ICD-C Flow Facts Documentation

(v1.1 / 27.7.2010 / T. Kelter)
(v1.0 / 8.8.2008 / R. Pyka)
Partial translation of Thesis by D. Schulte, 2007 [1]

Definition of Flow Facts

Flow Facts have been defined by R. Kirner in his PhD Thesis [2] as follows:

Definition 1:

> Flow facts are meta-information which provide hints about the set of possible control flow paths of a program *P*. The corresponding set of control flow paths is denoted as $CFP_{ff}(P)$

Furthermore, Kirner defines two types of flow facts. The first type describes properties which are implicit due to the structure and semantics of the program, the second type of flow facts is explicitly annotated by the user:

Definition 2:

> Flow facts which exist due to the structure and semantics of the program code are called implicit flow facts (ff_{impl}). Flow facts which are provided by the user are called annotated flow facts (ff_a)

The option for user provided flow facts is important for a precise WCET-analysis, because the implicit flow facts usually have only a limited quality. This is due to the limitations of analysis tools and due to the fact that the control flow of a program depends on the input data, which is not considered in the computation of implicit flow facts.

There is no clear distinction between ff_{impl} and ff_a. Both sets are not required to be disjoint. The representation of flow facts in ICD-C is basically the representation of ff_a. Nevertheless the flow facts could be generated automatically.

Flow fact design decisions

This implementation of flow facts differs in three major aspects from what is known from the literature:

- Code optimizations performed in the compiler do not invalidate the flow facts.
- Flow facts have to be compatible with external tools. (i.e. aiT)
- Flow facts are tightly integrated into the abstract syntax tree representation of the code.

Basic aspects of the flow facts implementation

In the current literature, flow fact information is attached to the edges of the control flow graph. With respect to annotated flow facts, this has several disadvantages. Control flow edges are not explicitly expressed in the C program code and are therefore not obvious to the user. The natural way of annotating flow facts in a C program would be to define them in relation to program statements. Program statements correspond to a subset of program sequence points. Beside the fact that this kind of flow fact annotation is slightly less precise than annotating them to control flow edges, it has some technical advantages: Flow facts attached to statements fit well into the abstract syntax tree representation of the program code within ICD-C. External tools like aiT use the same annotation method which results in less conversion overhead. There are three types of flow facts which can by annotated in ICD-C: Flow restrictions, loop bounds and entrypoints.

Flow restrictions and Loop bounds

Flow restrictions allow to express the execution count of one statement in relation to the execution count of other statements. This annotation method would provide sufficient information to express any kind of linear dependency.

Flow restrictions are expressed in terms of inequations. Each side of an inequation is a weighted sum of execution counts of statements. The operator can be one of the following: less-then-equal, equal or greater-than-equal. The semantic of such an inequation is slightly different from what is expected from regular mathematical inequations.

Example: A flow restriction

$$1 * stmt_1 <= 15 * stmt_2$$

states that one execution of $stmt_1$ is limited by 15 times the execution count of $stmt_2$. The notion becomes more intuitive if $stmt_1$ and $stmt_2$ are not considered to be the statements themselves, but variables in an inequation system holding the actual execution count of its corresponding statement.

Besides the flow restrictions, ICD-C offers another annotation method. Loop bounds are a simplified method for loop annotation. Loop bounds provide an upper and lower bound for the number of iterations of the annotated loop.

Entrypoints

Entrypoints denote points in the CFG where the control flow may start. Typically this is the “main” function of the program, but in a (possibly interrupt-driven) multi-task system there may be multiple entrypoints in a single set of source files. These entrypoints may even share some common code. To be able to express this for the timing analyzer, each function in the ICD-C can be marked as an being an entrypoint.

Specification of flow facts in the source code

According to the previous section, flow facts attached to program statements. ICD-C uses ANSI-C pragmas for this purpose. Flow facts encapsulated into pragmas can be annotated in the source code without introducing incompatible changes to the syntax of the C program. Therefore, even flow fact annotated code can be processed by legacy compilers and tools which do not take advantage of this information.

The C standard offers two styles of pragmas:

```
#pragma ANNOTATION ↵
```

or

```
_Pragma("ANNOTATION")
```

where ANNOTATION expands as follows in the case of flow facts:

```
ANNOTATION        |=  MARKER | FLOWRESTRICTION | LOOPBOUND |
                       ENTRYPOINT
```

Besides flow restrictions and loop bounds, additional annotations are required to express flow facts in the source code. So-called markers are used to identify program statements in flow restrictions.

```
MARKER           |= marker NAME
NAME             |= Identifier
```

Each marker relates an identifier to the corresponding statement where the marker-pragma has been placed. If the pragma is found to be attached to an expression, the marker is related to the parent statement of that expression. Some restrictions apply to the place where markers can be set:

- Compound statements can not be marked. This is not a limitation, since compound statements do not match exactly the control flow graph of the program. Flow facts are more precisely expressed in relation to a single statement, which in turn have an exact matching to a corresponding basic block.
- Each function has an default marker with the function name as identifier. It gets related to the entry basic block of this function.
- Markers placed at loop statements are related to their conditional expressions. Therefore, unconditional loops can not be marked that way. Nevertheless, the loop body statements can be marked.

The identifier name is an arbitrary alphanumerical string. Internally, a reference to the actual statement is used instead of the string. This saves effort while relating to that marker. From the pure semantical point of view, the marker represents a decision variable which provides information about the execution counts of that statement on a control path. The set of values is restricted in loop bounds and flow restrictions which define that decision variable.

Flow restrictions

Flow restrictions allow to define execution counts of statements in relation to execution counts of other statements.

```
FLOWRESTRICTION  |=  flowrestriction SIDE COMPARATOR SIDE
COMPARATOR       |=  >=  | <= | =
SIDE             |= SIDE + SIDE | NUM * REFERENCE
NUM              |= Non-negative-Integer
REFERENCE        |= Identifier | Function-name
```

References to statements are expressed in terms of identifiers, where the identifier is equal to the identifier in the marker referencing the statement. Function names are implicitly defined and do not require any marker.

Example: Flow restrictions can be used to express precise upper execution bounds for statements located inside nested loops. Furthermore, multi-entry loops, goto-loops etc. can also be annotated. The basic approach to define maximum iteration counts for loops is to mark a statement outside the loop nest and one inside. The inner statement has to be executed in each loop iteration, regardless of any other condition. Afterwards, a flow restriction has to be defined which limits the maximum execution count of the inner statement.

“`flowrestriction 1 * `*`inner-marker`*` <= max * `*`outer-marker`*”

Where 'max' is the actual maximal execution count of the inner statement.

A simple example showing the use of flow restrictions and markers in that example scenario is shown in the following listing:

```
_Pragma( "marker outer-marker" )
Stmt-A;
for ( i = 0 ; i < 10 ; i++ )
     for ( j = i ; j < 10 ; j++ )
          _Pragma( "marker inner-marker" )
          Stmt-B;

_Pragma( "flowrestriction 1*inner-marker <= 55*outer-marker" )
```

As the flow restriction states, Stmt-B is being executed no more than 55 times. This is more precise than what could be annotated using loop bounds, since loop bounds can only define the bounds per loop, therefore booth loops have to be annotated with their maximum iteration counts separately, which is 10 in both cases. Using a loop bound annotation, only a less precise maximum execution count for Stmt-B of no more than 100 executions can be deduced.

In a similar way the maximum recursion depth of recursive functions can be limited. In that case, the call-expression statement has to be marked (i.e. 'call-marker'). The second marker is not required, because the reference points to the function entry which has an implicit marker. A flow restriction limits the recursion depth in a similar way as for a loop:

“`flowrestriction 1 *` *`recfunc`* `<= max *` *`call-marker`*”

Flow restrictions are a powerful description method. They allow the description of even more exotic dependencies like mutually exclusive execution of statements. Nevertheless, it is not possible to express deeply nested dependencies as shown in the following example.

```
if ( cond )
     x = true ; // Stmt-A
for ( ... )
     if ( x )  Stmt-B;
```

The execution of Stmt-A implies the execution of statement Stmt-B which can't be expressed with flow restrictions.

Loop bounds

Loop bounds allow to specify the minimum and maximum iteration counts for each loop.

```
LOOPBOUND          |= loopbound min NUM max NUM
NUM                |= Non-negative-Integer
```

It is not required to mark (loop-)statements for loop bound annotation, since each loop bound applies to exactly that statement where the pragma has been defined. Loop bound pragmas can be annotated only to for-, while- and do-while-statements. Other loop types (i. e. goto-loops) or loops without an exit condition can only be annotated using flow restrictions.

In the case WCET estimation has to be performed, some additional limitations exist. Loop bounds translate directly to aiT's loop bounds. Therefore a particular loop has to be recognizable by aiT. Best practice is to annotate only loops which are known to be recognized by aiT. As a general rule, any simple for- while- or do-while-loop with exit condition are recognized in aiT. On the other side, multi-entry loops, loops with labels, case- or default statements in the loop body should not be annotated, they are likely to be not recognized. Nevertheless flow restrictions can be used instead.

Entrypoints

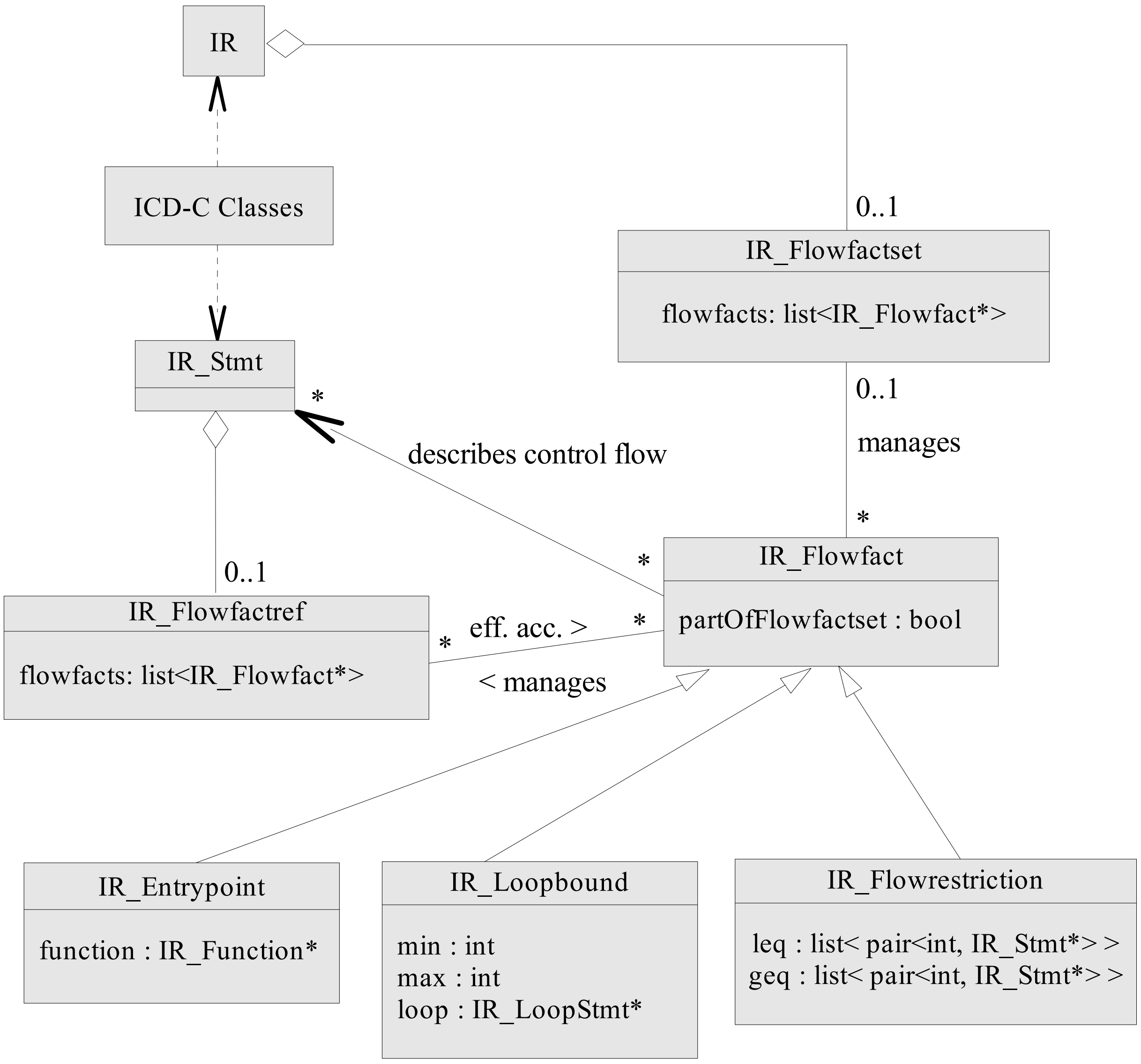
Entrypoints allow to specify that the control flow may initially enter the code at the marked function. If no entrypoint annotation is given, the “main” function implicitly becomes the only entrypoint.

```
ENTRYPOINT         |= entrypoint
```

The annotation must be put in front of the name of the function which is to be marked, just after the return value. Both, the declaration and the definition of the function can be marked, which has the same effect.

Data structures

Flow facts are represented in ICD-C as additional data structures. They are tightly integrated into the abstract syntax tree. The following picture shows the dependencies between object classes and their integration into ICD-C.



A good programing style encapsulates all class data members. Accordingly, each class provides get() and set() methods for their data members. Flow fact information is integrated into the ICD-C abstract syntax tree as user-data objects. For statements, the class IR_Flowfactref encapsulates references to flow facts relevant to this statement, additionally at IR level a set of all flow facts is kept in an object of class IR_Flowfactset. The user-data objects have a common ID “flowfacts”. Flow fact classes are derived from IR_AttachableObject which ensures automatic destruction after all references to such an object are removed.

Besides the container classes, the actual flow facts are constructed by inheriting from class IR_Flowfact. This allows the definition of a common set of methods which has to be provided by any flow fact, additionally this keeps the approach extensible to any kind of additional flow facts (i.e. other control flow related information).

Currently, two types of flow facts are known: IR_Loopbound and IR_Flowrestriction. Class IR_Loopbound keeps two integer values denoting the minimum and maximum iteration count, and a pointer to the loop statement this flow fact belongs to. Iteration counts may not be negative. As a special case, both values may be set to -1 which indicates the presence of a loop without specifying any loop bound.

IR_Flowrestriction keeps the data in a normalized form of the flow restriction inequation. There are two lists, one for the left-hand side and one for the right-hand side of the inequation. Each entry in the list consists of a constant factor and a reference to a statement. All members of the left- or right-hand side list are summed up to form the expression for the corresponding side of the inequation. The normalized operator is the less-or-equal operator. Some restrictions apply to the content of the list:

- Factors have to be non-negative
- Each statement may be referenced only once
- If a factor becomes null, the whole sum will be simplified to null

There are well defined access methods addToLeq(int factor, IR_Stmt *stmt) and addToGeq(int factor, IR_Stmt *stmt) which ensure the enforcement of these restrictions.

Special caution has to be taken when removing ICD-C objects from the IR which participate in the flow fact definition. Before such an object can be deleted safely, it may not be referenced in any flow fact. It is in the user's responsibility to ensure this, since ICD-C does not provide any notification methods.

The common access pattern is to iterate over all flow facts. This is supported by the IR_Flowfactset container attached to the IR. Each flow fact is stored in this set. For the standard application scenario, this centralized data repository turned out to be the most efficient way. On the one side, the transformation of flow facts between different abstraction levels requires a global view of all flow facts. Furthermore, due to the deeply referenced format of the flow fact representation, in some cases there is no single point in the ICD-C abstract syntax tree where this flow restriction fits best. On the other side, the centralized data repository prevents efficient access to flow facts which are related to a particular statement. This would result in a regular search iterating over all flow facts, which in turn would degrade the performance significantly.

To avoid inefficient data access, additional data structures have been introduced. Each IR_Stmt object which participates in any flow fact definition, has an IR_Flowfactref object in its user data. This object encapsulates a list of flow facts which are related to this statement. These additional references are kept synchronized automatically. Whenever a new flow fact is added or an existing one is modified, the references are updated accordingly. The same applies if a flow fact is removed, every reference is removed, too.

Each flow fact incorporates a flag which marks it as part of a flow fact set. The flag is set when a flow fact is added to a set. This information is used to check whether references have to be updated on a modification of that flow fact object. Additionally, the first time a flow fact is inserted into a set, all references are initialized according to the content.

Output method

Sometimes it is useful to print out a list of all flow facts. Each class implements a write(ostream &str, int indent = 0) method. The write method of the IR-related IR_Flowfactset prints all the flow facts, while the write method in IR_Flowfactref prints only the flow facts related to that particular statement.

References

[1] Schulte, Daniel: *Modellierung und Transformation von Flow Facts in einem WCET-optimierenden Compiler.* Dortmund, Technische Universität Dortmund, Dipl. Thesis, Mai 2007

[2] Kirner, Raimund: *Extending Optimising Compilation to Support Worst-Case Execution Time Analysis*. Wien, Technische Universität Wien, PhD, Mai 2003

IR_Flowfact Class Reference

```
#include <irflowfact.h>
```

Detailed Description

Abstract base class for different kind of Flow Facts.

This class provides a minimum set of functions, every Flow Fact has to support. Every kind of Flow Facts has to implement these functions to guarantee a common management by the Flowfactset and Flowfactref-mechanisms.

Definition at line 41 of file irflowfact.h.

Public Member Functions

- **IR_Flowfact** ()
  Default contructor.
- virtual **~IR_Flowfact** ()
  Default destructor.
- virtual void **write** (ostream &str, int indent=0) const =0
  Writes the Flow Fact.
- virtual bool **isPartOfFlowfactset** ()
  Determines, whether this Flow Fact belongs to the Flowfactset.
- virtual void **setPartOfFlowfactset** (bool)=0
  Specifies, whether this Flow Fact belongs to the Flowfactset.
- virtual bool **isContained** (**IR_Stmt** *)=0
  Determines, whether the hole Flow Fact is contained in the specified Statement.
- virtual bool **isSignificant** ()
  Indicates, whether a Flow Fact is significant for WCET-calculation.

Protected Member Functions

- virtual void **addRefToStmt** (**IR_Stmt** *)=0
  Adds Flow Fact to Flowfactref.
- virtual void **removeRefFromStmt** (**IR_Stmt** *)=0
  Removes Flow Fact from Flowfactref.

Protected Attributes

- bool **partOfFlowfactset**
  Indicates, whether this is part or not of IR's Flowfactset.

Constructor & Destructor Documentation

IR_Flowfact::IR_Flowfact ()

Default contructor.

virtual IR_Flowfact::~IR_Flowfact () `[virtual]`

Default destructor.

Member Function Documentation

virtual void IR_Flowfact::write (ostream & *str*, int *indent* = 0) const `[pure virtual]`

Writes the Flow Fact.

This function is used to write the Flow Fact to Logfiles,... The output has to be indented by the numbers of space characters specified by indent.

Implemented in **IR_Flowrestriction** (*p.24*), and **IR_Loopbound** (*p.28*).

virtual bool IR_Flowfact::isPartOfFlowfactset () `[virtual]`

Determines, whether this Flow Fact belongs to the Flowfactset.

This function returns the partOfFlowfactset-Bool. If a Flowfactset holds this Flow Fact, the Bool is set to true, otherwise false.

virtual void IR_Flowfact::setPartOfFlowfactset (bool) `[pure virtual]`

Specifies, whether this Flow Fact belongs to the Flowfactset.

This function sets the partOfFlowfactset-Bool. If a Flowfactset holds this Flow Fact, the Bool has to be true, otherwise false.

This function should be used carefully, because the automatic management of Flowfactrefs is influenced by this. If this Bool does not correspond with the state of the Flow Fact, inconstent datastructures can be generated.

Implemented in **IR_Flowrestriction** (*p.26*), and **IR_Loopbound** (*p.29*).

virtual bool IR_Flowfact::isContained (IR_Stmt *) `[pure virtual]`

Determines, whether the whole Flow Fact is contained in the specified Statement.

If every element of a Flow Fact is contained in the specified Statement and its substatement hierarchy, then the function returns true, otherwise (at least one element of the Flow Fact is not part of the specified Statement and isr substatementhierarchy) false.

This information is for example used to decide, how Flow Facts (in particular the Flowrestrictions) have to be handled by the Flowfactupdater after the copy of statements.

Implemented in **IR_Flowrestriction** (*p.26*), and **IR_Loopbound** (*p.29*).

virtual bool IR_Flowfact::isSignificant () `[virtual]`

Indicates, whether a Flow Fact is significant for WCET-calculation.

In some cases, a Flow Fact may not be significant for WCET-calculation, for example:

- A Loopbound with min: -1 and max: -1
- A Flowrestriction with 0 <= SUM. In such cases, this function should return false, in all other cases true. This could be used to remove all not necessary Flow Facts.

virtual void IR_Flowfact::addRefToStmt (IR_Stmt *) `[protected, pure virtual]`

Adds Flow Fact to Flowfactref.

The Flowfactref of the given Statement is told, that this Flow Fact is using its Statement (so that that Flowfactref has to hold a reference to this Flow Fact), but only, when this Flow Fact is part of a flowfactset. If no Flowfactref already exists, a new one (for the given statement) will be created.

The individual definition of this function allows to build Flow Facts, which are not back-referenced by the Flowfactrefs.

Implemented in **IR_Flowrestriction** (*p.26*), and **IR_Loopbound** (*p.29*).

virtual void IR_Flowfact::removeRefFromStmt (IR_Stmt *) `[protected, pure virtual]`

Removes Flow Fact from Flowfactref.

The Flowfactref of the given Statement is told, that this Flow Fact is not longer using its Statement, but only, when this Flow Fact is part of a flowfactset.

The individual definition of this function allows to build Flow Facts, which are not back-referenced by the Flowfactrefs.

Implemented in **IR_Flowrestriction** (*p.26*), and **IR_Loopbound** (*p.30*).

Field Documentation

bool IR_Flowfact::partOfFlowfactset `[protected]`

Indicates, whether this is part or not of IR's Flowfactset.

This is important for decisions whether a reference to a **IR_Stmt** has to be updated or not.

Definition at line 133 of file irflowfact.h.

IR_Flowfactref Class Reference

```
#include <irflowfactref.h>
```

Detailed Description

Class holding all Flow Facts this stmt is referenced by.

This class provides a comfortable access to the Flow Facts, which concern the Statement, the Flowfactref is build for.

The data are managed by the Flow Facts (and Flowfactset) automatically.

Its part of the ICD-C datastructure and can be found as userdata with id "flowfactref".

Definition at line 51 of file irflowfactref.h.

Public Member Functions

- **IR_Flowfactref** (**IR_Stmt** *)
  Default constructor.
- **~IR_Flowfactref** ()
  Default destructor.
- list< **IR_Flowfact** * > **getFlowfacts** () const
  Returns a List of all Flowfacts which belongs to this Statement.
- list< **IR_Loopbound** * > **getLoopbounds** () const
  Returns a List of all Loopbounds which belongs to this Statement.
- list< **IR_Flowrestriction** * > **getFlowrestrictions** () const
  Returns a List of all Flowrestrictions which belongs to this Statement.
- void **write** (ostream &, int=0) const
  Writes the Flow Fact.
- **IR_Stmt** * **getStmt** () const
  This function sets the statement this object is attached to.
- void **setStmt** (**IR_Stmt** *)
  This function returns the statement this object is attached to.

Protected Member Functions

- void **addFlowfact** (**IR_Flowfact** *)
  Adds a reference to a Flow Facts.
- void **removeFlowfact** (**IR_Flowfact** *)
  Removes a reference to a Flow Facts.

Friends

- class **IR_Flowfact**
- class **IR_Loopbound**
- class **IR_Flowrestriction**
- class **IR_Flowfactset**

Constructor & Destructor Documentation

IR_Flowfactref::IR_Flowfactref (IR_Stmt *)

Default constructor.

Creates an empty Flowfactref for the given statement (is automatically added as Userdata to this ICD-C Statement).

IR_Flowfactref::~IR_Flowfactref ()

Default destructor.

Member Function Documentation

list<IR_Flowfact *> IR_Flowfactref::getFlowfacts () const

Returns a List of all Flowfacts which belongs to this Statement.

Only Flow Facts which are part of the Flowfactset are returned. The list is only a copy to hide the internal datastructure.

list<IR_Loopbound *> IR_Flowfactref::getLoopbounds () const

Returns a List of all Loopbounds which belongs to this Statement.

Only Loopbounds which are part of the Flowfactset are returned. The list is only a copy to hide the internal datastructure.

list<IR_Flowrestriction *> IR_Flowfactref::getFlowrestrictions () const

Returns a List of all Flowrestrictions which belongs to this Statement.

Only Flowrestrictions which are part of the Flowfactset are returned. The list is only a copy to hide the internal datastructure.

void IR_Flowfactref::write (ostream &, int = `0`) const

Writes the Flow Fact.

This function is used to write the Flow Fact to Logfiles,... The output has to be indented by the numbers of space characters specified by indent.

All Flow Facts concerning the statement this flowfactref is build for are written by their write-methode.

IR_Stmt* IR_Flowfactref::getStmt () const

This function sets the statement this object is attached to.

void IR_Flowfactref::setStmt (IR_Stmt *)

This function returns the statement this object is attached to.

void IR_Flowfactref::addFlowfact (IR_Flowfact *) `[protected]`

Adds a reference to a Flow Facts.

This method is used by the Flow Facts to tell the Flowfactref, that they concern it and are part of the

Flowfactset.

void IR_Flowfactref::removeFlowfact (IR_Flowfact *) `[protected]`

Removes a reference to a Flow Facts.

This method is used by the Flow Facts to tell the Flowfactref, that they does not concern it any longer or are removed from the Flowfactset.

Friends And Related Function Documentation

friend class IR_Flowfact `[friend]`

Definition at line 54 of file irflowfactref.h.

friend class IR_Loopbound `[friend]`

Definition at line 55 of file irflowfactref.h.

friend class IR_Flowrestriction `[friend]`

Definition at line 56 of file irflowfactref.h.

friend class IR_Flowfactset `[friend]`

Definition at line 57 of file irflowfactref.h.

IR_Flowfactset Class Reference

```
#include <irflowfactset.h>
```

Detailed Description

Class holding all Flow Facts of the **IR**.

All Flow Facts for WCET-Analysis are collected and handled by this class.

It is added to the **IR** as user data (identification: "flowfacts").

Definition at line 47 of file irflowfactset.h.

Public Member Functions

- **IR_Flowfactset** (**IR** *)
  Default constructor.
- **~IR_Flowfactset** ()
  Default destructor.
- void **addFlowfact** (**IR_Flowfact** *)
  Add a Flow Fact for WCET Analysis.
- void **removeFlowfact** (**IR_Flowfact** *)
  Removing a Flow Fact from WCET Analysis.
- void **symbolicUpdate** (**IR_Stmt** *)
  Try transitive enlargement for the given Statement.
- list< **IR_Loopbound** * > **getLoopbounds** () const
  Returns a List of all Loopbounds.
- list< **IR_Flowrestriction** * > **getFlowrestrictions** () const
  Returns a List of all Flowrestrictions.
- list< **IR_EntryPoint** * > **getEntryPoints** () const
  Returns a List of all Entrypoints.
- bool **correctCopyOfStmt** (**IR_Stmt** *, **IR_Stmt** *)
  Correction of Flow Facts after copy of statements.
- void **minimizeFlowfacts** (int)
  Removes not necessary Flow Facts.
- void **write** (ostream &, int=0) const
  Writes the Flow Fact.

Constructor & Destructor Documentation

IR_Flowfactset::IR_Flowfactset (IR *)

Default constructor.

Creates an empty Flowfactset for the given **IR** (is automatically added as Userdata to this ICD-C **IR**).

IR_Flowfactset::~IR_Flowfactset ()

Default destructor.

Member Function Documentation

void IR_Flowfactset::addFlowfact (IR_Flowfact *)

Add a Flow Fact for WCET Analysis.

By adding a Flow Fact, this Flow Fact is available for automatic translations and updates of Flow Facts

By setting the partOfFlowfactset-Bool to true, the automized administration of Flowfactrefs is invoked.

void IR_Flowfactset::removeFlowfact (IR_Flowfact *)

Removing a Flow Fact from WCET Analysis.

By removing a Flow Fact, this Flow Fact is not longer available for automatic translations and updates of Flow Facts, but is not destroyed by this.

But by setting the partOfFlowfactset-Bool to false, the automized administration of Flowfactrefs is disabled.

After removing a Flow Facts, the Flow Fact could be destroid without errors.

void IR_Flowfactset::symbolicUpdate (IR_Stmt *)

Try transitive enlargement for the given Statement.

If two Flowrestrictions A <= B and B <= C are part of this Flowfactset, the transitive information A <= C is implicit specified. By updating Flowrestrictions, this information could be destroyed (A <= B and B <= C may become A <= higher_exec_than_B and lower_exec_than_B <= C).

To keep this information, this function will add all implicit transitive informations generated by the specified Statement explicitely.

Only Flowrestrictions are involved.

list<IR_Loopbound *> IR_Flowfactset::getLoopbounds () const

Returns a List of all Loopbounds.

Only Loopbounds which are part of the Flowfactset are returned. The list is only a copy to hide the internal datastructure.

list<IR_Flowrestriction *> IR_Flowfactset::getFlowrestrictions () const

Returns a List of all Flowrestrictions.

Only Flowrestrictions which are part of the Flowfactset are returned. The list is only a copy to hide the internal datastructure.

list<IR_EntryPoint *> IR_Flowfactset::getEntryPoints () const

Returns a List of all Entrypoints.

Only Entrypoints which are part of the Flowfactset are returned. The list is only a copy to hide the internal datastructure.

bool IR_Flowfactset::correctCopyOfStmt (IR_Stmt *, IR_Stmt *)

Correction of Flow Facts after copy of statements.

To copy a stmt means for user data, that the reference is copied to the new statement. Because of the automatic handling of **IR_Flowfactref**, this behaviour is undesired. So this method will correct that by following:

- remove all copied references to make data consistent
- Loopbounds will be copied to new stmts
- Flowrestrictions will be copied to new restrictions if all parts are part of the new stmt-hierarchy otherwise be updated The last step demands a request of this method on top-level stmts, the copy function was used for.

Use: first parameter: old stmt; second one: new corresponding(!) stmt

Returns true, if correction was successful, otherwise false.

void IR_Flowfactset::minimizeFlowfacts (int)

Removes not necessary Flow Facts.

The Integer defines the Minimization-Level. Removes following Flow Facts:

- Flow Facts with no significance (from 0 on)
- identical Flow Facts (planned, from 1 on)

void IR_Flowfactset::write (ostream &,   int = 0) const

Writes the Flow Fact.

This function is used to write the Flow Fact to Logfiles,... The output has to be indented by the numbers of space characters specified by indent.

All Flow Facts are written by their write-methode.

IR_Flowfactupdater Class Reference

```
#include <irflowfactupdater.h>
```

Detailed Description

Provides Update mechanisms for ICD-C's Flow Facts.

The Flow Facts were added to the **IR** by the FlowfactmanagerCtoICDC.

Because of their strong dependence to the control flow, they have to be updated every time the (control flow of the) program may be changed. For this, the **IR_Flowfactupdater** provides some methods which have to be used in the optimizations. The programmer of the optimizations is responsible for the correct use of the update mechanism.

Please keep in mind: a wrong or a forgotten update of the Flow Facts could result in a too small WCET without any warning or hint!

Definition at line 71 of file irflowfactupdater.h.

Public Member Functions

- **IR_Flowfactupdater** ()
  Default constructor.

Protected Types

- enum **flowfactsmode** { **leq** = 1, **equ** = 2, **geq** = 3 }
  Defines the types of substitions.

Protected Member Functions

- bool **flowfactsPrepareUpdate** (**IR_Stmt** *)
  Flowrestrictionupdate: Creates data structure for update.
- void **flowfactsAddSingle** (**IR_Stmt** *, **IR_Stmt** *, **flowfactsmode**)
  Flowrestrictionupdate: Collecting informations for update.
- void **flowfactsAddSingle** (**IR_Stmt** *, **IR_BasicBlock** *=0, **flowfactsmode**=equ)
  Flowrestrictionupdate: Collecting informations for update.
- void **flowfactsAdd** (**IR_Stmt** *, **IR_Stmt** *, **flowfactsmode**)
  Flowrestrictionupdate: Collecting informations for update.
- void **flowfactsAdd** (**IR_Stmt** *, int, **IR_Stmt** *, **flowfactsmode**)
  Flowrestrictionupdate: Collecting informations for update.
- void **flowfactsAdd** (**IR_Stmt** *, list< pair< int, **IR_Stmt** * > >, **flowfactsmode**)
  Flowrestrictionupdate: Collecting informations for update.
- void **flowfactsSearchUpdate** (**IR_Stmt** *)
  Flowrestrictionupdate: Try to collect available informations automatically.
- void **flowfactsPreventSearchUpdate** (**IR_Stmt** *)
  Flowrestrictionupdate: Disables the automatic try of information collection.
- void **flowfactsDoUpdate** (**IR_Stmt** *)
  Flowrestrictionupdate: Use collected data and do the update.

- void **flowfactsDoUpdate** (**IR_Stmt** *, int, **IR_Stmt** *)
  Flowrestrictionupdate: Prepare and perform update in once.
- void **flowfactsRemoveUnreachableStmt** (**IR_Stmt** *)
  Performs update of Flow Facts for statements which are never executed.
- void **flowfactsRemoveUnreachableStmt** (set< **IR_Stmt** * >)
  Performs update of Flow Facts for statements which are never executed.
- void **flowfactsRemoveUnreachableFunc** (**IR_Function** &)
  Performs update of Flow Facts for statements which are never executed.
- void **flowfactsDeleteFlowrestrictions** (**IR_Stmt** *)
  Deletes all Flowrestrictions for a Stmt.
- void **flowfactsMoveLoopbounds** (**IR_LoopStmt** *, **IR_LoopStmt** *)
  Moves all Loop bounds from first given loop to second one.
- void **flowfactsCreateLoopbound** (**IR_LoopStmt** *, int, int)
  Creates a new loop bound.
- void **flowfactsDeleteLoopbound** (**IR_LoopStmt** *)
  Deletes all Loop bounds from given loop.
- pair< int, int > **flowfactsDetermineNewNumbersOfIterations** (**IR_LoopStmt** *, **IR_LoopAnalyzer** *, int=1)
  Merges different iteration information about a loop.
- void **flowfactsCorrectCopyOfStmt** (**IR_Stmt** *, **IR_Stmt** *)
  Correct data inconsistence in consequence of copy stmt.
- void **flowfactsUnswitchLoop** (**IR_LoopStmt** *, **IR_SelectionStmt** *)
  Update after unswitching of loop (only for Flowrestictions!).
- void **flowfactsLog** (string)
  Writes a message to the logfile specified by FLOWFACTLOG.
- void **printFlowfactsToLog** (**IR_Stmt** *)
  Writes the Flow Fact.

Static Protected Member Functions

- static void **flowfactsStaticExchange** (**IR_Stmt** *, **IR_Stmt** *)
  Exchanges Statements in Flowrestrictions.

Data Structures

- class **IR_Updatedata**
  Data container for managing updates.

Member Enumeration Documentation

enum IR_Flowfactupdater::flowfactsmode `[protected]`

Defines the types of substitions.

To add possible alternatives for substitution of statements to the right container, flowfactsmode gives information about the exec. frequency of the current alternative:

- leq: destination name for alternatives with a less equal execution frequency

- equ: destination name for alternatives with a equal execution frequency
- geq: destination name for alternatives with a greater equal execution frequency

Enumerator:

leq
equ
geq

Definition at line 93 of file irflowfactupdater.h.

```
93 { leq = 1, equ = 2, geq = 3 };
```

Constructor & Destructor Documentation

IR_Flowfactupdater::IR_Flowfactupdater ()

Default constructor.

Member Function Documentation

bool IR_Flowfactupdater::flowfactsPrepareUpdate (IR_Stmt *) `[protected]`

Flowrestrictionupdate: Creates data structure for update.

Update of Flowrestriction: 1. Step: Prepare Update

If at least one Flowrestriction is influenced by the specified statement (i.e. the Flowrestriction references the statement), a data container for collecting informations for an update is created. Otherwise this step (and all following steps because of an absent data container) is aborted.

Returns true, if a Flowrestriction with reference to this stmt exists (and an update will be necessary).

void IR_Flowfactupdater::flowfactsAddSingle (IR_Stmt *, IR_Stmt *, flowfactsmode) `[protected]`

Flowrestrictionupdate: Collecting informations for update.

Update of Flowrestriction: 2. Step: Collect informations

Parameters:

- first Statement: Identifies the Statement for which the update is specified
- second Statement: Prove succs and preds of this statement to find another one with the same (in some cases also a greater or a smaller) execution frequency.
- flowfactsmode: Specifies, whether the second statement is executed as often as the first one (equ), more often (geq) or less often (leq).

The result of this search is saved in the data container specified by the first statement.

This method is not forced to end in finding a possible update.

void IR_Flowfactupdater::flowfactsAddSingle (IR_Stmt *, IR_BasicBlock * = `0`, flowfactsmode = `equ`) `[protected]`

Flowrestrictionupdate: Collecting informations for update.

Update of Flowrestriction: 2. Step: Collect informations

Parameters:

- Statement: Identifies the Statement for which the update is specified
- Basic block: Search within this basic block for another statement. By default, the basic block of the specified statement is used (equ).
- flowfactsmode: Specifies, whether the statements of the specified basic block are executed as often as the specified statement (equ), more often (geq) or less often (leq).

The result of this search is saved in the data container specified by the first statement.

This method is not forced to end in finding a possible update.

void IR_Flowfactupdater::flowfactsAdd (IR_Stmt *, IR_Stmt *, flowfactsmode) `[protected]`

Flowrestrictionupdate: Collecting informations for update.

Update of Flowrestriction: 2. Step: Collect informations

Parameters:

- first Statement: Identifies the Statement for which the update is specified
- second Statement: Statement which should be used instead
- flowfactsmode: Specifies, whether the second statement is executed as often as the first one (equ), more often (geq) or less often (leq).

(Short for flowfactsAdd(firstArg, 1, secondArg, thirdArg);

The result of this search is saved in the data container specified by the first statement.

void IR_Flowfactupdater::flowfactsAdd (IR_Stmt *, int, IR_Stmt *, flowfactsmode) `[protected]`

Flowrestrictionupdate: Collecting informations for update.

Update of Flowrestriction: 2. Step: Collect informations

Parameters:

- first Statement: Identifies the Statement for which the update is specified
- Integer: Factor for the second statement
- second Statement: Statement which should be used instead
- flowfactsmode: Specifies, whether the second statement (multiplied with the integer) is executed as often as the first one (equ), more often (geq) or less often (leq).

Example: flowfactsAdd(id_stmt, m, another_stmt, equ) A summand à la n * id_stmt may become m * n * another_stmt.

The result of this search is saved in the data container specified by the first statement.

void IR_Flowfactupdater::flowfactsAdd (IR_Stmt *, list< pair< int, IR_Stmt * > >, flowfactsmode) `[protected]`

Flowrestrictionupdate: Collecting informations for update.

Update of Flowrestriction: 2. Step: Collect informations

Parameters:

- Statement: Identifies the Statement for which the update is specified
- list of pairs: sum of products of factors and statements which should be used instead
- flowfactsmode: Specifies, whether the sum is executed as often as the first one (equ), more often (geq) or less often (leq).

The result of this search is saved in the data container specified by the first statement.

void IR_Flowfactupdater::flowfactsSearchUpdate (IR_Stmt *) `[protected]`

Flowrestrictionupdate: Try to collect available informations automatically.

Update of Flowrestriction: 3. Step: Autocollection of informations

void IR_Flowfactupdater::flowfactsPreventSearchUpdate (IR_Stmt *) `[protected]`

Flowrestrictionupdate: Disables the automatic try of information collection.

Update of Flowrestriction: 3. Step: Autocollection of informations

If no precise Update is possible and no **flowfactsSearchUpdate(IR_Stmt*)** for this Statement was tried before, the flowfactsDoUpdate will perform a **flowfactsSearchUpdate()** to find a precise one. In some cases, this behaviour is not desired and could be prevented for a Statement by this methode.

void IR_Flowfactupdater::flowfactsDoUpdate (IR_Stmt *) `[protected]`

Flowrestrictionupdate: Use collected data and do the update.

Update of Flowrestriction: 4. Step: Perform update

The update (for the specified statement) will be performed using the collected data for this.

void IR_Flowfactupdater::flowfactsDoUpdate (IR_Stmt *, int, IR_Stmt *) `[protected]`

Flowrestrictionupdate: Prepare and perform update in once.

Update of Flowrestriction: 1.- 4. Step in once

This is short for:

- flowfactsPrepareUpdate(firstArg);
- flowfactsAdd(firstArg, secondArg, thirdArg, equ);
- flowfactsDoUpdate(firstArg);

void IR_Flowfactupdater::flowfactsRemoveUnreachableStmt (IR_Stmt *) `[protected]`

Performs update of Flow Facts for statements which are never executed.

void IR_Flowfactupdater::flowfactsRemoveUnreachableStmt (set< IR_Stmt * >) `[protected]`

Performs update of Flow Facts for statements which are never executed.

void IR_Flowfactupdater::flowfactsRemoveUnreachableFunc (IR_Function &) `[protected]`

Performs update of Flow Facts for statements which are never executed.

void IR_Flowfactupdater::flowfactsDeleteFlowrestrictions (IR_Stmt *) `[protected]`

Deletes all Flowrestrictions for a Stmt.

Every Flowrestriction, which references the specified Statement, will be deleted.

This may be useful, if no update is available. It may result in a infeasible WCET-calculation, but this update is in every case safe!

void IR_Flowfactupdater::flowfactsMoveLoopbounds (IR_LoopStmt *, IR_LoopStmt *) `[protected]`

Moves all Loop bounds from first given loop to second one.

All Loop bounds (as a rule, max. one loop bound should be specified for a loop statement) of a Loop specified by the first loop statement will be moved to the loop specified by the second statement.

void IR_Flowfactupdater::flowfactsCreateLoopbound (IR_LoopStmt *, int, int) `[protected]`

Creates a new loop bound.

For the specified Loop Statement, a new loop bound will be created and automatically added to the Flowfactset. The first Integer specifies the minimum number of iteration, the second one the maximum.

WARNING: Only one loop bound per loop statement is allowed. If you are not sure whether a loopbound already exists, use **flowfactsDeleteLoopbound()** to guarantee an unique loopbound.

void IR_Flowfactupdater::flowfactsDeleteLoopbound (IR_LoopStmt *) `[protected]`

Deletes all Loop bounds from given loop.

pair<int, int> IR_Flowfactupdater::flowfactsDetermineNewNumbersOfIterations (IR_LoopStmt *, IR_LoopAnalyzer *, int = 1) `[protected]`

Merges different iteration information about a loop.

This method allows to merge automatically between different iteration frequency informations of a loop (at this time from ICD-C Loop Analyzer and a loop bound), and returns the calculated min and max (as a pair).

For support of loop unrolling, a unroll factor could be specified. Min will be rounded down, max rounded up.

static void IR_Flowfactupdater::flowfactsStaticExchange (IR_Stmt *, IR_Stmt *) `[static, protected]`

Exchanges Statements in Flowrestrictions.

Every occurrence of the first statement in Flowrestrictions will be exchanged by the second one.

IR_Transform provides static methods, and by this the Updater has to be static, too. This methods provides a equivalent exchange and should not be used by other classes.

void IR_Flowfactupdater::flowfactsCorrectCopyOfStmt (IR_Stmt *, IR_Stmt *) `[protected]`

Correct data inconsistence in consequence of copy stmt.

Copying a Statement will cause data inconsistencies relating to Flow Facts. This method will fix this.

The origin statement is the first, the copy of the statement is the second argument.

void IR_Flowfactupdater::flowfactsUnswitchLoop (IR_LoopStmt *, IR_SelectionStmt *) `[protected]`

Update after unswitching of loop (only for Flowrestictions!).

First attempt to develop a closer update for loop unswitching.

void IR_Flowfactupdater::flowfactsLog (string) `[protected]`

Writes a message to the logfile specified by FLOWFACTLOG.

void IR_Flowfactupdater::printFlowfactsToLog (IR_Stmt *) `[protected]`

Writes the Flow Fact.

This function is used to write the Flow Fact to Logfiles,...

IR_Flowrestriction Class Reference

```
#include <irflowrestriction.h>
```

Detailed Description

Description of infeasible paths by inequalities.

A Flowrestriction is an inequalitiy, for example:

3*Codeblock1 + 2*Codeblock2 <= 6*Codeblock3

It consists of a greater-equal and a less-equal side. A restriction holds informations about the execution frequency of Codeblocks (Flow Facts).

Definition at line 48 of file irflowrestriction.h.

Public Member Functions

- **IR_Flowrestriction** ()
  Default constructor.
- **IR_Flowrestriction** (list< pair< int, **IR_Stmt** * > >, list< pair< int, **IR_Stmt** * > >)
  Constructor.
- virtual **~IR_Flowrestriction** ()
  default destructor
- virtual void **write** (ostream &, int=0) const
  Writes the Flow Fact.
- void **addToLeq** (int, **IR_Stmt** *)
  Adds a summand to Flowrestrictions less equal side.
- void **addToGeq** (int, **IR_Stmt** *)
  Adds a summand to Flowrestrictions greater equal side.
- int **removeSummand** (**IR_Stmt** *)
  Removes a summand from Flowrestriction.
- void **exchangeSummand** (**IR_Stmt** *, **IR_Stmt** *)
  Exchanges a summand in a Flowrestriction.
- list< pair< int, **IR_Stmt** * > > **getLeq** () const
  Returns a copy of the leq-side.
- list< pair< int, **IR_Stmt** * > > **getGeq** () const
  Returns a copy of the geq-side.
- bool **isPartOfLeq** (**IR_Stmt** *) const
  Tests, whether a Statement belongs to the leq side of a Flowrestriction.
- bool **isPartOfGeq** (**IR_Stmt** *) const
  Tests, whether a Statement belongs to the geq side of a Flowrestriction.
- bool **isPartOfThis** (**IR_Stmt** *) const
  Tests, whether a Statement belongs to this Flowrestriction.
- virtual bool **isContained** (**IR_Stmt** *)
  Tests, whether the entire Flow Fact is contained in the specified Statement.

- bool **isPartOfSet** (set< **IR_Stmt** * >) const
  Tests, whether the hole Flow Fact is contained in the specified Statements.
- virtual bool **isSignificant** () const
  Indicates, whether a Flow Fact is significant for WCET-calculation.
- virtual void **setPartOfFlowfactset** (bool)
  Specifies, whether this Flow Fact belongs to the Flowfactset.

Protected Member Functions

- virtual void **addRefToStmt** (**IR_Stmt** *)
  Adds Flow Fact to Flowfactref.
- virtual void **removeRefFromStmt** (**IR_Stmt** *)
  Removes Flow Fact from Flowfactref.

Constructor & Destructor Documentation

IR_Flowrestriction::IR_Flowrestriction ()

Default constructor.

Creates an empty Flowrestriction. The Summands could be added by:

- **addToLeq()**
- **addToGeq()**

This Flow Fact is not automatically added to the Flowfactset!

IR_Flowrestriction::IR_Flowrestriction (list< pair< int, IR_Stmt * > >, list< pair< int, IR_Stmt * > >)

Constructor.

Creates a Flowrestriction, the first argument determines the summands of the less equal side, the second argument determines the summands of the greater equal side.

This Flow Fact is not automatically added to the Flowfactset!

virtual IR_Flowrestriction::~IR_Flowrestriction () `[virtual]`

default destructor

Member Function Documentation

virtual void IR_Flowrestriction::write (ostream &, int = 0) const `[virtual]`

Writes the Flow Fact.

This function is used to write the Flow Fact to Logfiles,... The output has to be indented by the numbers of space characters specified by indent.

The output is:

Flowrestriction: ...*... + ...*... <= ...*...

Implements **IR_Flowfact** (*p.8*).

void IR_Flowrestriction::addToLeq (int, IR_Stmt *)

Adds a summand to Flowrestrictions less equal side.

The Statement with the given Integer as Factor (negative Factors will add the summand to the geq-side) is added to the leq side.

This is the only function, witch manipulates the intern data structure (and by this invokes the updates of Flowfactrefs).

void IR_Flowrestriction::addToGeq (int, IR_Stmt *)

Adds a summand to Flowrestrictions greater equal side.

The Statement with the given Integer as Factor (negative Factors will add the summand to the leq-side) is added to the geq side.

This method uses the addToLeq-Methode.

int IR_Flowrestriction::removeSummand (IR_Stmt *)

Removes a summand from Flowrestriction.

A hole summand is removed from Flowrestriction and its factor is returned. (No differentiation between leq and geq side!). Zero signalise, that the Summand is not found.

This method uses the addToLeq-Methode.

void IR_Flowrestriction::exchangeSummand (IR_Stmt *, IR_Stmt *)

Exchanges a summand in a Flowrestriction.

The Statement of the Summand, specified by the first argument, is exchanged by the Statememt spezified by the second one. The factor remains unchanged (No differentiation between leq and geq side!).

This method uses the addToLeq-Methode.

list<pair<int, IR_Stmt *> > IR_Flowrestriction::getLeq () const

Returns a copy of the leq-side.

A copy of the leq-side is returned (by this, the internal data structure could not be indirect manipulated).

list<pair<int, IR_Stmt *> > IR_Flowrestriction::getGeq () const

Returns a copy of the geq-side.

A copy of the leq-side is returned (by this, the internal data structure could not be indirect manipulated).

bool IR_Flowrestriction::isPartOfLeq (IR_Stmt *) const

Tests, whether a Statement belongs to the leq side of a Flowrestriction.

If the Statement is referenced on the left side of this Flowrestriction, true is returned, otherwise false.

bool IR_Flowrestriction::isPartOfGeq (IR_Stmt *) const

Tests, whether a Statement belongs to the geq side of a Flowrestriction.

If the Statement is referenced on the right side of this Flowrestriction, true is returned, otherwise false.

bool IR_Flowrestriction::isPartOfThis (IR_Stmt *) const

Tests, whether a Statement belongs to this Flowrestriction.

If the Statement is referenced in this Flowrestriction (leq or geq side), true is returned, otherwise false.

virtual bool IR_Flowrestriction::isContained (IR_Stmt *) `[virtual]`

Tests, whether the entire Flow Fact is contained in the specified Statement.

If every element of a Flow Fact is contained in the specified Statement and its substatement hierarchy, then the function returns true, otherwise (at least one element of the Flow Fact is not part of the specified Statement and its substatement hierarchy) false.

Implements **IR_Flowfact** (*p.8*).

bool IR_Flowrestriction::isPartOfSet (set< IR_Stmt * >) const

Tests, whether the hole Flow Fact is contained in the specified Statements.

This is similar to isContained, but a set of Statements could be specified (their substatement hierarchy is no(!) considered).

virtual bool IR_Flowrestriction::isSignificant () const `[virtual]`

Indicates, whether a Flow Fact is significant for WCET-calculation.

In some cases, a Flow Fact may not be significant for WCET-calculation, for example:

- A Flowrestriction with 0 <= SUM. In such cases, this function returns false, in all other cases true.

virtual void IR_Flowrestriction::setPartOfFlowfactset (bool) `[virtual]`

Specifies, whether this Flow Fact belongs to the Flowfactset.

This function sets the partOfFlowfactset-Bool. If a Flowfactset holds this Flow Fact, the Bool has to be true, otherwise false.

This function should be used carefully, because the automatic management of Flowfactrefs is influenced by this. If this Bool does not correspond with the state of the Flow Fact, inconsistent data structures can be generated.

Implements **IR_Flowfact** (*p.8*).

virtual void IR_Flowrestriction::addRefToStmt (IR_Stmt *) `[protected, virtual]`

Adds Flow Fact to Flowfactref.

The Flowfactref of the given Statement is told, that this Flow Fact is using its Statement (so that that Flowfactref has to hold a reference to this Flow Fact), but only, when this Flow Fact is part of a flowfactset. If no Flowfactref already exists, a new one (for the given statement) will be created.

Implements **IR_Flowfact** (*p.9*).

virtual void IR_Flowrestriction::removeRefFromStmt (IR_Stmt *) `[protected, virtual]`

Removes Flow Fact from Flowfactref.

The Flowfactref of the given Statement is told, that this Flow Fact is not longer using its Statement, but only, when this Flow Fact is part of a flowfactset.

Implements **IR_Flowfact** (*p.9*).

IR_Loopbound Class Reference

```
#include <irloopbound.h>
```

Detailed Description

Description of loopbounds for **IR_LoopStmt**.

A loopbound specifies the minimum and the maximum number of iterations (iterations, not number of tests of condition like in aiT).

Ff min (and max) == -1: no loopbound is available. This is implemented to detected loopbounds in LLIR also no loopbounds were specified yet. This is interesting for the translation to CRL2 which needs for every loop a single annotation.

Definition at line 47 of file irloopbound.h.

Public Member Functions

- **IR_Loopbound** ()
  Default constructor.
- **IR_Loopbound** (int, int, **IR_LoopStmt** *)
  constructor
- virtual **~IR_Loopbound** ()
  Default destructor.
- virtual void **write** (ostream &, int=0) const
  Writes the Flow Fact.
- void **setMin** (int)
  Sets the minimum number of iterations.
- void **setMax** (int)
  Sets the maximum number of iterations.
- void **setLoop** (**IR_LoopStmt** *)
  Sets the loopbound.
- int **getMin** () const
  Returns the minimum numbers of iteration specified in this loopbound.
- int **getMax** () const
  Returns the maximum numbers of iteration specified in this loopbound.
- **IR_LoopStmt** * **getLoopStmt** () const
  Returns the loop this loopbound is for.
- bool **isBoundSpecified** () const
  Returns true if loopbounds are specified.
- virtual bool **isContained** (**IR_Stmt** *)
  Tests, whether the whole Flow Fact is contained in the specified Statement.
- virtual void **setPartOfFlowfactset** (bool)
  Specifies, whether this Flow Fact belongs to the Flowfactset.

Protected Member Functions

- virtual void **addRefToStmt** (**IR_Stmt** *)
  Adds Flow Fact to Flowfactref.
- virtual void **removeRefFromStmt** (**IR_Stmt** *)
  Removes Flow Fact from Flowfactref.

Constructor & Destructor Documentation

IR_Loopbound::IR_Loopbound ()

Default constructor.

Creates an empty loopbound

IR_Loopbound::IR_Loopbound (int, int, IR_LoopStmt *)

constructor

Creates a loopbound. The first parameter is the minimum, the second the maximum numbers of iteration of the loop specified by the third parameter.

virtual IR_Loopbound::~IR_Loopbound () `[virtual]`

Default destructor.

Member Function Documentation

virtual void IR_Loopbound::write (ostream &, int = 0) const `[virtual]`

Writes the Flow Fact.

This function is used to write the Flow Fact to Logfiles,... The output has to be indented by the numbers of space characters specified by indent.

The output is:

Loopbound min: ... max: ... SOURCEFILECONTEXT

Implements **IR_Flowfact** (*p.8*).

void IR_Loopbound::setMin (int)

Sets the minimum number of iterations.

The new minimum has to be lower then the current maximum.

If the minimum is set to -1 (no loopbounds available), the maximum is also updated.

void IR_Loopbound::setMax (int)

Sets the maximum number of iterations.

The new maximum has to be larger then the current minimum.

If the minimum is -1 it is set to zero, because a loopbound is now specified.

void IR_Loopbound::setLoop (IR_LoopStmt *)

Sets the loopbound.

This function invokes the update of Flowfactrefs

int IR_Loopbound::getMin () const

Returns the minimum numbers of iteration specified in this loopbound.

If -1 is returned, no loopbound is available.

int IR_Loopbound::getMax () const

Returns the maximum numbers of iteration specified in this loopbound.

If -1 is returned, no loopbound is available.

IR_LoopStmt* IR_Loopbound::getLoopStmt () const

Returns the loop this loopbound is for.

If NULL is returned, no loop is specified. (But also loopbounds could be!)

Returning a loop does not implicate that a valid loopbound is specified.

bool IR_Loopbound::isBoundSpecified () const

Returns true if loopbounds are specified.

Does not consider whether a loopstatement is specified.

virtual bool IR_Loopbound::isContained (IR_Stmt *) `[virtual]`

Tests, whether the whole Flow Fact is contained in the specified Statement.

If the loopStatement is contained in the specified Statement and its substatementhierarchy, then the function returns true, otherwise false.

Implements **IR_Flowfact** (*p.8*).

virtual void IR_Loopbound::setPartOfFlowfactset (bool) `[virtual]`

Specifies, whether this Flow Fact belongs to the Flowfactset.

This function sets the partOfFlowfactset-Bool. If a Flowfactset holds this Flow Fact, the Bool has to be true, otherwise false.

This function should be used carefully, because the automatic management of Flowfactrefs is influenced by this. If this Bool does not correspond with the state of the Flow Fact, non-constent data structures can be generated.

Implements **IR_Flowfact** (*p.8*).

virtual void IR_Loopbound::addRefToStmt (IR_Stmt *) `[protected, virtual]`

Adds Flow Fact to Flowfactref.

The Flowfactref of the given Statement is told, that this Flow Fact is using its Statement (so that that Flowfactref has to hold a reference to this Flow Fact), but only, when this Flow Fact is part of a flowfactset. If no Flowfactref already exists, a new one (for the given statement) will be created.

Implements **IR_Flowfact** (*p.9*).

virtual void IR_Loopbound::removeRefFromStmt (IR_Stmt *) `[protected, virtual]`

Removes Flow Fact from Flowfactref.

The Flowfactref of the given Statement is told, that this Flow Fact is not longer using its Statement, but only, when this Flow Fact is part of a flowfactset.

Implements **IR_Flowfact** (*p.9*).

IR_ EntryPoint Class Reference

```
#include <irentrypoint.h>
```

Detailed Description

Description of entrypoints for **IR_Function**.

An entrypoint specifies that the control flow may enter the annotated function from the outside (through operating system calls, etc.). Thus a timing analysis starting from each of the given entrypoints should be conducted. If no entrypoint annotations are given, the `main` function is assumed to be the only entrypoint.

Public Member Functions

- **IR_EntryPoint** ()
  Creates an empty entrypoint.
- **IR_ EntryPoint** (**IR_Function** *)
  Creates an entrypoint for the specified function
- virtual **~IR_ EntryPoint** ()
  Default destructor.
- void **setFunction**(**IR_Function***);
  Sets the function
- **IR_Function** ***getFunction**() const;
  Returns the function to which this entrypoint belongs.
- virtual void **write** (ostream &, int=0) const
  Writes the Flow Fact.
- virtual bool **isContained** (**IR_Stmt** *)
  Tests, whether the whole Flow Fact is contained in the specified Statement.
- virtual void **setPartOfFlowfactset** (bool)
  Specifies, whether this Flow Fact belongs to the Flowfactset.

Protected Member Functions

- virtual void **addRefToStmt** (**IR_Stmt** *)
  Adds Flow Fact to Flowfactref.
- virtual void **removeRefFromStmt** (**IR_Stmt** *)
  Removes Flow Fact from Flowfactref.

Constructor & Destructor Documentation

IR_EntryPoint::IR_EntryPoint ()

Default constructor.

Creates an empty entrypoint.

IR_ EntryPoint::IR_ EntryPoint (IR_Function *)

constructor

Creates an entrypoint for the specified function.

virtual IR_ EntryPoint::~ IR_ EntryPoint () `[virtual]`

Default destructor.

Member Function Documentation

void IR_ EntryPoint::setFunction(IR_Function*);

Sets the function.

IR_Function *IR_ EntryPoint::getFunction() const;

Returns the function to which this entrypoint belongs.

irtual void IR_ EntryPoint::write (ostream &, int = 0) const `[virtual]`

Writes the Flow Fact.

This function is used to write the Flow Fact to Logfiles,... The output has to be indented by the numbers of space characters specified by indent.

Implements **IR_Flowfact** (*p.8*).

virtual bool IR_ EntryPoint::isContained (IR_Stmt *) `[virtual]`

Tests, whether the whole Flow Fact is contained in the specified Statement.

Always returns false for entrypoints.

Implements **IR_Flowfact** (*p.8*).

virtual void IR_ EntryPoint::setPartOfFlowfactset (bool) `[virtual]`

Specifies, whether this Flow Fact belongs to the Flowfactset.

This function sets the partOfFlowfactset-Bool. If a Flowfactset holds this Flow Fact, the Bool has to be true, otherwise false.

This function should be used carefully, because the automatic management of Flowfactrefs is influenced by this. If this Bool does not correspond with the state of the Flow Fact, non-constent data structures can be generated.

Implements **IR_Flowfact** (*p.8*).

virtual void IR_ EntryPoint::addRefToStmt (IR_Stmt *) `[protected, virtual]`

Adds Flow Fact to Flowfactref.

The Flowfactref of the given Statement is told, that this Flow Fact is using its Statement (so that that Flowfactref has to hold a reference to this Flow Fact), but only, when this Flow Fact is part of a flowfactset. If no Flowfactref already exists, a new one (for the given statement) will be created.

Implements **IR_Flowfact** (*p.9*).

virtual void IR_ EntryPoint::removeRefFromStmt (IR_Stmt *) `[protected, virtual]`

Removes Flow Fact from Flowfactref.

The Flowfactref of the given Statement is told, that this Flow Fact is not longer using its Statement, but only, when this Flow Fact is part of a flowfactset.

Implements **IR_Flowfact** (*p.9*).
