarchy.

Drawing of parts of scope graphs are implemented for full (fsg) and reduced (rsg) scope graphs. The parameters "function" and "max_levels" are used to control the reduction in the following way:

`function` selects the root of the graph.

`max_levels` controls the number of function call levels the graph will contain. Function scopes below that level will be sketched as empty boxes.

5.15. The annotation templates option

The option

```
-at,--annotation-templates=<file-name>
```

tries to generate abstract annotation templates for each imported and referenced identifier (i.e., reference to undefined entity). For references that was not possible to make templates for it will output error messages to the console. The annotation value will be set to TOP in case the type is known, else the value is left to fill in manually. Other uncertainties may be found as comments in the file. NOTE! In case of an unresolved reference, the function may in general be able to update any existing global variable, and it's the user's responsibility to use the annotation. Information will be provided on the source code level in case label-to-C-source mapping is provided (option -m). Meaning of <file-name>: the name of the generated file.

5.16. The cost lookup template option

The option

```
-ct,--cost-lookup-table-template=<file-name>
```

generates a cost lookup table template. Members of selected types are printed with zero value.

Parameters:

types=<X> The ALF types to include in the printout. If not given all types will be used. <X> can be one or more of:

`all` - All types, i.e. pr, gn, op, st, sp.

`pr` - Program run, a basic cost for each program run.

`gn` - Generic nodes, i.e. nodes in ALF syntax tree.

`op` - Operators, i.e. add, sub, ...

`st` - Statements, i.e. store, call, ...

`sp` - Statement pairs, i.e. <store,store>, <store,call>, ...

5.17. The function information option

The option

```
-fi,--function-information=<func-name(s)>
```

extracts call-graphs for selected functions and print info.

where `<func-name(s)>` is a comma-separated list of function names. For each function given, its corresponding call-graph will be derived. The call-graph includes all functions reachable from the given function. Info will then be derived and printed to different files according to the p parameter with appropriate names, namely: `<dir><func>.alf` ,`<dir><func>.imports`, `<dir><func>.exports`, and `<dir><func>.template.annot`. All functions must be reachable from the current root function in the call-graph.

Parameters:

`p=<X>` The things to be printed. If not given all printouts will be used. <X> can be one or more of:

`all` - (default) All types, i.e. alf, imp, exp, at, map, info.

`alf` - ALF program, the result is written to `<dir><func>.alf`

`imp` - Imports, result is written to `<dir><func>.imports`

`exp` - Exports, result is written to `<dir><func>.exports`

`at` - Annot template, result is written to `<dir><func>.template.annots`

`map` - Map, result is written to `<dir><func>.map`. Will only produce result when `-m` is used.

`info` - Extra info file for statistics on files content.

`d=<directory>` The (path and) directory to which all files produced by this option will be written. If not given all files will be stored in the directory in which SWEET is run.

5.18. Misc. options

The option

`-cs,--code-statistics`

Didn't work

produces various information about the code structure. Parameters:

`o=<file-name>`: the (path and) file to which the information will be printed. If no file name is given the result will be printed to stdout.

The option

`-v,--version`

shows the SWEET build version and date.

The option

`-h,--help`

shows help text. Parameters:

`topic=<X>` Help topic, i.e., what to show help text on. <X> can be one of:

`option` - (default) Shows help on SWEET's arguments and options

`annot` - Shows help on SWEET's input annotation language

`ffg` - Shows help on flow fact generation using AE. Also gives an introduction to SWEET's (context-sensitive valid-at-entry-of) flow fact format.

`trace` - Shows help on flow hypotheses generation using traces.

`s` Use syntax high lighting on help text.

6. Abstract Input Annotations

Abstract input annotations in SWEET provide a way to assign abstract values to variables (represented by addresses) at certain program points (represented by labels). This is normally not possible in C. The purpose of using abstract input annotations is usually to enable a multi path analysis of a program.

Variables and labels in the annotations refer to the ALF representation of the analysed program. It must be noted that, since annotations refer to the ALF file, they often must be changed if the program changes and the ALF code is re-generated, or if the AlfBackend is updated. The most robust way of defining annotations is therefore to use placements such as PROG_ENTRY or FUNC_ENTRY. Placements such as STMT_ENTRY or STMT_EXIT should be avoided, if possible.

The annotations are stored in a text file and given as input to SWEET using the annot=<file-name> parameter of the -i option. The syntax of each line in the file is:

`<pos> <upd> <var> <val> [ || <var> <val> ]* ;`

where **`<pos>`** is where to place the assignment(s). It can be one of:

1. Before a given program statement holding a certain label:

```
STMT_ENTRY <func> <lrefid> <loffs>
```

Note: If <loffs> is = 0, it can be omitted.

2. After a given program statement holding a certain label:

```
STMT_EXIT <func> <lrefid> <loffs>
```

Note: If <loffs> is = 0, it can be omitted. Note that the program statement can not be a scope, call or return statement.

3. At the beginning of a function:

```
FUNC_ENTRY <func>
```

The assignments will be made after the function declarations and initializations.

4. Before the first function in the program is called:

```
PROG_ENTRY
```

The assignments will be made after the global declarations and initializations.

5. A global, always valid, volatile value assignment:

```
VOLATILE
```

The value stored will never change. Note: this type of annotation is not yet implemented.

6. A global, always valid, assignment made instead of calls to the given function:

```
WHEN_CALLED <func>
```

No call will be made (only the given variable assignments). Note: this type of annotation is not yet implemented.

`<upd>` defines how to update the analysis variable(s) with the annotation value(s). It can be one of:

1. Completely replace the variable's current analysis value(s) with the annotation value(s):

```
ASSIGN
```

2. Take the intersection of the variable's current analysis value(s) and the annotation value(s):

```
GLB
```

Can not be used together with `VOLATILE` or `WHEN_CALLED`.

3. Take the union of the variable's current analysis value(s) and the annotation value(s):

```
LUB
```

Can not be used together with `VOLATILE` or `WHEN_CALLED`.

`<var>` is the variable (i.e. the frame) to be updated. The frame can be given *directly*, using the frame id (i.e., the variable name), or it can be given *indirectly*. using dereferencing. This is marked by `<size> *` followed by the frame id of the pointer variable pointing at the frame (similar to C notation). More than one level of dereferencing is allowed. For example, two levels of dereferencing using pointers of size 32 bits for both pointers is indicated by "32 * 32 *".

There are four ways of specifying the updating of the variable:

1. The value is stored at the start of the frame (i.e. offset = 0), and the value has the same size as the whole frame:

```
[size *...size *] <frefid>
```

2. The value is stored at a given offset from the start of the frame, and the value has a certain size (which has to be defined):

```
[size *...size *] <frefid> <foffs> <size>
```

3. The store of a value is repeated a given number of times, starting from the offset:

```
[size *...size *] <frefid> <foffs> <size> <repeat>
```

With a frefid f, offset o, size s, and repeat r, then the value is stored at:

<f,o>, <f,o+s>, ..., <f,o+(r-1)s>

4. Store a value in the return value of a function:

```
RET_VAL <foffs> <size>
```

Can only used together with `WHEN_CALLED`.

Note: [ ] marks optional value.

`<val>` is the value to be stored. It can be either of:

1. An integer value:

```
INT [ <i> | <ilow> <iupp> | <ilow> <iupp> <icmul> ]
```

An integer value could either be a signed integer value `i`, or a `<ilow> <iupp>` pair (the interval [ilow..iupp]). The value icmul is the multiplier of a congruence (not used).

2. A float interval:

```
FLOAT [ <f> | <flow> <fupp> ]
```

An float value could either be a signed float value `f` or a `<flow> <fupp>` pair (the interval [flow..fupp]).

3. One or several data pointer values:

```
ADDR [ [ <frefid> | <frefid> <foffs> | <frefid> <foffslow> <foffsupp> ]+ ]
```

Each fval could be just a `<frefid>` (i.e. foffs=0), an `<frefid> <foffs>` pair, or `<frefid> <foffslow> <foffsupp>`, which means that the offset is an interval [foffslow..foffsupp].

4. One or several label pointer values:

```
LABEL [ [ <lrefid> | <lrefid> <loffs> ]+ ]
```

Each lval could be either a `<lrefid> <loffs>` pair, or just a `<lrefid>` (i.e. `loffs=0`).

5. A TOP value can be TOP ("super TOP") (i.e. nothing is known about the type) or of a certain subtype. A TOP value can represent any value, including the four subtypes of TOP.

```
TOP | TOP_INT | TOP_FLOAT | TOP_ADDR | TOP_LABEL
```

Explanation of the fields:

`<func>` - The ALF name of the function where to put the annotation statement. A string. May be surrounded with quotes (").

`<lrefid>` - An ALF label identifier for a statement. To form a complete label it must be associated with an offset. The label must be present in the program. A string. May be surrounded with quotes (").

`<loffs>` - A label offset. Used together with a lrefid to form a label. An unsigned integer. Note: not used (i.e., always assumed to be 0).

`<frefid>` - A frame identifier (i.e. a variable). Identifies the frame that should be updated by the assignment. When used in pointer values, `<frefid>` identifies the frame pointed to. A string. May be surrounded with quotes (").

`<foffs>` - The offset from the start address of the frame where the value will be stored. The offset is counted in bits. For an atomic data type the offset is 0, while for an array or a struct `<foffs>` gives the offset to the field to write to. When used in pointer values, `<foffs>` gives the relative offset from the start address of `<frefid>` which should be pointed to, i.e. the address at `<frefid> + <offs>`. An unsigned integer.

`<foffslow>` - The lower bound of an offset interval counted in bits. An unsigned integer <= foffsupp.

`<foffsupp>` - The upper bound of an offset interval counted in bits. An unsigned integer >= foffslow.

`<size>` - The size of the value that will be stored. Counted in bits. An unsigned integer.

`<repeat>` - The number of times the given value storage will be repeated at consecutive addresses in the same frame. An unsigned integer >= 1.

`<i>` - A signed integer value.

`<ilow>` - The lower limit in an integer interval value. A signed integer value <= iupp.

`<iupp>` - The upper limit in an integer interval value. A signed integer value >= ilow.

`<icmul>` - The multiplier of a congruence. The resulting domain is: <ilow> + <icmul>Z. Not used.

`<f>` - An signed floating point value.

`<flow>` - The lower limit in a floating point interval value. A signed floating point value <= fupp.

`<fupp>` - The upper limit in a floating point interval value. A signed float value >= flow.

Each annotation must end with a semicolon.

The `||` notation is used for making a parallel store of abstract values. At least one <upd> <var> <val> tuple must be given. It is allowed for two parallel assignments to update the same frame, but the values to be stored must however not overlap in the frame. This is checked by SWEET, which will report an error if overlaps occur (or may occur). It is not allowed to refer to the same program point by several annotations. Use parallel assignments if several annotations should be done at the same program point. (Note that an assignment to STMT_ENTRY and an assignment to STMT_EXIT to the same label are considered as separate program points). It is not allowed to assign values outside a frame.

Comments may be given in the annotation file using C or C++ style comments.

Integer values are given using C syntax, i.e., decimal values start with an integer > 0 or a '-', hexadecimal values start with 0x or 0X, and octal values start with a 0.

Floating values are also given using C syntax, for example as 5, 1.25, 2.457E2, or -3.7E-4. Floats in annotations are currently assumed to be of 32 or 64 bits size. If the float is given another size an error will be generated. This is because we do not know how many bits the exponent and fraction of the float should occupy. We currently assume IEEE 754-2008 floats, i.e. 32-bits floats are assumed to have 8 bits exp and 23 bits frac, while 64-bits floats are assumed to have 11 bits exp and 52 bits frac. If the target computer uses another floating point standard, it is wise to use the setting

```
-do floats=top
```

when analysing the code. Doing so, the floating point calculations will always be safe (yield TOP_FLOAT), but the annotations will of course be meaningless. See section The Domain Option for more details.

When using a frefid that exists in several ALF scopes, the normal scoping rules of ALF will be used to find the right frame to update, i.e. first the local scopes on the current function scope will be searched (if any), then the function scope, and finally the global scope. If no such frefid can be found an error will be generated.

A PROG_ENTRY annotation will take effect after all global initializations have been made, but before the first statement has been taken. Thus, PROG_ENTRY annotations can only refer to global frame identifiers. When having an FUNC_ENTRY annotation on the first function in the program, the PROG_ENTRY annotation will be processed before the FUNC_ENTRY annotation. Similarly, when having a STMT_ENTRY annotation on the first statement in the program the PROG_ENTRY annotation will be processed before the STMT_ENTRY annotation.

When having both an FUNC_ENTRY annotation and a STMT_ENTRY annotation on the first statement in the same function the FUNC_ENTRY annotation will be processed before the STMT_ENTRY annotation.

When an STMT_EXIT annotation refers to a statement which is the predecessor to a statement which have a STMT_ENTRY annotation, the STMT_EXIT annotation will be processed before the given STMT_ENTRY annotation.

A VOLATILE annotation is only applicable to global frefs. In ALF, a volatile fref holds data whose contents may change at any time in a way not under the control of the program. A fref (or parts of a fref) assigned an abstract value using the VOLATILE keyword will always hold the given value. Thus, normal assignments or other type of value annotations to volatile variables will have no effect. This also holds for a variable declared volatile in ALF, for which only the VOLATILE annotation can be used to assign it an (abstract) value. The annotation should provide a safe upper bounds on what values the fref may hold during all possible executions of the program.

A WHEN_CALLED annotation will be applied whenever a call to the given function should be made, (either using normal function calls or using function pointers). The function call will then not be made (and thus no flow facts will be generated for the function). Instead, the execution will continue at the statement following the call statement, but with variables updated according to the given value assignments. The annotation should be valid for all possible calls to the function within the analyzed program for all possible executions of the program. In ALF sizes of imported variables are unknown. The same holds for the sizes of return values of imported functions. This means that when assigning a value to an imported fref or assigning a return value to an imported function, an offset and a size must always be provided.

ANNOTATION EXAMPLES

```
STMT_ENTRY main BB0 ASSIGN x INT -1 5; // Annotation 1

STMT_EXIT foo BB72 5 ASSIGN x INT 3 4 || y INT 2; // Annotation 2

FUNC_ENTRY bar ASSIGN s 0 32 INT 1 67 || s 32 16 INT -12 14 || s
48 32 ADDR x; // Annotation 3

PROG_ENTRY ASSIGN g ADDR c d 8 12 e 16; // Annotation 4

PROG_ENTRY ASSIGN w 0 8 20 INT -1; // Annotation 5

FUNC_ENTRY baz ASSIGN f 32 64 FLOAT -1.23 2.457E2; // Annotation 6

STMT_EXIT main BB12 ASSIGN z INT 0 6 2 || v INT -2 8 3; // Annotation 7

FUNC_ENTRY ko ASSIGN p TOP_ADDR; // Annotation 8

FUNC_ENTRY mu ASSIGN fp LABEL main ko mu; // Annotation 9

STMT_ENTRY bu BB7 0 GLB w INT 1 100; // Annotation 10

STMT_EXIT bu BB7 0 LUB w INT 0 70; // Annotation 11

VOLATILE ASSIGN k INT 66 68; // Annotation 12

VOLATILE ASSIGN u 0 8 INT 5 || k 9 16 INT 4; // Annotation 13

WHEN_CALLED foo ASSIGN g INT 6 7 || RET_VAL 0 32 TOP_INT; // Annotation
14

STMT_ENTRY "main" "main::entry::4" ASSIGN 32 * 32 * "pp_pi" FLOAT
3.14 // Annotation 15
```

Explanations:

Annotation 1 adds an annotation before the statement labelled <BB0,0> in function main which assigns integer interval [-1..5] to variable x.

Annotation 2 adds an annotation after the statement labelled <BB72,5> in function foo which assigns integer interval [3..4] to variable x and value 2 to variable y.

Annotation 3 adds an annotation after entry of function bar which updates a larger aggregate data structure s with several values. It assigns the integer interval [1..67] to the first 32 bits of s, interval [-12..14] to the following 16 bits of s, and a pointer value holding the addresses of variable x to the following 32 bits.

Annotation 4 adds an annotation at the global program scope which assigns a set of pointer values to global variable g. The pointer g will point to the start of the global variable c, or offset 8 to 12 in the global data structure d, or to offset 16 in the global data structure e. Note: if the offset is not given (ADDR d) it is interpreted as ADDR d 0.

Annotation 5 adds an annotation at the global program scope which assigns -1 to each of the first 20 bytes of a global data structure w.

Annotation 6 adds an annotation at the entry of function baz which assigns a 64 bit interval float value -1.23 to 2.457 * 10^2 to bits 32 to 96 of variable f.

Annotation 7 adds an annotation after the statement labelled <BB12,0> in function main which assigns integer interval [0..6] and conference 0 + 2Z, (whose intersection is the values 0,2,4,6), to z. Similarly, v is assigned the integer interval -2..9 and the congruence -2 + 3Z, (whose intersection is the values -2,1,4,7). Note: congruences are not currently supported by SWEET.

Annotation 8 adds an annotation after entry of function ko which assigns the data pointer p to hold a TOP_ADDR pointer value (i.e., all possible data addresses).

Annotation 9 adds an annotation after entry of function mu which assigns the function pointer fp to hold the start addresses of functions main, ko and my respectively.

Annotation 10 adds an annotation before the statement labelled <BB7,0> in function bu which specifies that w should be given the intersection of its current analysis value and the [1..100] interval. If, for example, the current analysis value is [50..200] the resulting value will be [50..100].

Annotation 11 adds an annotation before the statement labelled <BB7,0> in function bu which specifies that w should be given the union of its current analysis value and the [0..70] interval. If, for example, the current analysis value is [50..100] the resulting value will be [0..100].

Annotation 12 sets the k variable imp to always hold the [66..68] interval.

Annotation 13 sets the first 8 bits of u to always hold 5 and the following 8 bits to always hold 4.

Annotation 14 sets every call to function foo to store [6..7] in global variable g. It also sets the first 32 bit of the function's return value to a 32 bit TOP_INT value.

Annotation 15 sets the variable pointed to by two indirections to 3.14. Pointer size 32 is used for both pointers, starting with the pointer "pp_pi". The pointers and the variable could have been declared in the following way in a C program:

```
float pi;

float * p_pi = &pi;

float ** pp_pi = &p_pi;
```

BNF GRAMMAR

absann_file -> absann_list

absann_list -> absann_list absann

absann -> position update var_val_list ;

position -> STMT_ENTRY func lrefid | STMT_ENTRY func lrefid loffs | STMT_ENTRY func lrefid | STMT_EXIT func lrefid loffs | STMT_EXIT func lrefid | FUNC_ENTRY func | PROG_ENTRY | VOLATILE | WHEN_CALLED func

update -> ASSIGN | GLB (not for VOLATILE_ASSIGN or WHEN_CALLED) | LUB (not for VOLATILE_ASSIGN or WHEN_CALLED)

var_val_list -> var_val | var_val_list || var_val

var_val -> var vals

var -> psizes_stars fid | psizes_stars fid offs size | psizes_stars fid offs size repeat

psizes_stars -> psize * psizes_stars |

fid -> frefid | RET_VAL (only for WHEN_CALLED)

var -> frefid

vals -> vals OR val | val

val -> INT i | INT ilow iupp | INT ilow iupp icmul | FLOAT f | FLOAT flow fupp | ADDR frefid_foffs_list | LABEL lrefid_loffs_list | TOP | TOP_INT | TOP_FLOAT | TOP_ADDR | TOP_LABEL

frefid_foffs_list -> frefid | frefid foffs | frefid foffs foffs | frefid_foffs_list frefid | frefid_foffs_list frefid foffs | frefid_foffs_list frefid foffs foffs

lrefid_loffs_list -> lrefid | lrefid loffs | lrefid_offset_list lrefid | lrefid_offset_list lrefid loffs

func -> <string>

lrefid -> <string>

loffs -> <unsigned_int>

frefid -> <string>

foffs -> <unsigned_int>

size -> <unsigned_int>

psize -> <unsigned int>

repeat -> <unsigned_int>

i -> <signed int>

ilow -> <signed_int>

iupp -> <signed_int>

icmul -> <unsigned_int>

f -> <float>

flow -> <float>

fupp -> <float>

6.1. Alternative to Abstract Input Annotations

Abstract input annotations are used to set variables to abstract values. This is normally not possible in C, since variables are expected to have a single value. There are, however, some "tricks" in C which you can use as alternatives to abstract input annotations when you want to create abstract values. One example follows.

- Testing an uninitialized value and then merge at join. Example: create an interval:

  The variable `x` in the code snippet

  ```
  int i, x;

  if (i)

  x = 1;

  else
  ```

```
x = 10;
```

will have the value [1..10] in the SWEET analysis after the code is analysed. This requires that merge at join is used with the SWEET command:

```
sweet -i=pgm.alf,std_hll.alf -ae merge=je
```

Input annotations have the advantage, however, that the program source code does not have to be touched.

7. Output Annotation Specifications

Output annotation specifications provide a way in SWEET to access values of variables at certain program points during abstract execution. Output annotation specifications are stored in a text file and given as input to SWEET using the outannot=<file-name> parameter of the -i option. They define where in a program and for which variable the value of the variable should be accessed.

what might be the use of that.

The output is in the form of an abstract annotation (see Section Abstract Input Annotations). This is beneficial since it simplifies iterative uses of SWEET where obtained analysis results are used in new analyses as inputs.

The syntax of an “output annotation specification” given to SWEET is:

```
<pos> <var> ;
```

The syntax of an “output annotation” (the result) after analysis is:

```
<pos> ASSIGN <var> <val> ;
```

The update type is ASSIGN since it replaces the variable's current analysis value with the annotation value. If <pos> is visited several times during abstract execution of the analysed program, then an output annotation with the least upper bound of the possible values is produced. For example, if the value of the integer variable is [2..7] in one iteration of the enclosing loop, and [3..10] in the second, the result will be [2..10].

<pos> is where to place the assignment(s). It can be one of:

1. Before a given program statement holding a certain label:

```
STMT_ENTRY <func> <lrefid> <loffs>
```

Note: If <loffs> is = 0, it can be omitted.

2. After a given program statement holding a certain label:

```
STMT_EXIT <func> <lrefid> <loffs>
```

Note: If <loffs> is = 0, it can be omitted. Note that the program statement can not be a scope, call or return statement.

<var> is the variable (i.e. the frame) to be accessed. It can be one of:

1. The value is stored at the start of the frame (i.e. offset = 0), and the value has the same size as the whole frame:

```
<frefid>
```

2. The value is stored at a given offset from the start of the frame, and the size of the stored value has a certain size:

```
<frefid> <foffs> <size>
```

where foffs and size are given in bits.

If <size> is left out, then SWEET uses the current size of the variable.

The BNF grammar for output annotations is a subset of the BNF grammar for abstract input annotations.

Example of the use of output annotations:

Assume that we have a program ex1.c [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/ex1.c] with a global variable k that we are interested in the possible values of at different program points. After translation to ALF using the AlfBackend, we have the file ex1.alf [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/ex1.alf] in which we look for positions to put in the output annotation specification. The file ex1.ann [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/ex1.ann] contains abstract input annotations that set the variables i and j to intervals:

```
PROG_ENTRY ASSIGN i INT 1 5 || j INT 2 10 ;
```

We decide to access the variable k in the loop and before the return point. After looking in the ALF file we find the labels for these two program points, and the resulting output annotation specifications are:

```
STMT_EXIT main main::bb3::5 0 k ;
```

```
STMT_ENTRY main main::bb20::3 0 k ;
```

We write these lines in the file ex1.oas (output annotation specification) and run the AE with the command

```
sweet -i=ex1.alf,std_hll.alf outannot=ex1.oas annot=ex1.ann -ae
```

The resulting file ex1.out looks like the following:

```
STMT_EXIT main main::bb3::5 0 ASSIGN "k" 0 32 INT 0 1010 ;
```

```
STMT_ENTRY main main::bb20::3 0 ASSIGN "k" 0 32 INT 10 1011 ;
```

showing the size (32) and possible values of k in the loop, where it is [0..1010], and at the return point, where it is [10..1011].

Output annotation specifications can be used for all types of variables. Note that local variables often are stored in temporaries in ALF code generated by the AlfBackend. This may make it harder to define the variable to access. This is due to the way LLVM works.

Output annotation specifications can be used for all types of values in SWEET (INT, FLOAT, ADDR and LABEL). The following shows an example of the use of all of these types. The program ex2.c [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/ex2.c] contains the integer i, the float f, the pointer to integer p and the function pointer pt2f.

After translation to ALF using the AlfBackend, we have the file ex2.alf [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/ex2.alf]. We now do the following with the values:

1. Observe the initial values of the variables. This is done with the output annotation

   ```
   STMT_EXIT main main::bb 0 "i" || "f" || "p" || "pt2f" ;
   ```

   in the .oas file. The basic block "main::bb" is the first basic block in main.

   This gives the output

   ```
   STMT_EXIT main main::bb 0 ASSIGN "f" 0 32 FLOAT 0.5 || "i" 0 32
   INT 0 || "p" 0 64 ADDR "arr" || "pt2f" 0 64 ADDR "$null" ;
   ```

E se retornarmos o
valor do input
para o main?

in the .out file.

2. Set the variables to some abstract values. This is done with the abstract annotation

   ```
   STMT_ENTRY main main::bb6::7 0 ASSIGN "f" 0 32 FLOAT 1 3.14159
   || "i" 0 32 INT 1 2 || "p" 0 64 ADDR "arr" 0 4 || "pt2f" 0 64
   LABEL f1 f2 ;
   ```

   in the .ann file. These assignments are done at the start of the basic block with label main::bb6::7, which is situated after the if-else statement in the program.

3. Observe the resulting values of the variables after step 2. This is done with the output annotation

   ```
   STMT_EXIT main main::bb6::7 0 "i" || "f" || "p" || "pt2f" ;
   ```

   in the .oas file. The values are accessed at the end of the basic block with label main::bb6::7.

   This gives the output

   ```
   STMT_EXIT main main::bb6::7 0 ASSIGN "f" 0 32 FLOAT 1 3.14159 || "i"
   0 32 INT 1 2 || "p" 0 64 ADDR "arr" 0 4 || "pt2f" 0 64 LABEL f1 f2 ;
   ```

   in the .out file.

4. Set the variables to some concrete values. This is done with the abstract annotation

   ```
   STMT_ENTRY main main::bb6::10 0 ASSIGN "f" 0 32 FLOAT 2.66 || "i"
   0 32 INT 7 || "p" 0 64 ADDR "arr" 8 || "pt2f" 0 64 LABEL "f1" ;
   ```

   in the .ann file. These assignments are done at the start of the basic block with label main::bb6::10, which is after basic block addressed in step 2 and 3.

5. Observe the resulting values of the variables after step 4. This is done with the output annotation

   ```
   STMT_EXIT main main::bb6::10 0 "i" || "f" || "p" || "pt2f" ;
   ```

   in the .oas file. The values are accessed at the end of the basic block with label main::bb6::10.

   This gives the output

   ```
   STMT_EXIT main main::bb6::10 0 ASSIGN "f" 0 32 FLOAT 2.66 || "i" 0
   32 INT 7 || "p" 0 64 ADDR "arr" 8 || "pt2f" 0 64 LABEL f1 ;
   ```

   in the .out file.

The SWEET command to accomplish this is

```
sweet -i=ex2.alf,std_hll.alf annot=ex2.ann outannot=ex2.oas -ae -do
floats=est
```

The input files are ex2.ann [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/ex2.ann] and ex2.oas [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/ex2.oas]. The results are written to ex2.out [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/ex2.out].

There is also a parameter

`merged_output`

which controls how the values in the .out file are represented.

`merged_output=yes` merges the output, i.e., each variable is represented as one value which includes all possible values the variable can have at the program point.

merged_output=no does not merge the output, i.e., each variable is represented as a list of several values (separated by "OR") which are possible at the program point.

Example of the use of the parameter:

The program example.alf [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/example.alf] generates different values for the address "jump_addr" at the start of the function "dyn_jump". We are interested in the possible values of that address. We generate the following output specification (file dyn_jump.oas [http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/dyn_jump.oas]) to access the values of the address at that program point:

```
STMT_ENTRY start dyn_jump 0 jump_addr ;
```

The SWEET command to accomplish this is

```
sweet -i=example.alf outannot=dyn_jump.oas merged_output=no -c -ae
merge=none ft=n -f
```

which gives the resulting output

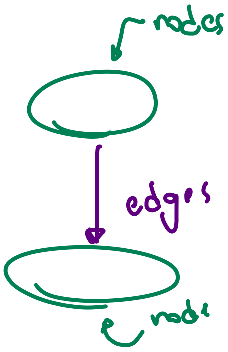
```
STMT_ENTRY start dyn_jump 0 ASSIGN "jump_addr" 0 16 INT 540 OR INT
548 OR INT 523 ;
```

which means that the address "jump_addr" is 540, 548 or 523. Should we have used the parameter value merged_output=yes, we should have the result

```
STMT_ENTRY start dyn_jump 0 ASSIGN "jump_addr" 0 16 INT 523 548 ;
```

instead, which gives the address as an interval. That is not what we wanted.

8. SWEET's Flow Fact Language



SWEET can produce flow information in several different formats. Basically, flow information gives safe lower and/or upper bounds on how many times different code entities, such as control-flow graph nodes or edges, are executed in different program contexts. For programs with many different execution paths, the flow information provides lower and/or upper bounds of the collected execution behaviour of all these paths.

SWEET's value annotations (see Section Abstract Input Annotations) can be used to specify constraints on possible input values of a program. The derived bounds holds for all possible concrete input value combinations and their corresponding program executions. For example, if the following code snippet may be run with input value x equal to 0, 1, or 2, then an upper bound on the number of times node BB0 (the loop header node) may be taken is 3 (given by x=0) and a lower bound is 1 (given by x=2):

```
int func(int x) {
 while(x < 2) // BB0
  x++;
 return x;
}
```

The given bound on BB0's executions is actually a local bound, valid each time func is entered. If we, for example, consider BB0's executions in a global program context, i.e. a bound on the number of times BB0 can be taken during the whole program's execution, it might be a value much larger than 3 (if func can be called several times during program execution). SWEET allows flow information to be derived and also to be expressed in different contexts. For example, loop bounds can be derived and given both in a function-local or in a program-global context. Also, flow information can be constrained to only hold in certain function call-string contexts.

The context-sensitive valid-at-entry-of flow fact format.

The (new) format for SWEET's flow information is called 'context-sensitive valid-at-entry-of flow facts'. Each generated flow fact has the following format (see the end of section for a BNF grammar):

```
CALL_STRING : VALID_AT_ENTRY_OF : FOREACH_OR_TOTAL : CONSTRAINT ;
```

`CALL_STRING` is a list of function calls that must have been executed for the flow fact to hold. The calls start with the program start function. The call string is represented as a list of ((calling_func, calling_stmt), called_func) tuples.

`VALID_AT_ENTRY_OF` is a function or a loop in a function. The flow fact holds each time the given function or loop is entered until it is exited. It provides a way to give flow information in a local, semi- local or global context. When referring to functions the VALID_AT_ENTRY_OF is a function identifier, while referring to loops it is a (func, stmt) tuple.

`FOREACH_OR_TOTAL` is `<>, <i..j>` (foreach type) or `[]` (total type).

`FOREACH <>` concerns iterations of loops, and specifies that the flow fact is valid for each individual iteration of the loop specified in VALID_AT_ENTRY_OF.

`FOREACH <i..j>` concerns iterations of loops, and specifies that flow fact holds for each individual iteration starting at the i:th and ending at the j:th iteration of the loop. The iteration (i.e. what i and j are compared against) of a loop is assumed to be reset every time the loop is entered and incremented every time the loop header node is taken.

`TOTAL []` concerns either a function or a loop, and specifies that the flow fact is valid for the sum of executions, from the entry to exit of the VALID_AT_ENTRY_OF.

`CONSTRAINT` is a linear constraint relating the execution of program entities, such as nodes, (e.g., BB0) or edges, (e.g., BB0->BB1) in the program CFGs to integer constants.

Illustrative example on the usage of call-strings and entry-of.

Consider the following code snippet:

```
main() {  // BB0
 baz(3);  // BB1
 foo(5);  // BB2
}
foo(int x) {  // BB3
 bar(x);  // BB4
 bar(x-1); // BB5
}
baz(int z) {  // BB6
 for(i=0; i<2; i++)
  bar(z); // BB7
}
bar(int y) {  // BB8
 int i=0;
 while(i < y) // BB9
 i++;
}
```

We note that bar is called from several places and that the loop in bar execute differently depending on the parameter bar was called with. Assuming that we want to express (and derive) a loop bound on the loop in bar we can do this in a number of ways:

- We can have a loop bound which is valid each time bar is entered, (i.e. a local loop bound) independently on what function calls made to reach bar. The loop bound is expressed as a constraint on the number of times the loop header node may be executed (note the empty call string) (e.g. given by `sweet -ae ffg=uhsf csl=0`)

```
: : bar : [] : BB9 <= 6 ;
```

- We can have local loop bounds valid for each entry of bar, for each possible call string down to bar (e.g. given by `sweet -ae ffg=uhsf -f`):

  ```
  ((main,BB1),baz) ((baz,BB7),bar) : bar : [] : BB9 <= 4 ;
  ((main,BB2),foo) ((foo,BB4),bar) : bar : [] : BB9 <= 6 ;
  ((main,BB2),foo) ((foo,BB5),bar) : bar : [] : BB9 <= 5 ;
  ```

  Note that we differ between the two call-sites in foo to bar.

- We can have a global loop bound, providing a bound on the number of times BB9 may be taken valid for the whole program execution (e.g. given by `sweet -ae ffg=uhsp csl=0`):

  ```
  : main : [] : BB9 <= 19 ;
  ```

  Note that bar is called twice in baz.

- We can have local loop bounds valid for each entry of bar, but separated on each individual one-length call-strings, i.e. we give different local loop bounds for the loop in bar depending on if the execution reached bar through baz or foo (e.g. given by `sweet -ae ffg=uhsf csl=1`):

- `((main,BB1),baz) : bar : [] : BB9 <= 4 ; ((main,BB2),foo) : bar : [] : BB9 <= 6 ;`

Illustrative example on valid-at-entry-of loop flow facts

The above examples only provided flow facts valid for an entry of a particular function. We can also give flow facts valid for each entry of a particular loop. The loop is identified by its loop header cfg node. The following code snippet, which holds a loop-nest, illustrates the idea.

```
bip(int x) {   // BB0
 int i = 0;
 while(i < x) {  // BB1
  int j = i; // BB2
  while(j < x) { // BB3
   j++; // BB4
  }
  i++;   // BB5
 }
}
```

Assuming that we provide an input value annotation of x = 1..10 then an upper bound on the number of times that BB1 (the loop header node) can be taken for each entry of the outer loop is 11, given by x=10 (e.g. given by `sweet -ae ffg=uhss csl=0`):

```
: (bip,BB1) : [] : BB1 <= 11 ;
```

For the inner loop we can provide a local loop bound valid for each entry of the inner loop from the outer loop (given by x=10 at first iteration of outer loop) (e.g. given by sweet `-ae ffg=uhss csl=0`):

```
: (bip,BB3) : [] : BB3 <= 11 ;
```

Using the two flow facts given above in a WCET calculation it would be assumed that BB3 will be taken 10*11=110 times each time the bip function is entered (note that loop iterations bounds are given the header nodes). This is however a pessimistic value, since BB3 can actually only be taken 11+10+9+..+2=65 times each time the outer loop is entered. The latter may result in a tighter WCET estimate and can be expressed as (e.g. given by `sweet -ae ffg=uhsf csl=0`):

```
: bip : [] : BB3 <= 65 ;
```

Illustrative example on foreach flow facts.

Compared to the total ([]) specifier used in the above examples, the foreach specifiers (<>, <i..j>) allow use to give constraints for each individual iteration, or some specific iterations of a loop. Consider the following code snippet:

```
bup(x) {   // BB0
 int i = 0;
 while(x <= 3) {  // BB1
  if(x > 0) // BB2
   i++; // BB3
  else
   i--; // BB4
  if(x > 2) // BB5
   i++; // BB6
  else
   i--; // BB7
  if(x <= 2) // BB8
   i++; // BB9
  else
   i--; // BB10
  x++;  // BB11
 }
}

main() {   // BB12
bup(2);    // BB13
bup(1);    // BB14
}
```

First, we can give a number of different loop bound flow facts (e.g. given by `sweet -ae ffg=uhsf -f`):

```
((main,BB13),bup) : bup : [] : BB1 <= 3 ;

((main,BB14),bup) : bup : [] : BB1 <= 4 ;
```

or (e.g. given by `sweet -ae ffg=uhsf csl=0`):

```
: bup : [] : BB1 <= 4 ;
```

or (e.g. given by `sweet -ae ffg=uhsp -f`):

```
: main : [] : BB2 <= 7 ;
```

We note that the false branch of the BB2 if-condition will never be taken during any loop iteration (i.e. an infeasible node). This may be expressed by (e.g. given by `sweet -ae ffg=insa csl=0`):

```
: (bup,BB1) : <> : BB4 = 0 ;
```

We also note that the true branches of the BB5 and BB8 if-conditions, as well as the two false branches, are exclusive, i.e. they can never be taken together during any iteration (even though each node may be taken during some iteration) (e.g. given by `sweet -ae ffg=inpa csl=0`):

```
: (bup,BB1) : <> : BB6 + BB9 < 2 ; : (bup,BB1) : <> : BB7 + BB10 < 2 ;
```

We can also derive flow information on longer infeasible paths, i.e. paths structurally possible, but not possible to execute in reality due to possible (input) data values. For example, the following specifies that nodes BB2, BB3, BB5, BB6, BB8, and BB9 never can be taken together during an loop iteration. This can be expressed by (e.g. given by `sweet -ae ffg=inna csl=0`):

```
: (bup,BB1) : <> : BB2 + BB3 +BB5 + BB6 + BB8 + BB9 <= 5 ;
```

Looking in even more detail we can note that, for each entry of the loop, neither BB6 nor BB10 can be taken during the first loop iteration. This can be expressed by (e.g. given by `sweet -ae ffg=inse csl=0`):

```
: (bup,BB1) : <1..1> : BB6 = 0 ;
```

```
: (bup,BB1) : <1..1> : BB10 = 0 ;
```

Abstract Execution (AE), recorder and collectors.

To be able to generate flow facts the AE extends abstract states with recorders. The recorders gather information on how a state has been executed in a certain part of the program, e.g., how many times different nodes or edges have been taken in a specific loop. Additionally, the program parts for which flow information should be derived (e.g. loops), get so called collectors associated to them. The collectors are used to successively accumulate recorded information from the states. The AE is turned on in SWEET using the -ae parameter. The ffg parameter specifies what flow fact generators to use, basically allowing SWEET to set up needed management for the requested recorders and collectors.

There are many different type of recorders and collectors, each resulting in the generation of certain type of flow facts. Thus, the ffg can be given more than one value if several types of flow facts should be generated during the same AE run. The values to the ffg parameter are specified using four different dimensions, (each values is provided as a four letter string):

1. bounds derived: upper (u), lower and upper (l), infeasible (i).

2. entities kept track of: headers (h), nodes (n), edges (e), call edges (c), loop body begin edges (b).

3. combination of entities: single (s), pairs (p), paths (n).

4. flow fact context: each iteration (e), all iterations (a), scope (s), function (f), program (p).

For example, `-ae ffg=uhss` tells the AE to create recorders and collectors deriving local upper header bounds, while `-ae ffg=lnsp` tells the AE to derive lower and upper bounds on the number of times different nodes may be taken during each execution of the program.

SWEET builds it flow analysis on a program representation called scope graph. Basically, the nodes and edges in different CFGs are divided into scopes, where each scope correspond to a loop, function, or recursive call-nest. The current version of the scope-graph is fully context-sensitive, i.e., if a function is called from several different call-sites, each call will result in a new scope for the function. The collectors are associated to scopes, and the flow information derived by the collectors are therefore context-sensitive as well. The old type of flow facts derived by SWEET where also highly connected to scopes, basically specifying that a given flow fact should be valid for each entry or iteration of a certain scope. For example:

```
loop_scope : [] : BB_loop_header <= 10 ;
```

Generation of context-sensitive valid-at-entry-of flow facts.

The generation of the new type of context-sensitive valid-at-entry-of flow facts is turned on using the parameter. It requires the AE to have been run and uses flow information collected by collectors associated to the scopes in a (fully context-sensitive) scope graph. The context-sensitive flow information is then partitioned and merged in respect to the context-sensitivity (i.e. the call-string length) requested. The l=<number> option specifies the call-string length to use (without giving the l option full context-sensitivity will be used). For example, `sweet -ae ffg=uhsf csl=0` specifies that we should derive loop bounds for all loops valid at each entry of the function in which they belong, and independently on the function calls used to reach the function. `sweet -ae ffg=uhsf csl=2` specifies that we should derive the same type of loop bounds but differ between call string of length 2. Similarly, `sweet -ae ffg=uhsf -f` also specifies that function local loop bounds should be derived. However, we will use full-context sensitivity, that is, we will separate between functions which have different call paths. For more examples on the usage of the -f option, see above. The -f parameter can currently not be run on recursive programs.

BNF grammar for context-sensitive valid-at-entry-of flow facts.

ff_file -> ff_list

ff_list -> ff_list ff | ff

ff -> call_string : valid_at_entry_of : foreach_or_total : constraint ;

call_string -> call_string call | call |

call -> ((func , stmt) , func)

valid_at_entry_of -> func | (func , stmt)

foreach_or_total -> [ ] | < > %% Only to be used with (func, stmt) | < int .. int > %% Only to be used with (func, stmt) %% First int should be >= 1 and <= second int.

constraint -> expr < expr | expr <= expr | expr == expr | expr > expr | expr >= expr

expr -> int | mul_expr + expr | mul_expr - expr

mul_expr -> int * count_var | count_var

count_var -> stmt | stmt->stmt

int -> 'integer'

func -> 'identifier'

stmt -> 'identifier'

9. Flow Hypotheses Generation Using Traces

SWEET allows derivation of flow hypothesis using traces. When running the AE in trace mode two inputs need to given to SWEET:

1. A file holding a set of traces. Each trace is a list of nodes ending by a semi-colon. Each trace must be a path of nodes possible in the CFGs.

2. A file holding a set of CFGs. Each CFG should be a connected graph with a dedicated start node, a set of nodes, and a set of edges connecting the nodes. Each CFG can also hold a set of call edges going from a node in the CFG to a named CFG.

SWEET will collect all the traces into a set of flow hypothesis. The format of these hypotheses are exactly the same as the flow facts. However, flow hypothesis are different from flow facts since they are not guaranteed to be safe lower and upper bounds on the program's possible executions. Instead, the flow hypothesis are only valid for the given traces, holding the lower and upper bounds encountered for the traces.

Illustrating example.

Consider the following example code snippet:

```
int foo(int j) {   // BB0
  while (j < 5) {  // BB1
```

```
    if(j < 3)      // BB2
      j++;         // BB3
    else
      j=j+2;       // BB4
    j++;           // BB5
  }
  return j;        // BB6
}
```

The code corresponds to the following CFG:

```
begin CFG
 name foo ;
 startnode BB0 ;
 nodes BB0 BB1 BB2 BB3 BB4 BB5 BB6 ;
 edges BB0->BB1 BB1->BB2 BB2->BB3
    BB2->BB4 BB3->BB5 BB4->BB5 BB5->BB1 BB1->BB6 ;
 calls ;
end CFG
```

Assuming that we have the following three traces:

```
BB0 BB1 BB2 BB3 BB5 BB1 BB2 BB3 BB5 BB1 BB2 BB4 BB5 BB1 BB6 ;
BB0 BB1 BB2 BB3 BB5 BB1 BB2 BB4 BB5 BB1 BB6 ;
BB0 BB1 BB2 BB4 BB5 BB1 BB6 ;
```

SWEET can, for example, generate the following flow hypotheses:

: foo : [] : BB1 >= 2 ; %% Lower bound for BB1 loop header

: foo : [] : BB1 <= 4 ; %% Upper bound for BB1 loop header

: foo : [] : BB2 <= 3 ; %% Upper bound for BB2 node

: foo : [] : BB5->BB1 <= 3 ; %% Upper bounds for loop back-edge

Flow hypotheses generation.

The basic input options to use are:

```
sweet -i=<f.cfg> lang=cfg -ae rtf=<f.trace> -f
```

where f.cfg and f.trace are the name of the files holding the cfg graphs and traces respectively. The flow hypothesis generation works similar to the AE by recorders and collectors. This means that every flow fact generator which could be given using `-ae ffg=...` could also be used to generate flow hypotheses. Similar to flow facts, the flow hypotheses can be exported in various formats.

For example, if all type of flow hypotheses should be generated use:

```
sweet -i=<f.cfg> lang=cfg -ae rtf=<f.trace> ffg=all
```

See Section Flow Fact Language for details of SWEET's flow fact generation. To assist in the flow hypothesis generation SWEET also supports:

- printouts of CFGs from ALF programs: `-i=<f.alf> -p`
- trace generation from an AE run on ALF programs: `-ae gtf=<f.trace> -p` (requires that SWEET runs in single path mode)
- running AE on ALF program by a trace: `-i=<f.alf> -ae rtf=<f.trace> -p`

BNF grammar for trace files.

tracefile -> trace_list

trace_list -> trace_list trace | trace

trace -> node_list ';'

node_list -> node_list nodeid | nodeid

nodeid -> <string>

BNF grammar cfg for files.

cfgfile -> cfg_list

cfg_list -> cfg_list cfg | cfg

cfg -> 'begin CFG' cfgname startnode nodes edges calls 'end CFG'

cfgname -> 'name' funcname ';'

startnode -> 'startnode' nodeid ';'

nodes -> 'nodes' node_list ';'

node_list -> node_list nodeid | nodeid

edges -> 'edges' edge_list ';'

edge_list -> edge_list edge | edge |

edge -> nodeid '->' nodeid

calls -> 'calls' call_list ';'

call_list -> call_list call | call

call -> nodeid '->' funcname

funcname -> <string>

nodeid -> <string>

10. SWEET's Debug Facilities

There are useful debugging facilities available in SWEET during abstract execution. It can be used for debugging of ALF code, to follow the abstract execution in detail (including watching intermediate values of variables) and debugging SWEET itself. Abstract execution (option `-ae`) includes the parameter `debug`=<X> which turns on debugging, where <X> can be either "`inter`" which starts the interactive debugger, or "`trace`" which prints a trace to the file `debug_msgs.txt`.

Interactive debugging

Available interactive debugger commands (all commands are case-insensitive):

`Help/h` - List all available debugger commands

`Stmt/stm` - Print the current ALF statement

`BackTrace/bt` - Print out the call stack (bottom to top) including the local variables

`Show <n>/sh <n>` - Show the contents of the frame "n" in the current state and scope

`States/sts` - Print summary of current program states

`State <n>/sta <n>` - Print information about state "n"

`Step/s` - Step to the next statement

`Run/r` - Resume abstract execution of the ALF program

`BreakPt <p>/bp <p>` - Set a breakpoint at p, where p is either an ALF program line number* for the (start of an) ALF statement or an ALF statement type ("`store`", "`switch`", "`jump`", etc).

`ClearBPs/cb` - Clear all breakpoints

`StackHeight/sth` - Start monitoring call stack height

`Quit/q` - Terminate the ALF program