Klocwork C/C++ Path Analysis API

3.0.0

Generated by Doxygen 1.8.13

Contents

Chapter 1

Klocwork Path Analysis

1.1 General Information

Path checkers are a family of Klocwork checkers that use control flow and data flow analyses for finding feasible execution paths between certain execution points in the program whose source is being analysed. Path checkers are algorithms built on top of the Klocwork data flow analysis engine. These algorithms define the particular kinds of data to trace along the execution paths, as well as the forbidden operations with the data that are reported as defects by the checkers. Path checkers operate against an intermediate representation of the code that is called MIR (Medium level Intermediate Representation). In this code model, each function is represented by a control flow graph composed of MIR nodes, each node encoding an operation, and a potential set of incoming and outgoing edges. Klocwork MIR approximates what is known as a three-address code, where operation in each node consists of at most three operands. Path checkers follow the control-flow paths through the MIR, searching for the defects defined by their internal algorithms. A typical Path checker tracks a value from a source (a starting point for analysis) to a sink (the end point where the defect is detected).

Klocwork Path checkers need to be written in C++. Klocwork product has many Path checkers built into its core and shipped with it, as well as several additional Path checkers that are shipped as plugins (such as MISRA path checkers plugin). Klocwork Path API allows to create custom Path checkers plugins, although the API's data structures are high level and aimed at simplifying the underlying concepts thus, to a certain extent, sacrificing the breadth of problems that such custom checkers can solve. The Klocwork built-in checkers are not limited by these considerations and make use of the whole power of the Klocwork Path analysis engine.

This document aims to provide descriptions of the data structures and interfaces provided by Klocwork Path API, as well as rudimentary examples of their applications to solutions of typical dataflow defect detection problems.

Klocwork Path analysis engine is currently used in its C, C++, and C# analyses.

1.2 Terms

- **Control flow analysis.** A source code analysis method identifying control paths and order of execution of the individual operations.
- **Data flow analysis.** An analysis method identifying the possible values of the variables defined in the program.
- **Path analysis.** Klocwork analysis method combining control flow and data flow analyses within the same framework. Precise determination of control paths requires knowledge of the controlling variables values while determination of the possible variable value ranges requires knowledge of the particular paths taken by the execution.

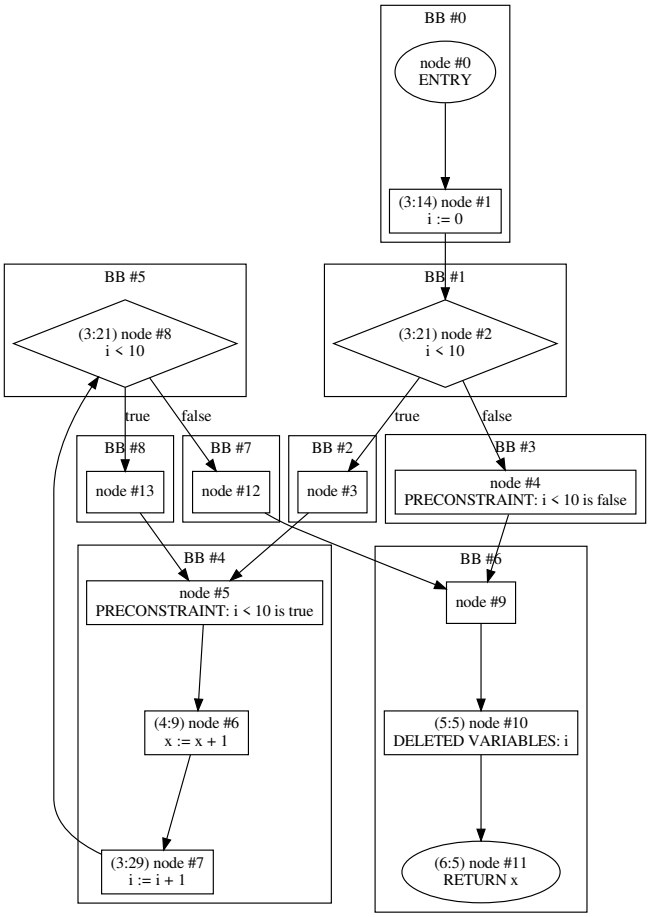
- **Control Flow graph of a program (CFG).** A representation, using graph notations, of all paths that might be traversed by the program in its execution. A CFG is composed of the following:
  - CFG nodes: Each node in the CFG represents a basic block of the program (see below).
  - Edges: represent the jumps in the Control Flow Graph between its nodes.
  - Basic Block. A portion of the code within a program with only one entry and only one exit. A basic block typically contains multiple subsequent operations.
- **Medium level Intermediate Representation (MIR).** A specific implementation of the CFG that Klocwork Path analysis engine builds and uses in the analysis. MIR consists of MIR nodes, connected by edges and grouped into the basic blocks. A single MIR corresponds to a single function defined in the source code.
- **MIR function.** A top-level container of a function CFG.
- **MIR nodes.** Data structures describing basic program operations, such as a variable assignment or a function call. Each node has a unique number assigned to it and a reference to the actual position in the source code. A node also describes two MIR expressions (see below), a 'read' expression and a 'write' expression, that, depending on the actual operation being described, can be empty or non-empty. There are different kinds of MIR nodes: expression block (single assignment) nodes, conditional branch (if-else, switch, and switch-throw nodes), and leaf and return nodes.
- **MIR expression.** A tree like data structure, describing variables, including temporary variables and fields of structures or classes, values, function or operator calls, pointer dereferences, binary, unary, ternary, or address of operations, and exception throws. An expression can contain a reference to the semantic information associated with that expression, where applicable. Semantic information can include name and type information, as well as relations to other entities in the source code.
- **Memory items.** Klocwork internal objects representing abstract locations in memory, typically variables or temporary objects.
- **Constraints.** Klocwork internal objects encoding numerical value ranges. A constraint associated to a memory item in a given MIR node is a typical result of Klocwork data flow analysis. A node or basic block pre-constraint describes the numerical conditions that the given entity must satisfy in order for the control flow to enter that node or block. Conversely, a post-constraint describes the numerical values that the entity will take after the control flow exits that node or basic block. All the above described concepts are accessible programmatically via Klocwork Path API, with the corresponding interfaces being declared in the header file **kpaMirUtil.hh** (p. 188).

1.3 Example of MIR Flowchart

To illustrate the MIR generation by Klocwork Path engine, consider the following code snippet:

```
int func(int x)
{
    for (int i = 0; i < 10; ++i) {
        x++;
    }
    return x;
}
```

The flow chart of the Klocwork generated MIR for this code snippet, along with the associated control flow information is depicted in the Figure below.



1.4 How Path Checkers Work

Before any Path checkers start their analysis, Klocwork Path engine performs a general control flow and data flow analysis to topologically sort MIR nodes and evaluate pre-conditions and post-conditions (pre- and post-constraints for all the relevant memory items) for all basic blocks. These can be retrieved with the calls to the API functions **kpa↩ ::bb_getPreConstraint()** (p. 116), **kpa::bb_getPostConstraint()** (p. 115), and **kpa::mi_getNodePreConstraint()** (p. 126).

Note

As mentioned above in the "Terms" section, the objects returned by these functions describe numerical values ranges for the requested memory items, e.g., $x \in [5, 10]$ or $y \in [20, 30] \cup [50, 60]$. Klocwork Path engine also calculates symbolic data flow information, e.g., $a > b + c$, however, the Path API does not currently have interfaces to expose this information. The symbolic information is extensively used internally by the engine, data propagators and checkers.

A Path checker will perform one or more traversals of the MIR in the direct and/or reverse topological order, depending on the internal checker configuration. Under the hood, any checker will have several MIR node visitors associated with it. A simple checker might just collect certain information of interest during the MIR traversal. An example of such simple traverser can be a checker that determines that a function does not use some of its parameters. Most Path checkers, however, require data flow information taken into account. A traverser that traces and accounts for the data flow information is called, in the Klocwork terminology, a 'propagation pass'. A pass 'propagates' data (typically, memory items along with certain numerical and symbolic conditions associated with them) through the MIR. The checker's MIR node visitors can transform this data in certain ways. The Klocwork

engine performing them can apply additional constraints to the propagated data as the pass enters new nodes, to consider the pre-calculated general data flow information. If the propagated data and the MIR node being visited satisfy certain conditions defined by the checker, a Path defect can be reported. The node where propagation starts is called a source node, and the one where the decision to report a defect is made, a sink node.

Under the hood, a pass calculates and updates its propagated data iteratively while traversing the MIR, thus it is possible for the node visitors of the checkers performing the pass, to enter the same nodes multiple times if loops or other nontrivial control flow constructions are present in the MIR. Typically, checkers shipped with Klocwork perform multiple forward and backward passes to solve their tasks. MIR traversers and propagation passes are low level internal concepts that are not exposed in the Klocwork Path API.

1.5 Klocwork Path API: Source-Sink checkers, Triggers, Source-Sink Processors, ...

Klocwork Path API introduces a concept of source-sink analyzer and defines an interface to it. A source-sink analyzer is a dataflow solver designed to find feasible paths between two MIR nodes, called, correspondingly, 'source' and 'sink' nodes. To achieve this goal, API defines so-called "triggers" that identify source and sink nodes by their properties and control details of data propagation between them.

A trigger is an abstract class whose `extract()` method receives an MIR node as a parameter. Depending on the node properties, the trigger can evaluate and add to its "trigger result" at this node a set of memory items that are then processed by the analyzer. Klocwork Path API defines five major categories of triggers: "source", "sink", "prop", "reject", and "check" triggers.

- All the memory items added to its result (triggered) by the **source** trigger will be scheduled for propagation by the analyzer. The propagation will start from the MIR node where the source has been triggered. For example, suppose we are writing a simple Null-Pointer Dereference checker. For such checker, a source trigger can add memory item for a pointer to its result in any nodes where assignment of those pointers to NULL takes place. For example, in assignment node `p = nullptr`, the trigger will add `p` to its result. That will cause the checker to propagate `p` from that node down the MIR.
- If any memory items triggered by the **sink** trigger are among the currently propagated memory items, then the defect source-sink path is considered identified. The path details are then scheduled to be reported by source-sink processor (see below) and the propagation is stopped. For example, if the trigger triggers a memory item for `p` in node where variable `p` is dereferenced, and that memory item was actually propagated, then we have found a Null-Pointer Dereference defect.
- If any memory items triggered by the **check** trigger are among the currently propagated memory items, then the propagation is aborted. This corresponds to an event found that proves that the code has no defect and thus makes further propagation pointless. For example, a variable `p` is explicitly checked in the node to be non-NULL.
- A **prop** trigger allows to add additional memory items to the list of propagated data. For example, in node `p1 = p + 4`, the trigger might add `p` to its "in" part and `p1` to its "out" part, to indicate that if `p` was propagated then `p1` should be propagated from now on as well.
- A **reject** trigger serves an opposite purpose, it indicates that certain memory items need to be removed from the list of propagated data starting from the triggering node.

A 'source-sink processor' is another object defined by the API that allow to customize details of defect reports, such as perform extra calculations or add more trace events. A `setProcessor()` method of analyzer object allows to use the custom processor in defect reports. All analyzers defined by API have a default defect processor that is used when not overriden by the `setProcessor()` method.

Klocwork product ships several general purpose checkers implementing the Source-Sink Analyzer interface. These checkers, that differ in details of their `analyze()` method, are declared in the **kpaSourceSinkAnalyzer.hh** (p. 195) header file and include:

- A "forward" source-sink analyzer, pointer to which is returned by the **getForwardSourceSink↩ Checker()** (p. 155) interface function. This is a straightforward but lightweight (and thus high performance) checker that does not propagate node specific conditions associated with the propagated data and thus cannot identify infeasible execution paths. For illustration, consider pseudocode

```
if (condition) {
    // a memory item mi is triggered as a source
    // mi propagation is started from now on
    // however, the forward analyzer does not 'remember' that
    // the condition needs to be satisfied everywhere along
    // the found path for that path to be feasible
}
...
if (!condition) {
    // mi sink is triggered here
    // the forward analyzer's propagation does not 'remember' about
    // condition thus the found path will be sent to the source-sink
    // processor that will report it as a (false positive) defect
}
```

  The forward analyzer will falsely report this pattern. It is thus recommended to only use for scenarios where the number of such patterns is expected to be low and checker performance needs to be high.

- A "conditional" source-sink checker pointer to which is returned by the **getConditionalSourceSink↩ Checker()** (p. 152) interface function. This checker will propagate the source conditions along with the memory items and thus will eliminate the false positive reports due to the erroneously taken infeasible paths illustrated in the above example. Propagating the conditions, however, can be computationally expensive, especially in the source with many branching nodes, thus performance of the conditional source-sink analyzer is typically noticeably slower than that of the forward one. Also, in practice, by design, this checker can run more than one conditional passes underneath that reduces its performance even further.

- A "backward" source-sink checkers. This is a checker similar to the forward source-sink checker except in it, the propagation is performed in the backward topological order. This checker is useful in situations when the number of sources is expected to be much larger than that of sinks, and thus a higher computational performance can be achieved if propagation starts from the sinks up towards the sources in the backward order.

- A "reverse" source-sink checker. This is an exotic checker suitable for use in the so-called reverse issues detection. Example of a "reverse" issue can be an unknown pointer value checked against NULL along one execution path and yet dereferenced along the other one. Under the hood, this checker is very similar to the "forward" source-sink checker, and is only different in certain internal propagation details. The "reverse" checker is designed for a specific use case; the "forward" source-sink checker is typically more suitable for general purposes.

- A "direct" checker, pointer to which is returned by the **getDirectChecker()** (p. 153) interface function. This is a simple MIR node traverser described in one of the sections above. Any sources or sinks triggered by this checkers triggers will be considered as defect paths, with the source and sink nodes be the same and equal to the triggered node.

Example: a simple Null Pointer Dereference checker with a Forward Source-Sink Checker (pseudocode).

```
SrcTrigger::extract(node result)
    if (node is assignment)
        and (node->read is a constant)
            and (value of that constant is zero)
                and (node->write is a pointer)
                    ==> add the pointer to result

SinkTrigger::extract(node result)
    if (node->read or node->write contains a dereferenced expression)
        ==> add pointer in this expression to the result
```

Then, for the following sample code:

```
1: int x;
2: int *p = 0;
3: int *q = &x;
4: ...
5: *q = 3;
6: ...
7: *p = 4
```

The analysis will be as follows:

- Checker starts node traversal to collect sources.
- At line 2, p will be added to result of the SrcTrigger. This will start a propagation pass for p.
- At line 3, q will not be added as another source since the right-hand side of the assignment at that line is not zero.
- At line 5, q will be added to result of the SinkTrigger, but it does not match the p variable in the first step, so no defect is reported.
- At line 7, p will be added to result in the SinkTrigger, and it matches the p variable in the first step, so the defect is reported.

1.6 Interprocedural Analysis

Any source-sink analyzer defined in the path API runs analysis in the scope of a single function. In order to propagate knowledge between different functions that are related via direct or indirect call relations, special analyzers need to be defined, in addition to the checkers, that work with Klocwork interprocedural analysis infrastructure.

Klocwork has a bottom-up approach to interprocedural analysis. In this approach, all functions in the project are first sorted in topological order to form a call graph. Analysis starts from leaves of this graph and proceeds in the backward topological order towards the graph root (that can be, e.g., function `main`). Results of analysis for any given function, that can be important for its callers behaviors, are cached in form of "function behavioral knowledge base", or FBKB, also referred here as simply "KB". During analysis of the callers, the called functions are not re-analyzed again; instead, the cached KB knowledge is used.

To prepare the knowledge for caching in FBKB, Klocwork runs "KB generators". A KB generator is a variety of a path checker that prepares and stores the knowledge necessary for interprocedural analysis for its specific dataflow problem. Klocwork analysis engine ships a number of built-in "general" KB generators that are responsible for integrity of dataflow information in interprocedural analysis in general. An example of this general knowledge can be, e.g., information about ranges of the values returned by function as well as conditions under which these ranges are valid. Besides that, Klocwork ships KB generators that are only specific for it built-in path checkers (such as ABV or MLK checker).

In addition, every installation of Klocwork will have a number of "shipped" KB records for typical commonly used library functions, such as those in C standard library. These "external" functions do not normaly have source code available for analysis, and the shipped KB is used in lieu of the cached analysis results for such functions. Any KB record encodes its related behaviors and associates them with function parameters or their fields. These associations are called "KB socket expressions" or, simply, "KB sockets". There are typically multiple KB records for the same function, describing different behaviors.

In order to add support for KB records associated with function calls to the custom checkers, the analyzer interface defines the following methods:

- `addKBSource(KB_name, ...)`. This method will internally add a source trigger associating calls having a "KB_name" record with defect source. The internal source trigger will extract memory items evaluated from the expression in the KB record found for the called function and add them to the trigger result as a defect source.

- `addKBSink(KB_name, ...)`. This method internally creates and adds to the analyzer a sink trigger that evaluates memory items from details of KB records "KB_name" found for called functions. The evaluated memory items are added to the trigger result and treated as defect sinks.
- `addKBCheck(KB_name)`. Similarly to the above source and sink, the evaluated memory items are added to the result of an internally created check trigger.
- `addKBReject(KB_name)`. Similarly to the above source and sink, the evaluated memory items are added to the result of an internally created reject trigger.

Custom path checkers created with Klocwork Path API will often require generation of their own custom KB records in order to support interprocedural analysis properly. The API is currently shipped with four KB generators that can be used for generating custom KB records.

- `getForwardSourceFBKBGenerator(KB_name, ...)`. This API function creates and returns pointer to an analyzer that has a SourceSinkProcesor defined to create a KB record. The redefined Source↩ SinkProcessor creates KB records for sink occurrences that are externally visible (e.g., returned values). Note that source KB generator creates KB for expressions triggered as sinks for its analysis.
- `getForwardSinkFBKBGenerator(KB_name, ...)`. Similarly to the source KB generator described above, this returns a pointer to an analyzer with an overridden SourceSinkProcessor, that creates KB records for found dataflow paths, except, the KB is created for externally visible expressions triggered as sources for this analyzer, such as function parameters (again, sink KB generator reports its sources as KB sockets).
- `getConditionalSourceFBKBGenerator(KB_name, ...)`. This function returns a pointer to an analyzer that generates source KB while eliminating infeasible conditions.
- `getConditionalSinkFBKBGenerator(KB_name, ...)`. This function returns a pointer to an analyzer that generates sink KB while eliminating infeasible conditions in the analyzed function.

Note that check and reject KB records can typically be generated using a sink KB generator with properly defined triggers.

In order for the interprocedural infrastructure to recognize a KB kind, it needs to be registered during checker initialization using interface function **`kpa::registerKBKindForConditionalSocket()`** (p. 169).

1.7 Important Changes in API v.3.0.0

In Klocwork release 2023.1, path API has been upgraded to v.3.0.0. This version is not backward compatible with the previous API versions. The breaking change in 2023.1 is due to the ability to analyze each compilation unit using multiple engine threads. This is a second-order parallelization, added on top of the already existing parallel analysis using separate engine instances for different compilation units. That existing parallel analysis was limited by how many nodes of the high-level callgraph could be analyzed independently in parallel, and presented a performance bottleneck for some projects. For example, if the entire program is written in just one compilation unit, the previously existing analysis paradigm was unable to benefit from multiple CPU hardware. The added second-order parallelization was designed to address this limitation.

When a thread runs analysis, it stores some of the related information necessary for analysis correctness in an opaque object called `ThreadContext`. This object is owned and maintained by the analysis thread. Some API interface functions now consume information stored in that object and need a reference to it passed to them as an extra argument. In particular, all factory functions creating analyzers and KB-generators now need a reference to `ThreadContext` passed to them. In addition, **kpa::Trigger** (p. 183) class now stores a reference to `Thread↩ Context` as its member `threadContext`, and no longer has a default constructor. Every **kpa::Trigger** (p. 183) object needs to be created using a constructor with a `ThreadContext &` parameter. The `threadContext` member can be passed to other API functions called by the trigger where needed.

The signature of `functionHook_t`, an entry point to checkers and KB generators, has been modified to require a reference to `ThreadContext` passed to them:

```
typedef void (*functionHook_t) (function_t, ThreadContext &);
```

The `functionHook_t` pointer is called by the engine to execute the custom checker. The `ThreadContext` object, as mentioned, is owned by the engine thread, and now reference to it is passed to every custom checker entry, to be used with analyzers, triggers, and other devices.

In multithreaded analysis, it is possible now that the same checker code is executed in parallel by different threads. To avoid race conditions, checkers should not store any data in global objects (either globally or statically visible). If an existing custom checker is using such global objects, it needs to be redesigned to store the information locally instead.

Once these changes are accommodated, the custom checkers need to be recompiled and redeployed. Any custom checker plugins compiled with a previous API version, will not load with Klocwork 2023.1.

Chapter 2

Tutorials

The tutorials provided in this chapter require a desktop installation of Klocwork. Custom checkers can be run with all other Klocwork installation types as well, when deployed properly with them.

2.1 Tutorial: Use kwcreatechecker to Create a Sample C/C++ Path Checker

To facilitate creation and deployment of the custom Path checker plugins, Desktop installations of Klocwork come with kwcreatechecker tool. In this tutorial, we will create a sample checker, called MallocDelete. This is a very basic checker, that is designed to detect scenarios in which a memory allocated with a call to `malloc()` is released with C++'s `operator delete`. This scenario is one of many that are supported by Klocwork's own built-in checker FMM. We will build this custom checker here for educational purposes only.

Step 1. Run kwcreatechecker

In the directory where you want the checker files to be created, run kwcreatechecker with the following options:

```
kwcreatechecker --language cxx --type path --code MallocDelete
```

This will make a MallocDelete subdirectory and create template for the checker and Makefile for building the checker plugin in that subdirectory, as well as a sample source file with a defect in it, that can be used to test the checker. The checker template created by default by kwcreatechecker contains a bare-bone analyzer that already supports our malloc-delete pattern detection.

Step 2. Build the checker

Now we will compile the checker and create a build specification file that we can use to test it.

Windows

- Open the Visual Studio prompt.
- Go to the MallocDelete checker directory.
- Run:

  ```
  nmake install
  ```

- To generate the build specification to test our deployed checker, run:

  ```
  nmake buildspec
  ```

Unix

- With GNU make in path, run:

```
make install
```

- To generate the build specification to test the deployed checker, run:

```
make buildspec
```

Step 3. Deploy the checker

Now we can deploy the checker to our desktop so that it can be used by the analysis engine to detect the new issue type. In the previous step, a MallocDelete.zip file should have been created. This file contains an archive of all the custom checker plugin's files. The Klocwork toolchain loads custom plugins from the following location:

- Unix: ~/.klocwork/plugins
- Windows: %USERPROFILE%\.klocwork\plugins

If the above directory does not exist, create it manually. Unzip the zip file into it.

Step 4. Test the checker

We are going to use kwcheck, the Klocwork command-line analysis tool, to test the checker and verify that the new issue type is detected in the test source code.

- Set up the project to test. The "[n]make buildspec" command in step 2 should have created a kwinject.out file containing the buildspec for analysis of the sample test source file created by kwcreatechecker. We will use it to set up a local Klocwork project for analysis:

```
kwcheck create -b kwinject.out
```

- Run the analysis

```
kwcheck run
```

If the checker has been correctly built and deployed, the following message should be found in the list of reported defects:

```
testcase.cc:5 MallocDelete (1:Critical) Analyze
Your message goes here
```

To modify the defect message, edit the checkers.xml file, then re-build and re-deploy the checker.

Discussion

Now that we've ensured our checker works and detects the issue, let's discuss it.

In the checker.cpp file created by kwcreatechecker tool, there is a `ktc_plugin_library_info()` function defined. This function is executed by the plugin load infrastructure within Klocwork as the plugin dynamic library is being loaded. From there, the infrastructure will eventually execute `init()` function, pointer to which is stored in a structure accessible from `ktc_plugin_library_info` return. The `init()` function, in turn, executes `registerFunctionHook(processFunction)` that schedules `processFunction` to be executed along with other checkers. During analysis, `processFunction(...)` will be executed as many times as there are functions in the compilation unit.

Inside `procesFunction(...)`, we create a conditional source-sink analyzer, and add a source and a sink trigger to it:

```
static void processFunction(function_t function, ThreadContext &threadContext)
{
    // Create analyzer
    SourceSinkAnalyzerPtr a = getConditionalSourceSinkChecker(KPA_PLUGIN_NAME_STR, threadContext);
    // Register triggers
    a->addSourceTrigger(new MallocTrigger(threadContext),
                        getEvent("Put message here (use '%v')", threadContext));
    a->addSinkTrigger  (new DeleteTrigger(threadContext),
                        getEvent("Put message here (use '%v')", threadContext));
    // Run analysis
    a->analyze(function);
}
```

The `MallocTrigger` object, added as a source, simply checks in its `extract()` method for a call to a function that has name `malloc`, and adds the expression storing the call returned value, ("written" expression here), in the trigger result.

```
class MallocTrigger : public Trigger {
    ...
    void extract(node_t node, TriggerResult *res)
    {
        if (expr_isCallTo(node_getReadExpression(node), "malloc") &&
      node_getWrittenExpression(node)) {
            res->add(node_getWrittenExpression(node), threadContext);
        }
    }
};
```

The `DeleteTrigger` object, added as a sink, checks for the node being a call to `operator delete`, and adds its argument to the trigger result.

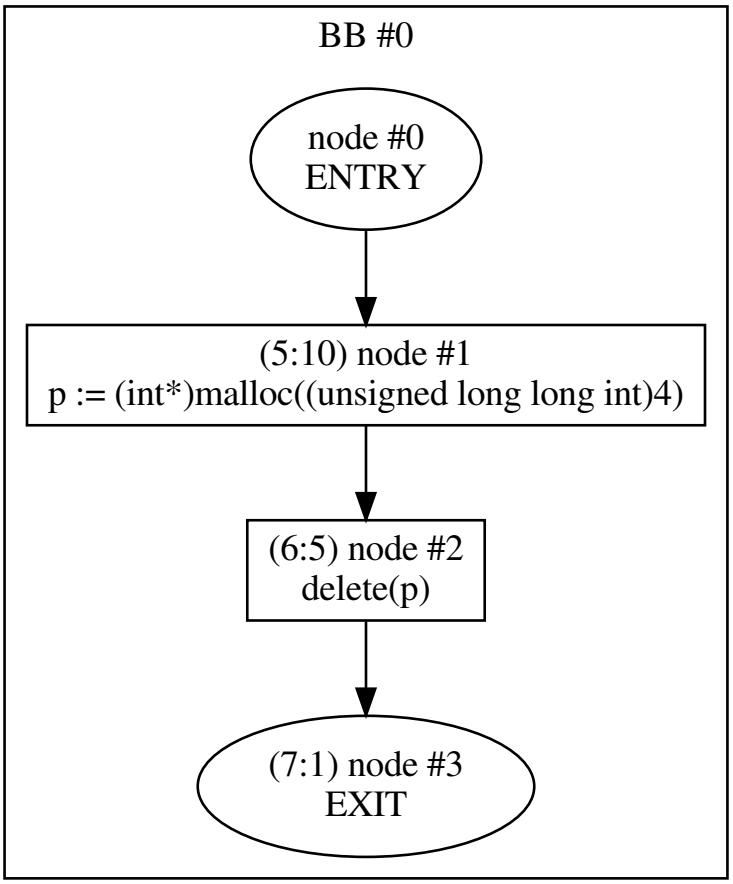
```
class DeleteTrigger : public Trigger
{
    ...
    void extract(node_t node, TriggerResult *res)
    {
        if (expr_isCallTo(node_getReadExpression(node), "delete") &&
            expr_getCallArgument(node_getReadExpression(node), 1))
        {
            res->add(expr_getCallArgument(node_getReadExpression(node), 1), threadContext);
        }
    }
};
```

Internally, any trigger's `res->add()` method converts the added MIR expressions to memory items. For the sink trigger, if the resulting memory items match any items propagated by the solver, the analyzer will issue a defect report.

There are two functions defined in the testcase.cc file created by kwcreatechecker. Let's consider the first one

```
void positive_test()
{
    int *p = (int *)malloc(4);
    delete p;
}
```

Its MIR is depicted in the following flowchart:



When checker's `a->analyze(function)` method is called, the MIR is traversed, with its all source triggers' `extract()` methods called in each node. Our small checker has only one source trigger, but analyzer can have arbitrary number of source triggers defined for better separation of different scenarios. In node 0, `expr_isCall↩To(node_getReadExpression(node), "malloc")` test in our `MallocTrigger` object evaluates to false, and no expressions are added to trigger result. In the next node 1 though, all the conditions are satisfied, and `res->add()` is called with the expression written in the node (variable `p`). This results in a dataflow propagation pass for `p` started by the analyzer from this node. For each such propagation pass, the rest of MIR reachable from the starting node is traversed in topological order, with dataflow information related to the propagated memory items updated in each node. The updated information can contain, e.g., information about aliases of the original memory item, or items overwritten in the node. In addition to it, `extract()` methods for all custom added check, reject, prop and sink triggers are called. In particular, in our example, `DeleteTrigger::extract()` is called at node 2. The conditions in the test within the method are satisfied in that node, that results in `p` being added to the trigger result. The analyzer compares the added memory items to the list of propagated ones, finds a match, and issues the defect. (In practice, conditional analyzer can run several more propagation passes to evaluate conditions accurately, but those are implementation details that are beyond the scope of this tutorial). Default analyzer's behavior is propagation in the branch is stopped for a source if a defect is reported. Finally, the original traversal identifying sources proceeds from node 1 to nodes 2 and 3, where no new sources are identified and thus no other propagation passes are started.

For the second function,

```
void negative_test()
{
    int *p = (int *)malloc(4);
    free(p);
}
```

the MIR is similar, except node 2 now has call to `free()`. Similar to the first example, `MallocTrigger` initiates propagation in node 1, but `DeleteTrigger` does not add anything to its result neither in node 2 nor in node 3 that results in no defects reported for that function.

The **`getEvent()`** (p. 153) calls in the checker's code create so-called `DataUpdater` objects that are designed to modify some information associated with the propagated items. The `DataUpdater` object created by **`getEvent()`** (p. 153) call adds trace events with, here, variable names inserted in slots '%v'. See details for **kpa::getEvent** (p. 153) below in this document. We can change the default trace messages created by kwcreatechecker to something nicer, e.g.,

```
a->addSourceTrigger(new MallocTrigger(threadContext),
                    getEvent("Pointer returned by call to '%f' is stored in '%v'", threadContext));
a->addSinkTrigger  (new DeleteTrigger(threadContext),
                    getEvent("'%v' is released with a call to '%f'", threadContext));
```

Finally, since the checker does not define a custom source-sink processor, a default one is used that simply issues the defect with the static message from the checkers.xml file packed in the zip file along with the checker's plugin library.

2.2 Tutorial: Add Support For Interprocedural Analysis

Let us slightly modify our test case to add a wrapper to the `malloc()` function:

```
int *malloc_wrapper()
{
    int *p = (int *)malloc(4);
    return p;
}

void positive_test()
{
    int *p = malloc_wrapper();
    delete p;
}
```

Since our analyzer works at the scope of a single function, and there is no longer a direct call to `malloc()` within `positive_test()`, the source trigger will not schedule any memory items for propagation, and no defect is detected with this test.

This scenario is the simplest example of a defect crossing single function boundaries, and it requires interprocedural analysis in order to be detected correctly.

As discussed in Chapter 1, Klocwork's approach to interprocedural analysis involves analyzing the called functions before their callers, and caching analysis results for the called functions in the knowledge base. Knowledge base records for a function, or simply its "KBs", can differ by their KB kinds. A specific KB kind is typically aimed at caching analysis results for its designated checker.

For our `malloc_wrapper()` example, we want to introduce a KB kind that will cache the fact that the function returns result previously returned by `malloc()`. Let us designate KB kind "MallocDeleteSourceKB" for this purpose.

Adding KB support for the checker is two-fold. First, we need to add support in the analyzer itself to treat functions having a "MallocDeleteSourceKB" record as additional defect sources. Second, we need to ensure that "Malloc↩ DeleteSourceKB" is automatically generated for functions returning pointer previously returned by `malloc()`.

Step 1. Add support for new KB in the analyzer. First, we need the engine to be aware of the "MallocDelete↩ SourceKB" KB kind. For this, we will use the API function **kpa::registerKBKindForConditionalSocket()** (p. 169). In the checker.cpp file, modify the init() function to add the following call to it:

```
registerKBKindForConditionalSocket("MallocDeleteSourceKB");
```

To support the new KB kind with the analyzer, we can simply add a built-in KB source trigger to it inside the `processFunction()`, using analyzer's `addKBSource()` method:

```
a->addKBSource("MallocDeleteSourceKB");
```

Step 2. Add KB generator. As discussed above, a KB generator is simply an analyzer with a special default SourceSinkProcessor that issues a KB record instead of a defect report. We will use **kpa::getConditionalSource↩ FBKBGenerator()** (p. 152) function here to create it.

A KB generator should be run separately from analysis, thus we need to create it in a separate function, say, `generateSourceKB()`. Similarly to `processFunction()` already present in our checker.cpp file, let's create KB generator interface, and create the analyzer in it:

```
SourceSinkAnalyzerPtr a = getConditionalSourceFBKBGenerator("MallocDeleteSourceKB", threadContext);
```

Similarly to the regular analyzer, our KB generator needs to use source triggers – exact same ones as the analyzer.

```
a->addKBSource("MallocDeleteSourceKB");
a->addSourceTrigger(new MallocTrigger(threadContext),
                    getEvent("Pointer returned by call to '%f' is stored in '%v'", threadContext));
```

Unlike the regular analyzer, the sink trigger for the source KB generator is a return statement within the analyzed function. The API ships a built-in trigger catching returned variables and adding them to the trigger result, that can be created by **kpa::getReturnTrigger()** (p. 168) function:

```
a->addSinkTrigger(getReturnTrigger(threadContext), getEvent("'%v' is returned", threadContext));
```

Note that if we were writing a sink KB generator instead of the source KB one, we would need to use the same sink triggers as the analyzer and **kpa::getInputTrigger()** (p. 166) as a source, to start analysis with function arguments.

Finish the KB generator by calling the regular `analyze()` method of the analyzer:

```
a->analyze();
```

Finally, the KB generator needs to be scheduled by execution inside the `init()` function:

```
registerKBGeneratorFunctionHook(generateSourceKB);
```

Putting it all together

Here is the source code of our checker that now supports interprocedural analysis for its sources.

```
using namespace kpa;

class MallocTrigger : public Trigger {
    REF_COUNTING_IMPL
public:
    MallocTrigger(ThreadContext &ctx) : Trigger(ctx) {}
    void extract(node_t node, TriggerResult *res)
    {
        if (expr_isCallTo(node_getReadExpression(node), "malloc") &&
      node_getWrittenExpression(node)) {
            res->add(node_getWrittenExpression(node), threadContext);
        }
    }
};

class DeleteTrigger : public Trigger {
    REF_COUNTING_IMPL
public:
    DeleteTrigger(ThreadContext &ctx) : Trigger(ctx) {}
    void extract(node_t node, TriggerResult *res)
    {
        if (expr_isCallTo(node_getReadExpression(node), "delete") &&
            expr_getCallArgument(node_getReadExpression(node), 1))
        {
            res->add(expr_getCallArgument(node_getReadExpression(node), 1), threadContext);
        }
    }
};

static void processFunction(function_t function, ThreadContext &threadContext)
{
    // Create analyzer
    SourceSinkAnalyzerPtr a = getConditionalSourceSinkChecker(KPA_PLUGIN_NAME_STR, threadContext);
    // Register triggers
    a->addKBSource("MallocDeleteSourceKB");
    a->addSourceTrigger(new MallocTrigger(threadContext),
                        getEvent("Pointer returned by call to '%f' is stored in '%v'", threadContext));
    a->addSinkTrigger  (new DeleteTrigger(threadContext),
                        getEvent("'%v' is released with a call to '%f'", threadContext));
    // Run analysis
    a->analyze(function);
}

static void generateSourceKB(function_t function, ThreadContext &threadContext)
{
    SourceSinkAnalyzerPtr a = getConditionalSourceFBKBGenerator("MallocDeleteSourceKB", threadContext);
    a->addKBSource("MallocDeleteSourceKB");
    a->addSourceTrigger(new MallocTrigger(threadContext),
                        getEvent("Pointer returned by call to '%f' is stored in '%v'", threadContext));
    a->addSinkTrigger(getReturnTrigger(threadContext), getEvent("'%v' is returned", threadContext));
    a->analyze(function);
}

static void init()
{
    registerKBKindForConditionalSocket("MallocDeleteSourceKB");
    registerFunctionHook(processFunction);
    registerKBGeneratorFunctionHook(generateSourceKB);
}
```

For a full interprocedural support, we also need to generate a sink KB for `operator delete` wrappers, which is done similarly to the source KB support.

Chapter 3

Deprecated List

Member kpa::constraint_getMaxValue (p. **117) (constraint_t cons, long int *a)**

This function is only dealing with 32 bit signed integers.
It is replaced by **constraint_getMaxValue(constraint_t cons)** (p. 118).

Member kpa::constraint_getMinValue (p. **119) (constraint_t cons, long int *a)**

This function is only dealing with 32 bit signed integers.
It is replaced by **constraint_getMinValue(constraint_t cons)** (p. 119).

Member kpa::constraint_getValue (p. **120) (constraint_t cons)**

This function is only dealing with 32 bit signed integers.
It is replaced by **constraint_getEQValue(constraint_t cons)** (p. 117).

Member kpa::constraint_isEQ (p. **121) (constraint_t cons, long int *a)**

This function is only dealing with 32 bit signed integers.
It is replaced by **constraint_getEQValue(constraint_t cons)** (p. 117).

Member kpa::constraint_isGE (p. **122) (constraint_t cons, long int *a)**

This function is only dealing with 32 bit signed integers.
It is replaced by a combination of **constraint_getMinValue(constraint_t cons)** (p. 119) and **constraint_hasMaxValue(constraint_t cons)** (p. 120).

Member kpa::constraint_isInterval (p. **122) (constraint_t cons, long int *a, long int *b)**

This function is only dealing with 32 bit signed integers.
It is replaced by a combination of **constraint_getMinValue(constraint_t cons)** (p. 119) and **constraint_getMaxValue(constraint_t cons)** (p. 118).

Member kpa::constraint_isLE (p. **123) (constraint_t cons, long int *a)**

This function is only dealing with 32 bit signed integers.
It is replaced by a combination of **constraint_hasMinValue(constraint_t cons)** (p. 121) and **constraint_getMaxValue(constraint_t cons)** (p. 118).

Member kpa::constraint_isNE (p. **124) (constraint_t cons, long int *a)**

This function is only dealing with 32 bit signed integers.
It is replaced by **constraint_getNEValue(constraint_t cons)** (p. 120).

Member kpa::constraint_isValue (p. **125) (constraint_t cons)**

This function was partially duplicated by **constraint_isEQ()** (p. 121).
It is replaced by **constraint_getEQValue(constraint_t cons)** (p. 117).

Member kpa::expr_getIntegerConstantValue (p. **93) (expr_t expr, int *error_flag)**

This function is only dealing with 64 bit signed integers.
It is replaced by **expr_getIntegerConstantValue(expr_t expr)** (p. 93).

Chapter 4

Module Index

4.1 Modules

Here is a list of all modules:

Chapter 5

Namespace Index

5.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 6

Hierarchical Index

6.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 7

Class Index

7.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 8

File Index

8.1 File List

Here is a list of all files with brief descriptions:

Chapter 9

Module Documentation

9.1 Obtaining configuration parameters for an error

Typedefs

- typedef struct ParameterNode * **kwapi_cfgparam_t**

Functions

- **kwapi_cfgparam_t** **kwapi_cfgparam_getRootParameterList** (const char *error)
- **kwapi_cfgparam_t** **kwapi_cfgparam_getListNodeByName** (**kwapi_cfgparam_t**, const char *name)
- **kwapi_cfgparam_t** **kwapi_cfgparam_getListNodeByRegexMatchingName** (**kwapi_cfgparam_t**, const char *name)
- const char * **kwapi_cfgparam_getName** (**kwapi_cfgparam_t**)
- const char * **kwapi_cfgparam_getType** (**kwapi_cfgparam_t**)
- **kw_size_t** **kwapi_cfgparam_getListLength** (**kwapi_cfgparam_t**)
- **kwapi_cfgparam_t** **kwapi_cfgparam_getListNodeByIndex** (**kwapi_cfgparam_t**, **kw_size_t** idx)
- int **kwapi_cfgparam_isParameter** (**kwapi_cfgparam_t**)
- const char * **kwapi_cfgparam_getParameterValue** (**kwapi_cfgparam_t**)
- const char * **kwapi_cfgparam_getParameterValueFromList** (**kwapi_cfgparam_t** parent, const char *paramName)
- const char * **kwapi_cfgparam_getConfigurationParameter** (const char *errorId, const char *paramName)
- const char *const * **kwapi_cfgparam_getCheckerErrors** (const char *checker_id)
- const char * **ktc_error_getConfigurationParameter** (const char *errorId, const char *paramName)
- int **kwapi_cfgparam_errorIsEnabled** (const char *error_id)

9.1.1 Detailed Description

Checker configuration XML file may store parameters that checker may access using functions from this section.

```
<error id="MYDEFECT" message="my message" severity="3" enabled="true">
  <parameter name="myparameter" value="yes"/>
</error>
```

9.1.2 Typedef Documentation

9.1.2.1 kwapi_cfgparam_t

```
typedef struct ParameterNode*  kwapi_cfgparam_t
```

9.1.3 Function Documentation

9.1.3.1 ktc_error_getConfigurationParameter()

```
const char* ktc_error_getConfigurationParameter (
            const char * errorId,
            const char * paramName )
```

9.1.3.2 kwapi_cfgparam_errorIsEnabled()

```
int kwapi_cfgparam_errorIsEnabled (
            const char * error_id )
```

Check that particular error was enabled in the configuration file

Parameters

error_id	identifier of an error (the same id as in <error> tag in configuration file)

Returns

0 if error was disabled or was not present in the configuration file, 1 otherwise

9.1.3.3 kwapi_cfgparam_getCheckerErrors()

```
const char* const* kwapi_cfgparam_getCheckerErrors (
            const char * checker_id )
```

Obtain a pointer to the internal array of error ids that are produced by a given checker

Parameters

checker_id	Identifier of a checker

Returns

a pointer to a zero terminated array of strings containing error identifiers or 0 if checker was not configured

9.1.3.4 kwapi_cfgparam_getConfigurationParameter()

```
const char* kwapi_cfgparam_getConfigurationParameter (
            const char * errorId,
            const char * paramName )
```

Quick access to parameters for errors that do not require parameterlist'.

Remarks

Essentialy gets a parameter by name from a root list `kwapi_cfgparam_t el = kwapi_cfgparam_getRootParameterList(errorId); if (el) { return kwapi_cfgparam_getParameterValueFromList(el, paramName); } else { return 0; }`

9.1.3.5 kwapi_cfgparam_getListLength()

```
kw_size_t kwapi_cfgparam_getListLength (
            kwapi_cfgparam_t  )
```

Returns 0 for parameter's, list length for parameterlist's

9.1.3.6 kwapi_cfgparam_getListNodeByIndex()

```
kwapi_cfgparam_t kwapi_cfgparam_getListNodeByIndex (
            kwapi_cfgparam_t ,
            kw_size_t idx )
```

Returns 0 for parameters, parameter node at index idx for lists, or 0 if index is outside of bounds

9.1.3.7 kwapi_cfgparam_getListNodeByName()

```
kwapi_cfgparam_t kwapi_cfgparam_getListNodeByName (
            kwapi_cfgparam_t ,
           const char * name )
```

Extract parameter node from a list by name,

Returns

0 if there is no parameter node with such name, or parameter node is not a parameterlist

9.1.3.8 kwapi_cfgparam_getListNodeByRegexMatchingName()

```
kwapi_cfgparam_t kwapi_cfgparam_getListNodeByRegexMatchingName (
            kwapi_cfgparam_t ,
           const char * name )
```

Extract parameter node from a list by name, where the keys associated with nodes are treated as PERL-compatible regular expressions (https://www.pcre.org) when verifying if name matches a list entry. Specifically, when kwapi_cfgparam_t denotes a ParameterList (i.e., a list of key-value pairs whose key is of type string and value is of type ParameterNode), this function will return the ParameterNode of the element e in kwapi_cfgparam_t where e's key is a PERL-compatible regular expression, and this regular expression matches name in its entirety. If there are multiple elements in kwapi_cfgparam_t whose key matches name, then the ParameterNode of the first element encountered is returned. Otherwise:

Returns

0 if there is no parameter node in list kwapi_cfgparam_t whose key is a PERL-compatible regular expression that matches name, or parameter node is not a parameterlist

9.1.3.9 kwapi_cfgparam_getName()

```
const char* kwapi_cfgparam_getName (
            kwapi_cfgparam_t  )
```

Get name of parameter node, or 0 if parameter has no name

9.1.3.10 kwapi_cfgparam_getParameterValue()

```
const char* kwapi_cfgparam_getParameterValue (
            kwapi_cfgparam_t  )
```

Return parameter string value for parameters, 0 for parameterlists

9.1.3.11 kwapi_cfgparam_getParameterValueFromList()

```
const char* kwapi_cfgparam_getParameterValueFromList (
            kwapi_cfgparam_t parent,
           const char * paramName )
```

Get parameter value from a parameter in the list by parameter name

Remarks

essentially a code: `kwapi_cfgparam_t el = kwapi_cfgparam_getListNodeByName(parent); if (el) { return kwapi_cfgparam_getParameterValue(paramName); } else { return 0; }`

9.1.3.12 kwapi_cfgparam_getRootParameterList()

```
kwapi_cfgparam_t kwapi_cfgparam_getRootParameterList (
            const char * error )
```

Get handler of root parameter list: it contains all parameter's and parameterlist's at the top level of <error> tag

9.1.3.13 kwapi_cfgparam_getType()

```
const char* kwapi_cfgparam_getType (
              kwapi_cfgparam_t  )
```

Get type of parameter node , or 0 if there is no type specified

9.1.3.14 kwapi_cfgparam_isParameter()

```
int kwapi_cfgparam_isParameter (
              kwapi_cfgparam_t  )
```

Returns 1 if node is parameter, 0 if it is parameterlist

9.2 MIR

Modules

- **Basic MIR types**
- **Constructors for kpa::integer_t**
- **Assignment operators for kpa::integer_t**
- **Conversion methods for kpa::integer_t**
- **Cast methods for kpa::integer_t**
- **Binary arithmetic operators for kpa::integer_t**
- **Unary arithmetic operators for kpa::integer_t**
- **Pre/post-inc/decrement operators for kpa::integer_t**
- **Arithmetic assignment operators for kpa::integer_t**
- **Relational operators for kpa::integer_t**
- **Binary bitwise operators for kpa::integer_t**
- **Bitwise assignment operators for kpa::integer_t**
- **Internal methods for kpa::integer_t (do not use)**
- **Binary operators when the left-hand side is a builtin integer and the right-hand side is a kpa::integer_t**
- **Extension points for Path Analysis**
- **Working with MIR nodes**
- **Working with MIR edges**
- **MIR expression trees**
- **Frontend information**
- **Numerical codes for the compilation unit language**

Classes

- class **kpa::integer_t**
- struct **kpa::edgeIterator_tag**

Functions

- **kpa::integer_t::~integer_t** ()

  Destructor of an integer.
- bool **kpa::integer_t::isValid** () const
- bool **kpa::integer_t::operator!** () const
- virtual size_t **kpa::NodeCollection::size** () const =0
- virtual **node_t** **kpa::NodeCollection::get** (unsigned index) const =0
- virtual **kpa::NodeCollection::~NodeCollection** ()

Variables

- size_t **kpa::edgeIterator_tag::n**
- void * **kpa::edgeIterator_tag::data**

9.2.1 Detailed Description

9.2.2 Function Documentation

9.2.2.1 get()

```
virtual  node_t kpa::NodeCollection::get (
             unsigned index ) const  [pure virtual]
```

9.2.2.2 isValid()

```
bool kpa::integer_t::isValid ( ) const
```

Check if the integer is not an invalid integer.

Returns

a boolean stating if the integer is valid

See also

integer_t::integer_t() (p. 39)

9.2.2.3 operator"!()

```
bool kpa::integer_t::operator!  ( ) const
```

Boolean negation operator for **kpa::integer_t** (p. 173).

Returns

a boolean stating if the current integer is not invalid and not equal to the value zero.

9.2.2.4 size()

```
virtual size_t kpa::NodeCollection::size ( ) const  [pure virtual]
```

9.2.2.5 ∼integer_t()

```
kpa::integer_t::∼integer_t ( )
```

Destructor of an integer.

9.2.2.6 ~NodeCollection()

```
virtual kpa::NodeCollection::~NodeCollection ( )  [inline], [virtual]
```

9.2.3 Variable Documentation

9.2.3.1 data

```
void* kpa::edgeIterator_tag::data
```

9.2.3.2 n

```
size_t kpa::edgeIterator_tag::n
```

9.3 Basic MIR types

Typedefs

- typedef struct mir_tFunction * **kpa::function_t**
- typedef size_t **kpa::sema_t**
- typedef struct mir_tNode * **kpa::node_t**
- typedef struct mir_tEdge * **kpa::edge_t**
- typedef union mir_tExpression * **kpa::expr_t**
- typedef struct mir_tBasicBlock * **kpa::bb_t**
- typedef const struct MemoryItem * **kpa::memitem_t**
- typedef struct NumericRange * **kpa::constraint_t**

Functions

- const char * **kpa::func_getName** (**function_t** func)

9.3.1 Detailed Description

Klocwork uses Medium level Intermediate Representation (MIR) of a program to detect path issues. MIR consists of control flow graph for every function. Nodes of control flow graph have information about operations performed by function. One statement of a program can be split into several MIR nodes and complicated control flow structures. MIR also provides access to semantics information for program types and variables.

9.3.2 Typedef Documentation

9.3.2.1 bb_t

```
typedef struct mir_tBasicBlock*  kpa::bb_t
```

Basic block descriptor type. Basic block is a sequence of nodes with the only entry node and the only exit node

9.3.2.2 constraint_t

```
typedef struct NumericRange*  kpa::constraint_t
```

Constraint on a memory item value

9.3.2.3 edge_t

```
typedef struct mir_tEdge*  kpa::edge_t
```

Opaque type for description of control flow edge in MIR graph

9.3.2.4 expr_t

```
typedef union mir_tExpression*  kpa::expr_t
```

Opaque type for description of MIR expression tree

9.3.2.5 function_t

```
typedef struct mir_tFunction*  kpa::function_t
```

Opaque type for function description in MIR graph

9.3.2.6 memitem_t

```
typedef const struct MemoryItem*  kpa::memitem_t
```

Memory item descriptor Memory item is an abstract memory location

9.3.2.7 node_t

```
typedef struct mir_tNode*  kpa::node_t
```

Opaque type for description of one control flow node in MIR graph

9.3.2.8 sema_t

```
typedef size_t  kpa::sema_t
```

opaque descriptor type for semantic information

9.3.3 Function Documentation

9.3.3.1 func_getName()

```
const char* kpa::func_getName (
              function_t func )
```

Get function name from function's description

9.4 Constructors for kpa::integer_t

Functions

- **kpa::integer_t::integer_t** ()
- **kpa::integer_t::integer_t** (signed char i)
- **kpa::integer_t::integer_t** (unsigned char i)
- **kpa::integer_t::integer_t** (signed short i)
- **kpa::integer_t::integer_t** (unsigned short i)
- **kpa::integer_t::integer_t** (signed int i)
- **kpa::integer_t::integer_t** (unsigned int i)
- **kpa::integer_t::integer_t** (signed long long i)
- **kpa::integer_t::integer_t** (unsigned long long i)
- **kpa::integer_t::integer_t** (const **integer_t** &other)

9.4.1 Detailed Description

9.4.2 Function Documentation

9.4.2.1 integer_t() [1/10]

```
kpa::integer_t::integer_t ( )
```

Create an invalid integer.

See also

integer_t::isValid (p. 35)

9.4.2.2 integer_t() [2/10]

```
kpa::integer_t::integer_t (
            signed char i )
```

Create an integer from a signed char value.

Parameters

i	the value to assign to this integer

9.4.2.3 integer_t() [3/10]

```
kpa::integer_t::integer_t (
            unsigned char i )
```

Create an integer from an unsigned char value.

Parameters

i	the value to assign to this integer

9.4.2.4 integer_t() [4/10]

```
kpa::integer_t::integer_t (
            signed short i )
```

Create an integer from a signed short value.

Parameters

i	the value to assign to this integer

9.4.2.5 integer_t() [5/10]

```
kpa::integer_t::integer_t (
            unsigned short i )
```

Create an integer from an unsigned short value.

Parameters

i	the value to assign to this integer

9.4.2.6 integer_t() [6/10]

```
kpa::integer_t::integer_t (
            signed int i )
```

Create an integer from a signed int value.

Parameters

i	the value to assign to this integer

9.4.2.7 integer_t() [7/10]

```
kpa::integer_t::integer_t (
            unsigned int i )
```

Create an integer from an unsigned int value.

Parameters

i	the value to assign to this integer

9.4.2.8 integer_t() [8/10]

```
kpa::integer_t::integer_t (
            signed long long i )
```

Create an integer from a signed long long value.

Parameters

i	the value to assign to this integer

9.4.2.9 integer_t() [9/10]

```
kpa::integer_t::integer_t (
            unsigned long long i )
```

Create an integer from an unsigned long long value.

Parameters

i	the value to assign to this integer

9.4.2.10 integer_t() [10/10]

```
kpa::integer_t::integer_t (
            const  integer_t & other )
```

Create an integer from another integer (copy constructor).

Parameters

other	the integer to copy

9.5 Assignment operators for kpa::integer_t

Functions

- **integer_t** & **kpa::integer_t::operator=** (signed char i)
- **integer_t** & **kpa::integer_t::operator=** (unsigned char i)
- **integer_t** & **kpa::integer_t::operator=** (signed short i)
- **integer_t** & **kpa::integer_t::operator=** (unsigned short i)
- **integer_t** & **kpa::integer_t::operator=** (signed int i)
- **integer_t** & **kpa::integer_t::operator=** (unsigned int i)
- **integer_t** & **kpa::integer_t::operator=** (signed long long i)
- **integer_t** & **kpa::integer_t::operator=** (unsigned long long i)
- **integer_t** & **kpa::integer_t::operator=** (const **integer_t** &other)

9.5.1 Detailed Description

9.5.2 Function Documentation

9.5.2.1 operator=() [1/9]

```
integer_t& kpa::integer_t::operator= (
            signed char i )
```

Assign a signed char value to an integer.

Parameters

i	the value to assign to the integer

Returns

a reference to 'this' to ease chaining.

9.5.2.2 operator=() [2/9]

```
integer_t& kpa::integer_t::operator= (
            unsigned char i )
```

Assign an unsigned char value to an integer.

Parameters

i	the value to assign to the integer

Returns

a reference to 'this' to ease chaining.

9.5.2.3 operator=() [3/9]

```
integer_t& kpa::integer_t::operator= (
            signed short i )
```

Assign a signed short value to an integer.

Parameters

i	the value to assign to the integer

Returns

a reference to 'this' to ease chaining.

9.5.2.4 operator=() [4/9]

```
integer_t& kpa::integer_t::operator= (
            unsigned short i )
```

Assign an unsigned short value to an integer.

Parameters

i	the value to assign to the integer

Returns

a reference to 'this' to ease chaining.

9.5.2.5 operator=() [5/9]

```
integer_t& kpa::integer_t::operator= (
            signed int i )
```

Assign a signed int value to an integer.

Parameters

i	the value to assign to the integer

Returns

a reference to 'this' to ease chaining.

9.5.2.6 operator=() [6/9]

```
integer_t& kpa::integer_t::operator= (
            unsigned int i )
```

Assign an unsigned int value to an integer.

Parameters

i	the value to assign to the integer

Returns

a reference to 'this' to ease chaining.

9.5.2.7 operator=() [7/9]

```
integer_t& kpa::integer_t::operator= (
            signed long long i )
```

Assign a signed long long value to an integer.

Parameters

i	the value to assign to the integer

Returns

a reference to 'this' to ease chaining.

9.5.2.8 operator=() [8/9]

```
integer_t& kpa::integer_t::operator= (
            unsigned long long i )
```

Assign an unsigned long long value to an integer.

Parameters

i	the value to assign to the integer

Returns

a reference to 'this' to ease chaining.

9.5.2.9 operator=() [9/9]

```
integer_t& kpa::integer_t::operator= (
            const  integer_t & other )
```

Assign the value of an integer to 'this' integer.

Parameters

other	the integer to copy

Returns

a reference to 'this' to ease chaining.

9.6 Conversion methods for kpa::integer_t

Functions

- long long **kpa::integer_t::getInt64** () const
- unsigned long long **kpa::integer_t::getUint64** () const
- char * **kpa::integer_t::toCharPtr** () const

9.6.1 Detailed Description

9.6.2 Function Documentation

9.6.2.1 getInt64()

```
long long kpa::integer_t::getInt64 ( ) const
```

Get the integer value as an int64 type.

Returns

the integer value as an int64 type with the following exceptions:

- If the value is bigger than INT64_MAX ('overflow'), then INT64_MAX is returned.
- If the value is smaller than INT64_MIN ('underflow'), then INT64_MIN is returned.
- If the value is invalid, then 0 is returned.

9.6.2.2 getUint64()

```
unsigned long long kpa::integer_t::getUint64 ( ) const
```

Get the integer value as an unsigned int64 type.

Returns

the integer value as an unsigned int64 type with the following exceptions:

- If the value is bigger than UINT64_MAX ('overflow'), then UINT64_MAX is returned.
- If the value is smaller than 0 ('underflow'), then 0 is returned.
- If the value is invalid, then 0 is returned.

9.6.2.3 toCharPtr()

```
char* kpa::integer_t::toCharPtr ( ) const
```

Get a C string representing the decimal number in the integer. It is the responsibility of the caller to free the allocated memory. This is useful for error messages.

Returns

a C string allocated on the heap representing the decimal number in the integer.

9.7 Cast methods for kpa::integer_t

Functions

- **integer_t kpa::integer_t::castToType** (**sema_t** si) const
- **integer_t kpa::integer_t::castToType** (**memitem_t** mi) const
- **integer_t kpa::integer_t::castToType** (**expr_t** expr) const

9.7.1 Detailed Description

9.7.2 Function Documentation

9.7.2.1 castToType() [1/3]

```
integer_t kpa::integer_t::castToType (
            sema_t si ) const
```

Cast an integer value to the type of the provided semantic information.

Parameters

si	the semantic information describing the target type.

Returns

the integer value casted to the type described by `si`. If the conversion failed, then the invalid integer is returned.

9.7.2.2 castToType() [2/3]

```
integer_t kpa::integer_t::castToType (
            memitem_t mi ) const
```

Cast an integer value to the type of the provided memory item.

Parameters

mi	the memory item providing the target type

Returns

the integer value casted to the type derived from `mi`. If the conversion failed, then the invalid integer is returned.

9.7.2.3 castToType() [3/3]

```
integer_t kpa::integer_t::castToType (
            expr_t expr ) const
```

Cast an integer value to the type of the provided expression.

Parameters

expr	the expression providing the target type.

Returns

the integer value casted to the type derived from `expr`. If the conversion failed, then the invalid integer is returned.

9.8 Binary arithmetic operators for kpa::integer_t

Functions

- **integer_t kpa::integer_t::operator+** (const **integer_t** &rhs) const
- **integer_t kpa::integer_t::operator-** (const **integer_t** &rhs) const
- **integer_t kpa::integer_t::operator∗** (const **integer_t** &rhs) const
- **integer_t kpa::integer_t::operator/** (const **integer_t** &rhs) const
- **integer_t kpa::integer_t::operator%** (const **integer_t** &rhs) const

9.8.1 Detailed Description

9.8.2 Function Documentation

9.8.2.1 operator%()

```
integer_t kpa::integer_t::operator% (
            const  integer_t & rhs ) const
```

Modulo operator between integers.

Parameters

rhs	the integer in the right-hand side of the modulo operator

Returns

an integer that represents the remainder of the division of the current integer by the right-hand side. An invalid integer is returned if the modulo cannot be performed (in particular, if the right-hand side has the value 0).

9.8.2.2 operator∗()

```
integer_t kpa::integer_t::operator* (
            const  integer_t & rhs ) const
```

Multiplication operator between integers.

Parameters

rhs	the integer in the right-hand side of the multiplication operator

Returns

an integer that represents the multiplication of the current integer by the right-hand side. An invalid integer is returned if the multiplication cannot be performed.

9.8.2.3 operator+()

```
integer_t kpa::integer_t::operator+ (
            const  integer_t & rhs ) const
```

Addition operator between integers.

Parameters

rhs	the integer in the right-hand side of the addition operator

Returns

an integer that represents the addition of the current integer with the right-hand side. An invalid integer is returned if the addition cannot be performed.

9.8.2.4 operator-()

```
integer_t kpa::integer_t::operator- (
            const  integer_t & rhs ) const
```

Subtraction operator between integers.

Parameters

rhs	the integer in the right-hand side of the subtraction operator

Returns

an integer that represents the subtraction of the right-hand side from the current integer. An invalid integer is returned if the subtraction cannot be performed.

9.8.2.5 operator/()

```
integer_t kpa::integer_t::operator/ (
            const  integer_t & rhs ) const
```

Division operator between integers.

Parameters

rhs	the integer in the right-hand side of the division operator

Returns

an integer that represents the division of the current integer by the right-hand side. An invalid integer is returned if the division cannot be performed (in particular, if the right-hand side has the value 0).

9.9 Unary arithmetic operators for kpa::integer_t

Functions

- **integer_t kpa::integer_t::operator-** () const
- **integer_t kpa::integer_t::operator+** () const
- **integer_t kpa::integer_t::operator~** () const

9.9.1 Detailed Description

9.9.2 Function Documentation

9.9.2.1 operator+()

```
integer_t kpa::integer_t::operator+ ( ) const
```

Unary positive operator for an integer.

Returns

a copy of the current integer.

9.9.2.2 operator-()

```
integer_t kpa::integer_t::operator- ( ) const
```

Unary minus operator for an integer.

Returns

an integer with the same absolute value as the current integer, but with the opposite sign. An invalid integer is returned if the operation cannot be performed.

9.9.2.3 operator~()

```
integer_t kpa::integer_t::operator~ ( ) const
```

Unary bitwise complement operator for an integer.

Returns

an integer that represents the bitwise complement of the current integer. An invalid integer is returned if the operation cannot be performed.

9.10 Pre/post-inc/decrement operators for kpa::integer_t

Functions

- **integer_t** & **kpa::integer_t::operator++** ()
- **integer_t** **kpa::integer_t::operator++** (int)
- **integer_t** & **kpa::integer_t::operator--** ()
- **integer_t** **kpa::integer_t::operator--** (int)

9.10.1 Detailed Description

9.10.2 Function Documentation

9.10.2.1 operator++() [1/2]

```
integer_t& kpa::integer_t::operator++ ( )
```

Pre-increment operator for an integer. After the call to this method, the integer will be incremented by one.

Returns

a reference to the current integer that has been incremented. An invalid integer is returned if the operation cannot be performed.

9.10.2.2 operator++() [2/2]

```
integer_t kpa::integer_t::operator++ (
            int  )
```

Post-increment operator for an integer. After the call to this method, the integer will be incremented by one.

Returns

a copy of the current integer before it was incremented. An invalid integer is returned if the operation cannot be performed.

9.10.2.3 operator--() [1/2]

```
integer_t& kpa::integer_t::operator-- ( )
```

Pre-decrement operator for an integer. After the call to this method, the integer will be decremented by one.

Returns

a reference to the current integer that has been decremented. An invalid integer is returned if the operation cannot be performed.

9.10.2.4 operator--() [2/2]

```
integer_t kpa::integer_t::operator-- (
            int  )
```

Post-decrement operator for an integer. After the call to this method, the integer will be decremented by one.

Returns

a copy of the current integer before it was decremented. An invalid integer is returned if the operation cannot be performed.

9.11 Arithmetic assignment operators for kpa::integer_t

Functions

- void **kpa::integer_t::operator+=** (const **integer_t** &rhs)
- void **kpa::integer_t::operator-=** (const **integer_t** &rhs)
- void **kpa::integer_t::operator∗=** (const **integer_t** &rhs)
- void **kpa::integer_t::operator/=** (const **integer_t** &rhs)
- void **kpa::integer_t::operator%=** (const **integer_t** &rhs)

9.11.1 Detailed Description

9.11.2 Function Documentation

9.11.2.1 operator%=()

```
void kpa::integer_t::operator%= (
            const  integer_t & rhs )
```

Modulo assignment operator for **kpa::integer_t** (p. 173). After the call to this method, the current integer represents the remainder of the division of the old value by the right-hand side. The resulting integer may become invalid if the modulo cannot be performed (e.g. if the right-hand side is the integer zero).

Parameters

rhs	the integer in the right-hand side of the modulo assignment operator

9.11.2.2 operator∗=()

```
void kpa::integer_t::operator*= (
            const  integer_t & rhs )
```

Multiplication assignment operator for **kpa::integer_t** (p. 173). The call to this method multiplies the current integer by the right-hand side. The resulting integer may become invalid if the multiplication cannot be performed.

Parameters

rhs	the integer in the right-hand side of the multiplication assignment operator

9.11.2.3 operator+=()

```
void kpa::integer_t::operator+= (
            const  integer_t & rhs )
```

Addition assignment operator for **kpa::integer_t** (p. 173). The call to this method adds the right-hand side to the current integer. The resulting integer may become invalid if the addition cannot be performed.

Parameters

rhs	the integer in the right-hand side of the addition assignment operator

9.11.2.4 operator-=()

```
void kpa::integer_t::operator-= (
            const  integer_t & rhs )
```

Subtraction assignment operator for **kpa::integer_t** (p. 173). The call to this method subtracts the right-hand side from the current integer. The resulting integer may become invalid if the subtraction cannot be performed.

Parameters

rhs	the integer in the right-hand side of the subtraction assignment operator

9.11.2.5 operator/=()

```
void kpa::integer_t::operator/= (
            const  integer_t & rhs )
```

Division assignment operator for **kpa::integer_t** (p. 173). The call to this method divides the current integer by the right-hand side. The resulting integer may become invalid if the division cannot be performed (e.g. if the right-hand side is the integer zero).

Parameters

rhs	the integer in the right-hand side of the division assignment operator

9.12 Relational operators for kpa::integer_t

Functions

- bool **kpa::integer_t::operator>** (const **integer_t** &rhs) const
- bool **kpa::integer_t::operator<** (const **integer_t** &rhs) const
- bool **kpa::integer_t::operator>=** (const **integer_t** &rhs) const
- bool **kpa::integer_t::operator<=** (const **integer_t** &rhs) const
- bool **kpa::integer_t::operator==** (const **integer_t** &rhs) const
- bool **kpa::integer_t::operator!=** (const **integer_t** &rhs) const

9.12.1 Detailed Description

9.12.2 Function Documentation

9.12.2.1 operator"!=()

```
bool  kpa::integer_t::operator!= (
            const  integer_t & rhs ) const
```

Check if the current integer is not equal to the right-hand side.

Parameters

rhs	the integer in the right-hand side of the comparison

Returns

a boolean stating if the current integer is not equal to the right-hand side. Note that invalid integers are considered equal to each other.

9.12.2.2 operator<()

```
bool kpa::integer_t::operator< (
            const  integer_t & rhs ) const
```

Check if the current integer is smaller than the right-hand side.

Parameters

rhs	the integer in the right-hand side of the comparison

Returns

a boolean stating if the current integer is smaller than the right-hand side. Note that an invalid integer is considered to be the smallest **kpa::integer_t** (p. 173) possible when comparing **integer_t** (p. 173).

9.12.2.3 operator<=()

```
bool kpa::integer_t::operator<= (
            const  integer_t & rhs ) const
```

Check if the current integer is smaller or equal than the right-hand side.

Parameters

rhs	the integer in the right-hand side of the comparison

Returns

a boolean stating if the current integer is smaller or equal than the right-hand side. Note that an invalid integer is considered to be the smallest **kpa::integer_t** (p. 173) possible when comparing **integer_t** (p. 173).

9.12.2.4 operator==()

```
bool kpa::integer_t::operator== (
            const  integer_t & rhs ) const
```

Check if the current integer is equal to the right-hand side.

Parameters

rhs	the integer in the right-hand side of the comparison

Returns

a boolean stating if the current integer is equal to the right-hand side. Note that invalid integers are considered equal to each other.

9.12.2.5 operator>()

```
bool kpa::integer_t::operator> (
            const  integer_t & rhs ) const
```

Check if the current integer is greater than the right-hand side.

Parameters

rhs	the integer in the right-hand side of the comparison

Returns

a boolean stating if the current integer is greater than the right-hand side. Note that an invalid integer is considered to be the smallest **kpa::integer_t** (p. 173) possible when comparing **integer_t** (p. 173).

9.12.2.6 operator>=()

```
bool kpa::integer_t::operator>= (
            const  integer_t & rhs ) const
```

Check if the current integer is greater or equal than the right-hand side.

Parameters

rhs	the integer in the right-hand side of the comparison

Returns

a boolean stating if the current integer is greater or equal than the right-hand side. Note that an invalid integer is considered to be the smallest **kpa::integer_t** (p. 173) possible when comparing **integer_t** (p. 173).

9.13 Binary bitwise operators for kpa::integer_t

Functions

- **integer_t kpa::integer_t::operator<<** (int shift) const
- **integer_t kpa::integer_t::operator>>** (int shift) const
- **integer_t kpa::integer_t::operator<<** (const **integer_t** &rhs) const
- **integer_t kpa::integer_t::operator>>** (const **integer_t** &rhs) const
- **integer_t kpa::integer_t::operator &** (const **integer_t** &rhs) const
- **integer_t kpa::integer_t::operator|** (const **integer_t** &rhs) const
- **integer_t kpa::integer_t::operator^** (const **integer_t** &rhs) const

9.13.1 Detailed Description

9.13.2 Function Documentation

9.13.2.1 operator &()

```
integer_t kpa::integer_t::operator& (
            const  integer_t & rhs ) const
```

Bitwise AND between **kpa::integer_t** (p. 173).

Parameters

rhs	the integer in the right-hand side of the operation

Returns

an integer representing the result of the bitwise AND between the current integer and the right-hand side. An invalid integer is returned if the operation cannot be performed.

9.13.2.2 operator<<() [1/2]

```
integer_t kpa::integer_t::operator<< (
            int shift ) const
```

Left shift a **kpa::integer_t** (p. 173) by a number of bits.

Parameters

shift	the number of bits to shift left the current integer

Returns

an integer representing the current integer shifted to the left by the right-hand side. An invalid integer is returned if the shift cannot be performed.

9.13.2.3 operator<<() [2/2]

```
integer_t kpa::integer_t::operator<< (
            const  integer_t & rhs ) const
```

Left shift a **kpa::integer_t** (p. 173) by a number of bits.

Parameters

rhs	an integer that represents the number of bits to shift left the current integer

Returns

an integer representing the current integer shifted to the left by the right-hand side. An invalid integer is returned if the shift cannot be performed.

9.13.2.4 operator>>() [1/2]

```
integer_t kpa::integer_t::operator>> (
            int shift ) const
```

Right shift a **kpa::integer_t** (p. 173) by a number of bits.

Parameters

shift	the number of bits to shift right the current integer

Returns

an integer representing the current integer shifted to the right by the right-hand side. An invalid integer is returned if the shift cannot be performed.

9.13.2.5 operator>>() [2/2]

```
integer_t kpa::integer_t::operator>> (
            const  integer_t & rhs ) const
```

Right shift a **kpa::integer_t** (p. 173) by a number of bits.

Parameters

rhs	an integer that represents the number of bits to shift right the current integer

Returns

an integer representing the current integer shifted to the right by the right-hand side. An invalid integer is returned if the shift cannot be performed.

9.13.2.6 operator^()

```
integer_t kpa::integer_t::operator^ (
            const  integer_t & rhs ) const
```

Bitwise XOR between **kpa::integer_t** (p. 173).

Parameters

rhs	the integer in the right-hand side of the operation

Returns

an integer representing the result of the bitwise XOR between the current integer and the right-hand side. An invalid integer is returned if the operation cannot be performed.

9.13.2.7 operator"|()

```
integer_t kpa::integer_t::operator| (
            const  integer_t & rhs ) const
```

Bitwise OR between **kpa::integer_t** (p. 173).

Parameters

rhs	the integer in the right-hand side of the operation

Returns

an integer representing the result of the bitwise OR between the current integer and the right-hand side. An invalid integer is returned if the operation cannot be performed.

9.14 Bitwise assignment operators for kpa::integer_t

Functions

- void **kpa::integer_t::operator<<=** (int shift)
- void **kpa::integer_t::operator>>=** (int shift)
- void **kpa::integer_t::operator &=** (const **integer_t** &rhs)
- void **kpa::integer_t::operator|=** (const **integer_t** &rhs)
- void **kpa::integer_t::operator^=** (const **integer_t** &rhs)

9.14.1 Detailed Description

9.14.2 Function Documentation

9.14.2.1 operator &=()

```
void kpa::integer_t::operator&= (
            const  integer_t & rhs )
```

Bitwise AND assignment by a **kpa::integer_t** (p. 173). After a call to this operator, the current integer will be the result of the bitwise AND between its old value and the value specified in the right-hand side. It becomes an invalid integer if the operation cannot be performed.

Parameters

rhs	the integer in the right-hand side of the operation

9.14.2.2 operator<<=()

```
void kpa::integer_t::operator<<= (
            int shift )
```

Left shift assignment by a number of bits. After a call to this operator, the current integer will be the result of the left shift of its old value by the number of bits specified in the shift. It becomes an invalid integer if the shift cannot be performed.

Parameters

shift	the number of bits to shift left the current integer

9.14.2.3 operator>>=()

```
void kpa::integer_t::operator>>= (
            int shift )
```

Right shift assignment by a number of bits. After a call to this operator, the current integer will be the result of the right shift of its old value by the number of bits specified in the shift. It becomes an invalid integer if the shift cannot be performed.

Parameters

shift	the number of bits to shift right the current integer

9.14.2.4 operator^=()

```
void kpa::integer_t::operator^= (
            const integer_t & rhs )
```

Bitwise XOR assignment by a **kpa::integer_t** (p. 173). After a call to this operator, the current integer will be the result of the bitwise XOR between its old value and the value specified in the right-hand side. It becomes an invalid integer if the operation cannot be performed.

Parameters

rhs	the integer in the right-hand side of the operation

9.14.2.5 operator"|=()

```
void kpa::integer_t::operator|= (
            const integer_t & rhs )
```

Bitwise OR assignment by a **kpa::integer_t** (p. 173). After a call to this operator, the current integer will be the result of the bitwise OR between its old value and the value specified in the right-hand side. It becomes an invalid integer if the operation cannot be performed.

Parameters

rhs	the integer in the right-hand side of the operation

9.15 Internal methods for kpa::integer_t (do not use)

Functions

- void **kpa::integer_t::setPimpl** (void *x)
- const void * **kpa::integer_t::getPimpl** () const

9.15.1 Detailed Description

9.15.2 Function Documentation

9.15.2.1 getPimpl()

```
const void* kpa::integer_t::getPimpl ( ) const
```

Note

Internal method. Do not use this method.

9.15.2.2 setPimpl()

```
void kpa::integer_t::setPimpl (
            void * x )
```

Note

Internal method. Do not use this method.

9.16 Binary operators when the left-hand side is a builtin integer and the right-hand side is a kpa::integer_t

Macros

- #define **DECLARE_INT_OP_INTEGER_T**(__X_OP__, __RETURN_TYPE__)

Functions

- **integer_t kpa::operator+** (const signed long long lhs_const, const **integer_t** &rhs)
- **integer_t kpa::operator+** (const signed int lhs_const, const **integer_t** &rhs)
- **integer_t kpa::operator+** (const unsigned long long lhs_const, const **integer_t** &rhs)
- **integer_t kpa::operator+** (const unsigned int lhs_const, const **integer_t** &rhs)
- **integer_t kpa::operator-** (const signed long long lhs_const, const **integer_t** &rhs)
- **integer_t kpa::operator-** (const signed int lhs_const, const **integer_t** &rhs)
- **integer_t kpa::operator-** (const unsigned long long lhs_const, const **integer_t** &rhs)
- **integer_t kpa::operator-** (const unsigned int lhs_const, const **integer_t** &rhs)
- **integer_t kpa::operator*** (const signed long long lhs_const, const **integer_t** &rhs)
- **integer_t kpa::operator*** (const signed int lhs_const, const **integer_t** &rhs)
- **integer_t kpa::operator*** (const unsigned long long lhs_const, const **integer_t** &rhs)
- **integer_t kpa::operator*** (const unsigned int lhs_const, const **integer_t** &rhs)
- **integer_t kpa::operator/** (const signed long long lhs_const, const **integer_t** &rhs)
- **integer_t kpa::operator/** (const signed int lhs_const, const **integer_t** &rhs)
- **integer_t kpa::operator/** (const unsigned long long lhs_const, const **integer_t** &rhs)
- **integer_t kpa::operator/** (const unsigned int lhs_const, const **integer_t** &rhs)
- **integer_t kpa::operator%** (const signed long long lhs_const, const **integer_t** &rhs)
- **integer_t kpa::operator%** (const signed int lhs_const, const **integer_t** &rhs)
- **integer_t kpa::operator%** (const unsigned long long lhs_const, const **integer_t** &rhs)
- **integer_t kpa::operator%** (const unsigned int lhs_const, const **integer_t** &rhs)
- **integer_t kpa::operator|** (const signed long long lhs_const, const **integer_t** &rhs)
- **integer_t kpa::operator|** (const signed int lhs_const, const **integer_t** &rhs)
- **integer_t kpa::operator|** (const unsigned long long lhs_const, const **integer_t** &rhs)
- **integer_t kpa::operator|** (const unsigned int lhs_const, const **integer_t** &rhs)
- **integer_t kpa::operator&** (const signed long long lhs_const, const **integer_t** &rhs)
- **integer_t kpa::operator&** (const signed int lhs_const, const **integer_t** &rhs)
- **integer_t kpa::operator&** (const unsigned long long lhs_const, const **integer_t** &rhs)
- **integer_t kpa::operator&** (const unsigned int lhs_const, const **integer_t** &rhs)
- **integer_t kpa::operator^** (const signed long long lhs_const, const **integer_t** &rhs)
- **integer_t kpa::operator^** (const signed int lhs_const, const **integer_t** &rhs)
- **integer_t kpa::operator^** (const unsigned long long lhs_const, const **integer_t** &rhs)
- **integer_t kpa::operator^** (const unsigned int lhs_const, const **integer_t** &rhs)
- bool **kpa::operator==** (const signed long long lhs_const, const **integer_t** &rhs)
- bool **kpa::operator==** (const signed int lhs_const, const **integer_t** &rhs)
- bool **kpa::operator==** (const unsigned long long lhs_const, const **integer_t** &rhs)
- bool **kpa::operator==** (const unsigned int lhs_const, const **integer_t** &rhs)
- bool **kpa::operator!=** (const signed long long lhs_const, const **integer_t** &rhs)
- bool **kpa::operator!=** (const signed int lhs_const, const **integer_t** &rhs)
- bool **kpa::operator!=** (const unsigned long long lhs_const, const **integer_t** &rhs)
- bool **kpa::operator!=** (const unsigned int lhs_const, const **integer_t** &rhs)
- bool **kpa::operator<** (const signed long long lhs_const, const **integer_t** &rhs)
- bool **kpa::operator<** (const signed int lhs_const, const **integer_t** &rhs)
- bool **kpa::operator<** (const unsigned long long lhs_const, const **integer_t** &rhs)

- bool **kpa::operator**< (const unsigned int lhs_const, const **integer_t** &rhs)
- bool **kpa::operator**> (const signed long long lhs_const, const **integer_t** &rhs)
- bool **kpa::operator**> (const signed int lhs_const, const **integer_t** &rhs)
- bool **kpa::operator**> (const unsigned long long lhs_const, const **integer_t** &rhs)
- bool **kpa::operator**> (const unsigned int lhs_const, const **integer_t** &rhs)
- bool **kpa::operator**<= (const signed long long lhs_const, const **integer_t** &rhs)
- bool **kpa::operator**<= (const signed int lhs_const, const **integer_t** &rhs)
- bool **kpa::operator**<= (const unsigned long long lhs_const, const **integer_t** &rhs)
- bool **kpa::operator**<= (const unsigned int lhs_const, const **integer_t** &rhs)
- bool **kpa::operator**>= (const signed long long lhs_const, const **integer_t** &rhs)
- bool **kpa::operator**>= (const signed int lhs_const, const **integer_t** &rhs)
- bool **kpa::operator**>= (const unsigned long long lhs_const, const **integer_t** &rhs)
- bool **kpa::operator**>= (const unsigned int lhs_const, const **integer_t** &rhs)

9.16.1 Detailed Description

9.16.2 Macro Definition Documentation

9.16.2.1 DECLARE_INT_OP_INTEGER_T

```
#define DECLARE_INT_OP_INTEGER_T(
               __X_OP__,
               __RETURN_TYPE__ )
```

Value:

```
extern KWAPI_DECLARE_CPP(__RETURN_TYPE__) operator __X_OP__ (const signed long long lhs_const, const
      integer_t& rhs); \
extern KWAPI_DECLARE_CPP(__RETURN_TYPE__) operator __X_OP__ (const signed int lhs_const, const integer_t&
      rhs); \
extern KWAPI_DECLARE_CPP(__RETURN_TYPE__) operator __X_OP__ (const unsigned long long lhs_const, const
      integer_t& rhs); \
extern KWAPI_DECLARE_CPP(__RETURN_TYPE__) operator __X_OP__ (const unsigned int lhs_const, const integer_t&
       rhs);
```

The following macro is used to declare all the operators of the form BUILTIN_INTEGER OP **kpa::integer_t** (p. 173).

E.g. the following case requires these declarations:

```
int x = 1;
integer_t wi = 2;
if (x < wi) { } // This requires definition of the operator < (int, integer_t)
```

9.16.3 Function Documentation

9.16.3.1 operator"!=() [1/4]

```
bool kpa::operator!= (
            const signed long long lhs_const,
            const  integer_t & rhs )
```

9.16.3.2 operator"!=() [2/4]

```
bool kpa::operator!= (
            const signed int lhs_const,
            const  integer_t & rhs )
```

9.16.3.3 operator"!=() [3/4]

```
bool kpa::operator!= (
            const unsigned long long lhs_const,
            const  integer_t & rhs )
```

9.16.3.4 operator"!=() [4/4]

```
bool kpa::operator!= (
            const unsigned int lhs_const,
            const  integer_t & rhs )
```

9.16.3.5 operator%() [1/4]

```
 integer_t kpa::operator% (
            const signed long long lhs_const,
            const  integer_t & rhs )
```

9.16.3.6 operator%() [2/4]

```
 integer_t kpa::operator% (
            const signed int lhs_const,
            const  integer_t & rhs )
```

9.16.3.7 operator%() [3/4]

```
integer_t kpa::operator% (
            const unsigned long long lhs_const,
            const  integer_t & rhs )
```

9.16.3.8 operator%() [4/4]

```
integer_t kpa::operator% (
            const unsigned int lhs_const,
            const  integer_t & rhs )
```

9.16.3.9 operator&() [1/4]

```
integer_t kpa::operator & (
            const signed long long lhs_const,
            const  integer_t & rhs )
```

9.16.3.10 operator&() [2/4]

```
integer_t kpa::operator & (
            const signed int lhs_const,
            const  integer_t & rhs )
```

9.16.3.11 operator&() [3/4]

```
integer_t kpa::operator & (
            const unsigned long long lhs_const,
            const  integer_t & rhs )
```

9.16.3.12 operator&() [4/4]

```
integer_t kpa::operator & (
            const unsigned int lhs_const,
            const  integer_t & rhs )
```

9.16.3.13 operator*() [1/4]

```
integer_t kpa::operator* (
            const signed long long lhs_const,
            const  integer_t & rhs )
```

9.16.3.14 operator*() [2/4]

```
integer_t kpa::operator* (
            const signed int lhs_const,
            const  integer_t & rhs )
```

9.16.3.15 operator*() [3/4]

```
integer_t kpa::operator* (
            const unsigned long long lhs_const,
            const  integer_t & rhs )
```

9.16.3.16 operator*() [4/4]

```
integer_t kpa::operator* (
            const unsigned int lhs_const,
            const  integer_t & rhs )
```

9.16.3.17 operator+() [1/4]

```
integer_t kpa::operator+ (
            const signed int lhs_const,
            const  integer_t & rhs )
```

9.16.3.18 operator+() [2/4]

```
integer_t kpa::operator+ (
            const unsigned int lhs_const,
            const  integer_t & rhs )
```

9.16.3.19 operator+() [3/4]

```
integer_t kpa::operator+ (
            const signed long long lhs_const,
            const  integer_t & rhs )
```

9.16.3.20 operator+() [4/4]

```
integer_t kpa::operator+ (
            const unsigned long long lhs_const,
            const  integer_t & rhs )
```

9.16.3.21 operator-() [1/4]

```
integer_t kpa::operator- (
            const unsigned int lhs_const,
            const  integer_t & rhs )
```

9.16.3.22 operator-() [2/4]

```
integer_t kpa::operator- (
            const signed long long lhs_const,
            const  integer_t & rhs )
```

9.16.3.23 operator-() [3/4]

```
integer_t kpa::operator- (
            const signed int lhs_const,
            const  integer_t & rhs )
```

9.16.3.24 operator-() [4/4]

```
integer_t kpa::operator- (
            const unsigned long long lhs_const,
            const  integer_t & rhs )
```

9.16.3.25 operator/() [1/4]

```
integer_t kpa::operator/ (
            const unsigned int lhs_const,
            const  integer_t & rhs )
```

9.16.3.26 operator/() [2/4]

```
integer_t kpa::operator/ (
            const signed long long lhs_const,
            const  integer_t & rhs )
```

9.16.3.27 operator/() [3/4]

```
integer_t kpa::operator/ (
            const signed int lhs_const,
            const  integer_t & rhs )
```

9.16.3.28 operator/() [4/4]

```
integer_t kpa::operator/ (
            const unsigned long long lhs_const,
            const  integer_t & rhs )
```

9.16.3.29 operator<() [1/4]

```
bool kpa::operator< (
            const unsigned long long lhs_const,
            const  integer_t & rhs )
```

9.16.3.30 operator<() [2/4]

```
bool kpa::operator< (
            const signed long long lhs_const,
            const  integer_t & rhs )
```

9.16.3.31 operator<() [3/4]

```
bool kpa::operator< (
            const signed int lhs_const,
            const  integer_t & rhs )
```

9.16.3.32 operator<() [4/4]

```
bool kpa::operator< (
            const unsigned int lhs_const,
            const  integer_t & rhs )
```

9.16.3.33 operator<=() [1/4]

```
bool kpa::operator<= (
            const unsigned long long lhs_const,
            const  integer_t & rhs )
```

9.16.3.34 operator<=() [2/4]

```
bool kpa::operator<= (
            const signed int lhs_const,
            const  integer_t & rhs )
```

9.16.3.35 operator<=() [3/4]

```
bool kpa::operator<= (
            const unsigned int lhs_const,
            const  integer_t & rhs )
```

9.16.3.36 operator<=() [4/4]

```
bool kpa::operator<= (
            const signed long long lhs_const,
            const  integer_t & rhs )
```

9.16.3.37 operator==() [1/4]

```
bool kpa::operator== (
            const signed long long lhs_const,
            const  integer_t & rhs )
```

9.16.3.38 operator==() [2/4]

```
bool kpa::operator== (
            const signed int lhs_const,
            const  integer_t & rhs )
```

9.16.3.39 operator==() [3/4]

```
bool kpa::operator== (
            const unsigned int lhs_const,
            const  integer_t & rhs )
```

9.16.3.40 operator==() [4/4]

```
bool kpa::operator== (
            const unsigned long long lhs_const,
            const  integer_t & rhs )
```

9.16.3.41 operator>() [1/4]

```
bool kpa::operator> (
            const signed int lhs_const,
            const  integer_t & rhs )
```

9.16.3.42 operator>() [2/4]

```
bool kpa::operator> (
            const signed long long lhs_const,
            const  integer_t & rhs )
```

9.16.3.43 operator>() [3/4]

```
bool kpa::operator> (
            const unsigned int lhs_const,
            const  integer_t & rhs )
```

9.16.3.44 operator>() [4/4]

```
bool kpa::operator> (
            const unsigned long long lhs_const,
            const  integer_t & rhs )
```

9.16.3.45 operator>=() [1/4]

```
bool kpa::operator>= (
            const unsigned int lhs_const,
            const  integer_t & rhs )
```

9.16.3.46 operator>=() [2/4]

```
bool kpa::operator>= (
            const signed long long lhs_const,
            const  integer_t & rhs )
```

9.16.3.47 operator>=() [3/4]

```
bool kpa::operator>= (
            const unsigned long long lhs_const,
            const  integer_t & rhs )
```

9.16.3.48 operator>=() [4/4]

```
bool kpa::operator>= (
            const signed int lhs_const,
            const  integer_t & rhs )
```

9.16.3.49 operator^() [1/4]

```
integer_t kpa::operator^ (
            const signed long long lhs_const,
            const  integer_t & rhs )
```

9.16.3.50 operator^() [2/4]

```
integer_t kpa::operator^ (
            const unsigned long long lhs_const,
            const  integer_t & rhs )
```

9.16.3.51 operator^() [3/4]

```
integer_t kpa::operator^ (
            const unsigned int lhs_const,
            const  integer_t & rhs )
```

9.16.3.52 operator^() [4/4]

```
integer_t kpa::operator^ (
            const signed int lhs_const,
            const  integer_t & rhs )
```

9.16.3.53 operator"|() [1/4]

```
integer_t kpa::operator| (
            const signed int lhs_const,
            const  integer_t & rhs )
```

9.16.3.54 operator"|() [2/4]

```
integer_t kpa::operator| (
            const unsigned int lhs_const,
            const  integer_t & rhs )
```

9.16.3.55 operator"" | () [3/4]

```
integer_t kpa::operator| (
            const unsigned long long lhs_const,
            const  integer_t & rhs )
```

9.16.3.56 operator"" | () [4/4]

```
integer_t kpa::operator| (
            const signed long long lhs_const,
            const  integer_t & rhs )
```

9.17 Extension points for Path Analysis

Typedefs

- typedef void(* **kpa::functionHook_t**) (**function_t**, ThreadContext &)

Functions

- void **kpa::registerFunctionHook** (**functionHook_t** function_hook)
- void **kpa::registerKBGeneratorFunctionHook** (**functionHook_t** function_hook)

9.17.1 Detailed Description

9.17.2 Typedef Documentation

9.17.2.1 functionHook_t

```
typedef void(* kpa::functionHook_t) ( function_t, ThreadContext &)
```

API hook type: 'for each function'

9.17.3 Function Documentation

9.17.3.1 registerFunctionHook()

```
void kpa::registerFunctionHook (
              functionHook_t function_hook )
```

Hook function to analyze each function in the source code

9.17.3.2 registerKBGeneratorFunctionHook()

```
void kpa::registerKBGeneratorFunctionHook (
              functionHook_t function_hook )
```

Hook function to create knowledge base for each function in the source code

9.18 Working with MIR nodes

Modules

- **Checking types of MIR node**
- **Checking additional node properties**

Functions

- **expr_t kpa::node_getReadExpression** (**node_t** n)
- **expr_t kpa::node_getWrittenExpression** (**node_t** n)
- int **kpa::node_getOutDegree** (**node_t** node)
- int **kpa::node_getInDegree** (**node_t** node)

9.18.1 Detailed Description

9.18.2 Function Documentation

9.18.2.1 node_getInDegree()

```
int kpa::node_getInDegree (
              node_t node )
```

Get number of edges coming to node

9.18.2.2 node_getOutDegree()

```
int kpa::node_getOutDegree (
              node_t node )
```

Get number of edges outgoing from node

9.18.2.3 node_getReadExpression()

```
expr_t kpa::node_getReadExpression (
              node_t n )
```

Get an expression tree for right hand side of an assignment in node

9.18.2.4 node_getWrittenExpression()

```
expr_t kpa::node_getWrittenExpression (
            node_t n )
```

Get an expression tree for left hand side of an assignment in node

Returns

- NULL if there is no left hand side expression in MIR;
- expression tree for temporary variable if expression is complex and requires more than one MIR node
  - Example1:
    a+b+c will have 2 MIR nodes temp=a+b; temp+c;
    node_getReadExpression() (p. 81) will return expression tree "temp" for the first node and NULL for the second
  - Example2:
    a[b+c]=f() will have 2 MIR nodes temp=b+c; a[temp]=f();
    node_getReadExpression() (p. 81) will return expression tree "temp" for the first node and "a[temp]" for the second.
- expression tree for the left hand side of an assignment if there is a left hand side in the node

9.19 Checking types of MIR node

Functions

- int **kpa::node_isExpression** (**node_t** node)
- int **kpa::node_isSwitch** (**node_t** node)
- int **kpa::node_isConditionalBranch** (**node_t** node)
- int **kpa::node_isLeaf** (**node_t** node)

9.19.1 Detailed Description

9.19.2 Function Documentation

9.19.2.1 node_isConditionalBranch()

```
int kpa::node_isConditionalBranch (
              node_t node )
```

Check if node is 'if' branching node

9.19.2.2 node_isExpression()

```
int kpa::node_isExpression (
              node_t node )
```

Check whether node is a single-assignment expression

9.19.2.3 node_isLeaf()

```
int kpa::node_isLeaf (
              node_t node )
```

Check if node is leaf node in the control-flow graph (i.e. has no outgoing edges)

9.19.2.4 node_isSwitch()

```
int kpa::node_isSwitch (
              node_t node )
```

Check if node is a switch header node

9.20 Checking additional node properties

Functions

- int **kpa::node_isReturn** (**node_t** node)
- int **kpa::node_isBreak** (**node_t** node)
- int **kpa::node_isContinue** (**node_t** node)
- int **kpa::node_isInitialization** (**node_t** node)
- int **kpa::node_isThrow** (**node_t** node)

9.20.1 Detailed Description

9.20.2 Function Documentation

9.20.2.1 node_isBreak()

```
int kpa::node_isBreak (
              node_t node )
```

Check if node is a 'break' statement

9.20.2.2 node_isContinue()

```
int kpa::node_isContinue (
              node_t node )
```

Check if node is a 'continue' statement

9.20.2.3 node_isInitialization()

```
int kpa::node_isInitialization (
              node_t node )
```

Check if node is an initialization

9.20.2.4 node_isReturn()

```
int kpa::node_isReturn (
              node_t node )
```

Check if node is 'return' statement

9.20.2.5 node_isThrow()

```
int kpa::node_isThrow (
              node_t node )
```

Check if node is a 'throw' statement

9.21 Working with MIR edges

Classes

- struct **kpa::edgeIterator_tag**

Typedefs

- typedef struct **kpa::edgeIterator_tag** **kpa::edgeIterator_t**

Functions

- **edgeIterator_t** **kpa::node_getInEdgeSet** (**node_t** node)
- **edgeIterator_t** **kpa::node_getOutEdgeSet** (**node_t** node)
- int **kpa::edgeIterator_valid** (**edgeIterator_t** it)
- void **kpa::edgeIterator_next** (**edgeIterator_t** ∗it)
- **edge_t** **kpa::edgeIterator_value** (**edgeIterator_t** it)
- int **kpa::edge_getKind** (**edge_t** edge)
- **node_t** **kpa::edge_getStartNode** (**edge_t** edge)
- **node_t** **kpa::edge_getEndNode** (**edge_t** edge)

- const int **kpa::EDGE_TRUE**
- const int **kpa::EDGE_FALSE**
- const int **kpa::EDGE_CONDITIONAL**
- const int **kpa::EDGE_UNCONDITIONAL**

9.21.1 Detailed Description

9.21.2 Typedef Documentation

9.21.2.1 edgeIterator_t

```
typedef struct  kpa::edgeIterator_tag  kpa::edgeIterator_t
```

9.21.3 Function Documentation

9.21.3.1 edge_getEndNode()

```
node_t kpa::edge_getEndNode (
            edge_t edge )
```

Get end node of the edge

9.21.3.2 edge_getKind()

```
int kpa::edge_getKind (
              edge_t edge )
```

Get edge kind

9.21.3.3 edge_getStartNode()

```
node_t kpa::edge_getStartNode (
              edge_t edge )
```

Get start node of the edge

9.21.3.4 edgeIterator_next()

```
void kpa::edgeIterator_next (
              edgeIterator_t * it )
```

Get next edge in the edge set pointed by iterator

9.21.3.5 edgeIterator_valid()

```
int kpa::edgeIterator_valid (
              edgeIterator_t it )
```

Check if iterator points to valid edge in the edges set.

9.21.3.6 edgeIterator_value()

```
edge_t kpa::edgeIterator_value (
              edgeIterator_t it )
```

Get edge from edgeset, to which iterator currently points Typical edge set access code would look like this: `edgeIterator_t ei; for (ei = node_getInEdgeSet(node); edgeIterator_valid(ei); edgeIterator_next(&ei)) { edge_t current_edge = edgeIterator_value(ei); }`

9.21.3.7 node_getInEdgeSet()

```
edgeIterator_t kpa::node_getInEdgeSet (
              node_t node )
```

Get edge set iterator for edges coming to node

9.21.3.8 node_getOutEdgeSet()

```
edgeIterator_t kpa::node_getOutEdgeSet (
            node_t node )
```

Get edge set iterator for edges outgoing from node

9.21.4 Variable Documentation

9.21.4.1 EDGE_CONDITIONAL

```
const int kpa::EDGE_CONDITIONAL
```

edge representing conditional branch of 'switch' statement

9.21.4.2 EDGE_FALSE

```
const int kpa::EDGE_FALSE
```

edge representing false branch of conditional statement

9.21.4.3 EDGE_TRUE

```
const int kpa::EDGE_TRUE
```

MirEdgeKinds Edge kinds edge representing true branch of conditional statement

9.21.4.4 EDGE_UNCONDITIONAL

```
const int kpa::EDGE_UNCONDITIONAL
```

edge that is always traversed if its start node is executed

9.22 MIR expression trees

Modules

- **Operation codes in MIR expressions**
- **Working with memory items**
- **Constraints on memory item values**
- **Positions in MIR**
- **Trace and events**
- **Issue reporting functions**
- **Semantic information in MIR**

Classes

- class **kpa::NodeCollection**

Typedefs

- typedef **Ptr**< **NodeCollection** > **kpa::NodeCollectionPtr**

Functions

- int **kpa::expr_isCallTo** (**expr_t** mir_expr, const char *func_name)
- int **kpa::expr_isCallToQualified** (**expr_t** mir_expr, const char *func_name)
- const char * **kpa::expr_getCallName** (**expr_t** expr, ThreadContext &threadContext)
- const char * **kpa::expr_getCallQualifiedName** (**expr_t** expr, ThreadContext &threadContext)
- const char * **kpa::expr_getCallFBKBName** (**expr_t** expr)
- int **kpa::expr_getNumberOfArguments** (**expr_t** call_expr)
- **expr_t** **kpa::expr_getCallArgument** (**expr_t** call_expr, int argnum)
- **memitem_t** **kpa::expr_getMemitem** (**expr_t** mir_expr, ThreadContext &threadContext)
- int **kpa::expr_isVariable** (**expr_t** expr)
- int **kpa::expr_isFunction** (**expr_t** expr)
- int **kpa::expr_isConstantValue** (**expr_t** expr)
- int **kpa::expr_isIntegerConstant** (**expr_t** expr)
- long long **kpa::expr_getIntegerConstantValue** (**expr_t** expr, int *error_flag)
- **integer_t** **kpa::expr_getIntegerConstantValue** (**expr_t** expr)
- int **kpa::expr_isStringConstant** (**expr_t** expr)
- char * **kpa::expr_getStringConstantValue** (**expr_t** expr)
- int **kpa::expr_isFloatConstant** (**expr_t** expr)
- long double **kpa::expr_getFloatConstantValue** (**expr_t** expr, int *error_flag)
- int **kpa::expr_isSizeofConstant** (**expr_t** expr)
- int **kpa::expr_getSizeofConstantValue** (**expr_t** expr, int *error_flag)
- int **kpa::expr_isAddress** (**expr_t** expr)
- int **kpa::expr_isIndex** (**expr_t** expr)
- int **kpa::expr_isDereference** (**expr_t** expr)
- int **kpa::expr_isField** (**expr_t** expr)
- int **kpa::expr_isMember** (**expr_t** expr)
- int **kpa::expr_isCall** (**expr_t** expr)
- int **kpa::expr_isBinaryOperation** (**expr_t** expr)
- int **kpa::expr_isUnaryOperation** (**expr_t** expr)
- int **kpa::expr_isTemporaryRegister** (**expr_t** expr)

- **NodeCollectionPtr kpa::getDefinitionNodeForTemporary** (**expr_t** temp, **node_t** n, ThreadContext &threadContext)
- int **kpa::expr_isParameter** (**expr_t** expr)
- **expr_t kpa::expr_getUnaryOperand** (**expr_t** unary_expr)
- **expr_t kpa::expr_getBinaryOperand1** (**expr_t** binary_expr)
- **expr_t kpa::expr_getBinaryOperand2** (**expr_t** binary_expr)
- **expr_t kpa::expr_getAddressed** (**expr_t** addr_expr)
- int **kpa::expr_getParameterNumber** (**expr_t** param_expr)
- **expr_t kpa::expr_getDereferenced** (**expr_t** deref_expr)
- **expr_t kpa::expr_getIndexBase** (**expr_t** index_expr)
- **expr_t kpa::expr_getIndexOffset** (**expr_t** index_expr)
- **constraint_t kpa::expr_getOffsetValue** (**node_t** node, **expr_t** index_expr, int ∗error_flag, ThreadContext &threadContext)
- **expr_t kpa::expr_getFieldBase** (**expr_t** field_expr)
- **expr_t kpa::expr_getFieldMember** (**expr_t** field_expr)
- **expr_t kpa::expr_getCalled** (**expr_t** call_expr)

9.22.1 Detailed Description

9.22.2 Typedef Documentation

9.22.2.1 NodeCollectionPtr

```
typedef Ptr< NodeCollection> kpa::NodeCollectionPtr
```

9.22.3 Function Documentation

9.22.3.1 expr_getAddressed()

```
expr_t kpa::expr_getAddressed (
            expr_t addr_expr )
```

Get addressed expression from ampersand operation. I.e. for expression '&e', returns expression 'e'. If expression is not an operation of address taking, returns 0.

9.22.3.2 expr_getBinaryOperand1()

```
expr_t kpa::expr_getBinaryOperand1 (
            expr_t binary_expr )
```

Get first operand of a binary operation

9.22.3.3 expr_getBinaryOperand2()

```
expr_t kpa::expr_getBinaryOperand2 (
            expr_t binary_expr )
```

Get second operand of a binary operation

9.22.3.4 expr_getCallArgument()

```
expr_t kpa::expr_getCallArgument (
            expr_t call_expr,
           int argnum )
```

Get expression for an actual argument of a call

Parameters

call_expr	call expression to extract actual argument from
argnum	number of an argument (counted from 1)

Returns

actual argument expression or 0 if call_expr is not a call expression or argnum exceeds number of actual arguments in a call expression

9.22.3.5 expr_getCalled()

```
expr_t kpa::expr_getCalled (
            expr_t call_expr )
```

Return called expression of a call (usually a function or method name but may be variable (pointer to function) or field expression, index expression etc)

Returns

0 if 'call_expr' is not a call expression

9.22.3.6 expr_getCallFBKBName()

```
const char* kpa::expr_getCallFBKBName (
            expr_t expr )
```

get name of the called function for use with FBKB

Returns

FBKB name of the called function if 'expr' is a function call and 0 otherwise

9.22.3.7 expr_getCallName()

```
const char* kpa::expr_getCallName (
              expr_t expr,
             ThreadContext & threadContext )
```

get name of the called function

Parameters

expr	MIR expression to get name for
threadContext	reference to ThreadContext object maintained by analysis engine

Returns

name of the called function if 'expr' is a function call and 0 otherwise

9.22.3.8 expr_getCallQualifiedName()

```
const char* kpa::expr_getCallQualifiedName (
              expr_t expr,
             ThreadContext & threadContext )
```

get fully qualified name of the called function

Parameters

expr	MIR expression to get name for
threadContext	reference to ThreadContext object maintained by analysis engine

Returns

name of the called function if 'expr' is a function call and 0 otherwise

9.22.3.9 expr_getDereferenced()

```
expr_t kpa::expr_getDereferenced (
              expr_t deref_expr )
```

Return dereferenced expression, i.e. for '∗p', return 'p'

Returns

derefernced expression, or 0 if 'deref_expr' is not a dereference

9.22.3.10 expr_getFieldBase()

```
expr_t kpa::expr_getFieldBase (
            expr_t field_expr )
```

Return base expression of the field expression, i.e. for 'a->b', return expression for 'a'

9.22.3.11 expr_getFieldMember()

```
expr_t kpa::expr_getFieldMember (
            expr_t field_expr )
```

Return member expression of the field expression, i.e. for 'a->b', return expression for 'b'

9.22.3.12 expr_getFloatConstantValue()

```
long double kpa::expr_getFloatConstantValue (
            expr_t expr,
           int * error_flag )
```

Get value of constant of float or double type

Parameters

expr	float constant expression
error_flag	out parameter: set to 1 if 'expr' is not a float constant

Returns

float constant value

9.22.3.13 expr_getIndexBase()

```
expr_t kpa::expr_getIndexBase (
            expr_t index_expr )
```

Return base pointer for index expression

Returns

An expression of base pointer, or 0 if 'index_expr' is not an index expression

9.22.3.14 expr_getIndexOffset()

```
expr_t kpa::expr_getIndexOffset (
            expr_t index_expr )
```

Return offset for index expression

Returns

An expression of offset in index expression, or 0 if 'index_expr' is not an index expression

9.22.3.15 expr_getIntegerConstantValue() [1/2]

```
long long kpa::expr_getIntegerConstantValue (
            expr_t expr,
           int * error_flag )
```

Get signed integer value from MIR expression tree for a constant.

Parameters

expr	expression tree
error_flag	pointed value will be set to 0 if no error was encountered, -1 otherwise (if 'expr' is a null pointer, or not an integer constant).

Deprecated This function is only dealing with 64 bit signed integers.
It is replaced by **expr_getIntegerConstantValue(expr_t expr)** (p. 93).

```
// Example of the proposed migration.
// Old code:
void f(expr_t expr)
{
    int err;
    long long x = expr_getIntegerConstantValue(expr, &err);
    if (!err) {
        ...
    }
}

// New code:
void f(expr_t expr)
{
    integer_t x = expr_getIntegerConstantValue(expr);
    if (x.isValid()) {
        ...
    }
}
```

9.22.3.16 expr_getIntegerConstantValue() [2/2]

```
integer_t kpa::expr_getIntegerConstantValue (
            expr_t expr )
```

Get the integer value from an MIR expression tree for a constant.

Parameters

expr	expression tree

Returns

the integer value for this expression if the expression is an integer constant. Otherwise, the invalid integer is returned (this can be detected by calling **kpa::integer_t::isValid()** (p. 35)).

9.22.3.17 expr_getMemitem()

```
memitem_t kpa::expr_getMemitem (
              expr_t mir_expr,
             ThreadContext & threadContext )
```

Get memory item from MIR expression tree, if possible

Parameters

expr	MIR expression to get the memory item for
threadContext	reference to ThreadContext object maintained by analysis engine

Returns

0 if extraction was not successful, memory item descriptor otherwise

9.22.3.18 expr_getNumberOfArguments()

```
int kpa::expr_getNumberOfArguments (
              expr_t call_expr )
```

Get number of actual arguments in a call expression

9.22.3.19 expr_getOffsetValue()

```
constraint_t kpa::expr_getOffsetValue (
              node_t node,
              expr_t index_expr,
             int * error_flag,
             ThreadContext & threadContext )
```

Calculate and return value of the offset in index expression

Parameters

node	node containing the index expression (for advanced calculation)
index_expr	index expression
error_flag	out parameter: set to -1 if there was a failure in calculating the value of index expression 'expr'
threadContext	reference to ThreadContext object maintained by analysis engine

Returns

constraint_t calculated constraint of offset in index expression Returns newly allocated copy of a constraint which must be later freed by a call to 'constraint_delete'

9.22.3.20 expr_getParameterNumber()

```
int kpa::expr_getParameterNumber (
              expr_t param_expr )
```

Get parameter number of expression associated with function parameter

Returns

function parameter number (1 for the first parameter, 0 for this), -1 if 'param_expr' is not a function parameter

9.22.3.21 expr_getSizeofConstantValue()

```
int kpa::expr_getSizeofConstantValue (
              expr_t expr,
             int * error_flag )
```

Get integer value of sizeof operation

Parameters

expr	sizeof expression
error_flag	out parameter: set to 1 if 'expr' is not a sizeof expression or size could not be determined at compile-time

Returns

sizeof value

9.22.3.22 expr_getStringConstantValue()

```
char* kpa::expr_getStringConstantValue (
              expr_t expr )
```

Get string value copy from constant string expression. Caller is responsible for 'free'ing result value.

Returns

string constant allocated in heap, or 0 if 'expr' is not a string constant.

9.22.3.23 expr_getUnaryOperand()

```
 expr_t kpa::expr_getUnaryOperand (
              expr_t unary_expr )
```

Get operand of an unary operation

9.22.3.24 expr_isAddress()

```
int kpa::expr_isAddress (
              expr_t expr )
```

Check whether MIR expression subtree represents taking of address ()

9.22.3.25 expr_isBinaryOperation()

```
int kpa::expr_isBinaryOperation (
              expr_t expr )
```

Check whether MIR expression subtree represents binary operation

9.22.3.26 expr_isCall()

```
int kpa::expr_isCall (
              expr_t expr )
```

Check whether MIR expression subtree represents a call

9.22.3.27 expr_isCallTo()

```
int kpa::expr_isCallTo (
              expr_t mir_expr,
             const char * func_name )
```

Check that expression tree is a call to particular function

Parameters

mir_expr	MIR expression tree to check
func_name	function name

Returns

0 if mir_expr is not a call to func_name, 1 otherwise

9.22.3.28 expr_isCallToQualified()

```
int kpa::expr_isCallToQualified (
              expr_t mir_expr,
             const char * func_name )
```

Check that expression tree is a call to particular function

Parameters

mir_expr	MIR expression tree to check
func_name	function name

Returns

0 if mir_expr is not a call to qualified func_name, 1 otherwise

9.22.3.29 expr_isConstantValue()

```
int kpa::expr_isConstantValue (
              expr_t expr )
```

Check whether MIR expression subtree represents constant value

9.22.3.30 expr_isDereference()

```
int kpa::expr_isDereference (
              expr_t expr )
```

Check whether MIR expression subtree represents dereference

9.22.3.31 expr_isField()

```
int kpa::expr_isField (
              expr_t expr )
```

Check whether MIR expression subtree represents field operation (a->b)

9.22.3.32 expr_isFloatConstant()

```
int kpa::expr_isFloatConstant (
              expr_t expr )
```

Check if expression is a constant of float, double, or long double type

9.22.3.33 expr_isFunction()

```
int kpa::expr_isFunction (
              expr_t expr )
```

Check whether MIR expression subtree represents function

9.22.3.34 expr_isIndex()

```
int kpa::expr_isIndex (
              expr_t expr )
```

Check whether MIR expression subtree represents indexing of array or pointer

9.22.3.35 expr_isIntegerConstant()

```
int kpa::expr_isIntegerConstant (
              expr_t expr )
```

Check if expression is a constant of one of integer types

9.22.3.36 expr_isMember()

```
int kpa::expr_isMember (
              expr_t expr )
```

Check whether MIR expression subtree is a member in the field operation. For example 'b' in expression 'a->b'

9.22.3.37 expr_isParameter()

```
int kpa::expr_isParameter (
              expr_t expr )
```

Check whether MIR expression subtree represents function parameter

9.22.3.38 expr_isSizeofConstant()

```
int kpa::expr_isSizeofConstant (
              expr_t expr )
```

Check if expression is a constant result of sizeof

9.22.3.39 expr_isStringConstant()

```
int kpa::expr_isStringConstant (
              expr_t expr )
```

Check if expression is a constant string

9.22.3.40 expr_isTemporaryRegister()

```
int kpa::expr_isTemporaryRegister (
              expr_t expr )
```

Check whether MIR expression subtree represents temporary value, generated by MIR converter to normalize MIR to SSA (static single assignment form)

9.22.3.41 expr_isUnaryOperation()

```
int kpa::expr_isUnaryOperation (
              expr_t expr )
```

Check whether MIR expression subtree represents unary operation

9.22.3.42 expr_isVariable()

```
int kpa::expr_isVariable (
              expr_t expr )
```

Check whether MIR expression subtree represents variable

9.22.3.43 getDefinitionNodeForTemporary()

```
NodeCollectionPtr kpa::getDefinitionNodeForTemporary (
              expr_t temp,
              node_t n,
             ThreadContext & threadContext )
```

Get set of MIR nodes defining the specified temporary

Parameters

expr	MIR expression storing the register
threadContext	reference to ThreadContext object maintained by analysis engine

Returns

smart pointer to the found node set

9.23 Operation codes in MIR expressions

Functions

- int **kpa::expr_getOperationCode** (**expr_t** binary_or_unary_expr)

Variables

- const int **kpa::OPCODE_NONE**
- const int **kpa::OPCODE_ADD**
- const int **kpa::OPCODE_ADDRESS**
- const int **kpa::OPCODE_ASL**
- const int **kpa::OPCODE_ASR**
- const int **kpa::OPCODE_BITAND**
- const int **kpa::OPCODE_BITNOT**
- const int **kpa::OPCODE_BITOR**
- const int **kpa::OPCODE_BITXOR**
- const int **kpa::OPCODE_CAST**
- const int **kpa::OPCODE_DEREF**
- const int **kpa::OPCODE_DIV**
- const int **kpa::OPCODE_EQ**
- const int **kpa::OPCODE_GE**
- const int **kpa::OPCODE_GT**
- const int **kpa::OPCODE_IDIV**
- const int **kpa::OPCODE_LE**
- const int **kpa::OPCODE_LOGAND**
- const int **kpa::OPCODE_LOGNOT**
- const int **kpa::OPCODE_LOGOR**
- const int **kpa::OPCODE_LT**
- const int **kpa::OPCODE_MAX**
- const int **kpa::OPCODE_MIN**
- const int **kpa::OPCODE_MOD**
- const int **kpa::OPCODE_UMOD**
- const int **kpa::OPCODE_MUL**
- const int **kpa::OPCODE_NE**
- const int **kpa::OPCODE_SIZEOF**
- const int **kpa::OPCODE_SUB**
- const int **kpa::OPCODE_THROW**

9.23.1 Detailed Description

9.23.2 Function Documentation

9.23.2.1 expr_getOperationCode()

```
int kpa::expr_getOperationCode (
            expr_t binary_or_unary_expr )
```

Get operation code from a MIR expression for binary or unary operation

9.23.3 Variable Documentation

9.23.3.1 OPCODE_ADD

```
const int kpa::OPCODE_ADD
```

'+'

9.23.3.2 OPCODE_ADDRESS

```
const int kpa::OPCODE_ADDRESS
```

'&'

9.23.3.3 OPCODE_ASL

```
const int kpa::OPCODE_ASL
```

'<<'

9.23.3.4 OPCODE_ASR

```
const int kpa::OPCODE_ASR
```

'>>'

9.23.3.5 OPCODE_BITAND

```
const int kpa::OPCODE_BITAND
```

'&'

9.23.3.6 OPCODE_BITNOT

```
const int kpa::OPCODE_BITNOT
```

'~'

9.23.3.7 OPCODE_BITOR

```
const int kpa::OPCODE_BITOR
```

'|'

9.23.3.8 OPCODE_BITXOR

```
const int kpa::OPCODE_BITXOR
```

'^'

9.23.3.9 OPCODE_CAST

```
const int kpa::OPCODE_CAST
```

(type)

9.23.3.10 OPCODE_DEREF

```
const int kpa::OPCODE_DEREF
```

'*x'

9.23.3.11 OPCODE_DIV

```
const int kpa::OPCODE_DIV
```

'/'

9.23.3.12 OPCODE_EQ

```
const int kpa::OPCODE_EQ
```

'=='

9.23.3.13 OPCODE_GE

```
const int kpa::OPCODE_GE
```

'>='

9.23.3.14 OPCODE_GT

```
const int kpa::OPCODE_GT
```

'>'

9.23.3.15 OPCODE_IDIV

```
const int kpa::OPCODE_IDIV
```

'/' - integer division

9.23.3.16 OPCODE_LE

```
const int kpa::OPCODE_LE
```

'<='

9.23.3.17 OPCODE_LOGAND

```
const int kpa::OPCODE_LOGAND
```

'&&'

9.23.3.18 OPCODE_LOGNOT

```
const int kpa::OPCODE_LOGNOT
```

'!'

9.23.3.19 OPCODE_LOGOR

```
const int kpa::OPCODE_LOGOR
```

'||'

9.23.3.20 OPCODE_LT

```
const int kpa::OPCODE_LT
```

'<'

9.23.3.21 OPCODE_MAX

```
const int kpa::OPCODE_MAX
```

'>?', returns maximum of two operands (GNU extension)

9.23.3.22 OPCODE_MIN

```
const int kpa::OPCODE_MIN
```

'<?', returns minimum of two operands (GNU extension)

9.23.3.23 OPCODE_MOD

```
const int kpa::OPCODE_MOD
```

"

9.23.3.24 OPCODE_MUL

```
const int kpa::OPCODE_MUL
```

'*'

9.23.3.25 OPCODE_NE

```
const int kpa::OPCODE_NE
```

'!='

9.23.3.26 OPCODE_NONE

```
const int kpa::OPCODE_NONE
```

No operation

9.23.3.27 OPCODE_SIZEOF

```
const int kpa::OPCODE_SIZEOF
```

'sizeof'

9.23.3.28 OPCODE_SUB

```
const int kpa::OPCODE_SUB
```

'-', binary - for subtraction

9.23.3.29 OPCODE_THROW

```
const int kpa::OPCODE_THROW
```

'throw'

9.23.3.30 OPCODE_UMOD

```
const int kpa::OPCODE_UMOD
```

" unsigned version

9.24 Working with memory items

Modules

- **Usage of memory items in MIR nodes**

Functions

- **memitem_t kpa::extractMemoryItem** (**expr_t** expr, ThreadContext &threadContext)
- **memitem_t kpa::memitem_getPointed** (**memitem_t** mi)
- **memitem_t kpa::memitem_getPointer** (**memitem_t** mi)
- **memitem_t kpa::memitem_getParent** (**memitem_t** mi)
- const char * **kpa::memitem_getName** (**memitem_t** mi)
- int **kpa::memitem_isGlobal** (**memitem_t** mi)
- int **kpa::memitem_isStatic** (**memitem_t** mi)
- int **kpa::memitem_isLocal** (**memitem_t** mi)
- int **kpa::memitem_isTemporary** (**memitem_t** mi)
- int **kpa::memitem_isFunctionArgument** (**memitem_t** mi)
- int **kpa::memitem_isAddress** (**memitem_t** mi)
- int **kpa::memitem_isPointer** (**memitem_t** mi)
- int **kpa::memitem_isPointerToConst** (**memitem_t** mi)
- int **kpa::memitem_isClass** (**memitem_t** mi)
- int **kpa::memitem_isBuiltin** (**memitem_t** mi)
- int **kpa::memitem_isUnion** (**memitem_t** mi)
- int **kpa::memitem_isInstantiation** (**memitem_t** mi)
- int **kpa::memitem_isArray** (**memitem_t** mi)
- int **kpa::memitem_isUnknown** (**memitem_t** mi)
- int **kpa::memitem_isArrowField** (**memitem_t** mi)
- **sema_t kpa::memitem_getSemanticInfo** (**memitem_t** mi)
- **sema_t kpa::memitem_getTypeSemanticInfo** (**memitem_t** mi)

9.24.1 Detailed Description

9.24.2 Function Documentation

9.24.2.1 extractMemoryItem()

```
memitem_t kpa::extractMemoryItem (
            expr_t expr,
           ThreadContext & threadContext )
```

Extract memory item from MIR expression

Parameters

expr	MIR expression to extract the memory item for
threadContext	reference to ThreadContext object maintained by analysis engine

Returns

memory item descriptor if MIR expression designates valid memory item, 0 otherwise

9.24.2.2 memitem_getName()

```
const char* kpa::memitem_getName (
            memitem_t mi )
```

Get memory item name

9.24.2.3 memitem_getParent()

```
memitem_t kpa::memitem_getParent (
            memitem_t mi )
```

Extract memory item which is a parent of another memory item

Returns

memory item descriptor if passed memory item is a field of a valid memory item, 0 otherwise

9.24.2.4 memitem_getPointed()

```
memitem_t kpa::memitem_getPointed (
            memitem_t mi )
```

Extract memory item pointed by another memory item

Returns

memory item descriptor if passed memory item points to valid memory item, 0 otherwise

9.24.2.5 memitem_getPointer()

```
memitem_t kpa::memitem_getPointer (
            memitem_t mi )
```

Extract memory item which is a pointer for another memory item

Returns

memory item descriptor if passed memory item is pointed by valid memory item, 0 otherwise

9.24.2.6 memitem_getSemanticInfo()

```
sema_t kpa::memitem_getSemanticInfo (
              memitem_t mi )
```

Get semantic information on memory item

9.24.2.7 memitem_getTypeSemanticInfo()

```
sema_t kpa::memitem_getTypeSemanticInfo (
              memitem_t mi )
```

Get semantic information on memory item type

9.24.2.8 memitem_isAddress()

```
int kpa::memitem_isAddress (
              memitem_t mi )
```

Check if memory item represents an address

9.24.2.9 memitem_isArray()

```
int kpa::memitem_isArray (
              memitem_t mi )
```

Check if memory item is an array

9.24.2.10 memitem_isArrowField()

```
int kpa::memitem_isArrowField (
              memitem_t mi )
```

Check if memory item is an arrow field, i.e. represents memory cell of a->b

9.24.2.11 memitem_isBuiltin()

```
int kpa::memitem_isBuiltin (
              memitem_t mi )
```

Check if memory item is a scalar of builtin type

9.24.2.12 memitem_isClass()

```
int kpa::memitem_isClass (
              memitem_t mi )
```

Check if memory item is object of a class

9.24.2.13 memitem_isFunctionArgument()

```
int kpa::memitem_isFunctionArgument (
              memitem_t mi )
```

Check if memory item is a function argument

9.24.2.14 memitem_isGlobal()

```
int kpa::memitem_isGlobal (
              memitem_t mi )
```

Check if memory item is a global variable

9.24.2.15 memitem_isInstantiation()

```
int kpa::memitem_isInstantiation (
              memitem_t mi )
```

Check if memory item is an instantiation of a template

9.24.2.16 memitem_isLocal()

```
int kpa::memitem_isLocal (
              memitem_t mi )
```

Check if memory item is a local variable

9.24.2.17 memitem_isPointer()

```
int kpa::memitem_isPointer (
              memitem_t mi )
```

Check if memory item is a pointer

9.24.2.18 memitem_isPointerToConst()

```
int kpa::memitem_isPointerToConst (
              memitem_t mi )
```

Check if memory item is a pointer to constant memory

9.24.2.19 memitem_isStatic()

```
int kpa::memitem_isStatic (
              memitem_t mi )
```

Check if memory item is static

9.24.2.20 memitem_isTemporary()

```
int kpa::memitem_isTemporary (
              memitem_t mi )
```

Check if memory item is a temporary value

9.24.2.21 memitem_isUnion()

```
int kpa::memitem_isUnion (
              memitem_t mi )
```

Check if memory item is an instance of union type.

Remarks

Union is always a class in C++, so memitem_isClass will also return 1 for any memory item that is union.

9.24.2.22 memitem_isUnknown()

```
int kpa::memitem_isUnknown (
              memitem_t mi )
```

Check if memory item is unknown

Remarks

in MIR created from properly compiled source code, this function always returns 0. Presence of unknown memory items usually implies missed header files or syntax errors

9.25 Usage of memory items in MIR nodes

Modules

- **Memory item usage constants**

Functions

- int **kpa::memitemUsage** (**node_t** node, **memitem_t** mi, ThreadContext &threadContext)
- **memitem_t kpa::memitemGetAliased** (**node_t** node, **memitem_t** mi, ThreadContext &threadContext)

9.25.1 Detailed Description

9.25.2 Function Documentation

9.25.2.1 memitemGetAliased()

```
memitem_t kpa::memitemGetAliased (
              node_t node,
              memitem_t mi,
             ThreadContext & threadContext )
```

Check if memory item is aliases with another memory item in a given node.

Parameters

node	node under investigation
mi	memory item
threadContext	reference to ThreadContext object maintained by analysis engine

Returns

0 if memory item is not aliased in this node, or memory item of an alias otherwise

9.25.2.2 memitemUsage()

```
int kpa::memitemUsage (
              node_t node,
              memitem_t mi,
             ThreadContext & threadContext )
```

Find out how memory item is used in node.

Parameters

node	node under investigation
mi	memory item for which we need to know type of usage in the node.
threadContext	reference to ThreadContext object maintained by analysis engine

Returns

combination of usage constants (

See also

Memory item usage constants (p. 112))

9.26 Memory item usage constants

Variables

- const int **kpa::MI_NO_ACTION**
- const int **kpa::MI_MIGHT_BE_READ**
- const int **kpa::MI_IS_READ**
- const int **kpa::MI_MIGHT_BE_CHANGED**
- const int **kpa::MI_IS_CHANGED**
- const int **kpa::MI_ALIASED**
- const int **kpa::MI_IS_READ_PARTIALLY**
- const int **kpa::MI_IS_READ_INDIRECTLY**
- const int **kpa::MI_IS_OVERWRITTEN**

9.26.1 Detailed Description

9.26.2 Variable Documentation

9.26.2.1 MI_ALIASED

```
const int kpa::MI_ALIASED
```

Memory item gets an alias in node.

Testing if memory item is aliased:

```
usage = memitemUsage(node, mi, threadContext);
if (usage & MI_ALIASED == MI_ALIASED) { ... }
```

9.26.2.2 MI_IS_CHANGED

```
const int kpa::MI_IS_CHANGED
```

Memory item is changed in node.

The difference from MI_IS_OVERWRITTEN is as follows: value of memory item is changed if access to memory item involves dereferencing a pointer and the value of this pointer is changed. Overwriting assumes that memory cell itself associated with memory item is changed. For example:

```
p = p->next; // 'p' is both MI_IS_READ and MI_IS_CHANGED but not MI_IS_OVERWRITTEN
p = q; // p is MI_IS_OVERWRITTEN
```

MI_IS_CHANGED assumes MI_MIGHT_BE_CHANGED

Testing if memory item is changed:

```
usage = memitemUsage(node, mi, threadContext);
if (usage & MI_IS_CHANGED == MI_IS_CHANGED) { ... }
```

9.26.2.3 MI_IS_OVERWRITTEN

```
const int kpa::MI_IS_OVERWRITTEN
```

The value of memory item is overwritten.

Also assumes MI_IS_CHANGED

Testing if memory item is overwritten:

```
usage = memitemUsage(node, mi, threadContext);
if (usage & MI_IS_OVERWRITTEN == MI_IS_OVERWRITTEN) { ... }
```

9.26.2.4 MI_IS_READ

```
const int kpa::MI_IS_READ
```

Memory item is read in node.

MI_IS_READ assumes MI_MIGHT_BE_READ but not vice versa. For example, for a node 'y=x+1', if we test memory item 'x' it is read and might be read at the same time and returned result will be MI_IS_READ. However if we have a call 'foo(x)', parameter may or may not be read by function, and returned result will be MI_MIGHT_BE↩ _READ.

Testing if memory item is read:

```
usage = memitemUsage(node, mi, threadContext);
if (usage & MI_IS_READ == MI_IS_READ) { ... }
```

9.26.2.5 MI_IS_READ_INDIRECTLY

```
const int kpa::MI_IS_READ_INDIRECTLY
```

Memory item is read indirectly.

9.26.2.6 MI_IS_READ_PARTIALLY

```
const int kpa::MI_IS_READ_PARTIALLY
```

Part of memory item contents is read. For example in code:

```
a = x.f;
```

part of 'x' is read. Testing if memory item is read partially:

```
usage = memitemUsage(node, mi, threadContext);
if (usage & MI_IS_READ_PARTIALLY) { ... }
```

9.26.2.7 MI_MIGHT_BE_CHANGED

```
const int kpa::MI_MIGHT_BE_CHANGED
```

Value memory item might be changed in node.

Testing if memory item may be changed:

```
usage = memitemUsage(node, mi, threadContext);
if (usage & MI_MIGHT_BE_CHANGED) { ... }
```

9.26.2.8 MI_MIGHT_BE_READ

```
const int kpa::MI_MIGHT_BE_READ
```

Memory item might be read in node.

To test that memory item may be changed in node, result of 'memitemUsage' must be bit-AND'ed with MI_MIGH↩
T_BE_READ:

```
usage = memitemUsage(node, mi, threadContext);
if (usage & MI_MIGHT_BE_READ) { ... }
```

9.26.2.9 MI_NO_ACTION

```
const int kpa::MI_NO_ACTION
```

Node does not use memory item in any way

9.27 Constraints on memory item values

Functions

- **constraint_t kpa::bb_getPreConstraint** (**memitem_t** mi, **bb_t** bb, **function_t** func, ThreadContext &threadContext)
- **constraint_t kpa::bb_getPostConstraint** (**memitem_t** mi, **bb_t** bb, **function_t** func, ThreadContext &threadContext)
- **constraint_t kpa::mi_getNodePreConstraint** (**memitem_t** mi, **node_t** node, **function_t** func, ThreadContext &threadContext)
- int **kpa::constraint_isValue** (**constraint_t** cons)
- int **kpa::constraint_getValue** (**constraint_t** cons)
- int **kpa::constraint_getMinValue** (**constraint_t** cons, long int *a)
- bool **kpa::constraint_hasMinValue** (**constraint_t** cons)
- **integer_t kpa::constraint_getMinValue** (**constraint_t** cons)
- int **kpa::constraint_getMaxValue** (**constraint_t** cons, long int *a)
- bool **kpa::constraint_hasMaxValue** (**constraint_t** cons)
- **integer_t kpa::constraint_getMaxValue** (**constraint_t** cons)
- int **kpa::constraint_isGE** (**constraint_t** cons, long int *a)
- int **kpa::constraint_isLE** (**constraint_t** cons, long int *a)
- int **kpa::constraint_isInterval** (**constraint_t** cons, long int *a, long int *b)
- int **kpa::constraint_isNE** (**constraint_t** cons, long int *a)
- **integer_t kpa::constraint_getNEValue** (**constraint_t** cons)
- int **kpa::constraint_isEQ** (**constraint_t** cons, long int *a)
- **integer_t kpa::constraint_getEQValue** (**constraint_t** cons)
- bool **kpa::constraint_containsValue** (**constraint_t** cons, const **integer_t** &value)
- bool **kpa::constraint_containsNoValues** (**constraint_t** cons)
- bool **kpa::constraint_containsAllValues** (**constraint_t** cons)
- void **kpa::constraint_toString** (**constraint_t** cons, char *buf, size_t bufsize)
- void **kpa::constraint_delete** (**constraint_t** cons)

9.27.1 Detailed Description

9.27.2 Function Documentation

9.27.2.1 bb_getPostConstraint()

```
constraint_t kpa::bb_getPostConstraint (
            memitem_t mi,
            bb_t bb,
            function_t func,
           ThreadContext & threadContext )
```

Get constraint on memory item value at the end of the basic block

Parameters

mi	memory item for which the constraint needs to be evaluated
bb	basic block for which the post-constraint is requested
func	currently analyzed function
threadContext	reference to ThreadContext object maintained by analysis engine Returned constraint should not be freed

9.27.2.2 bb_getPreConstraint()

```
constraint_t kpa::bb_getPreConstraint (
            memitem_t mi,
            bb_t bb,
            function_t func,
           ThreadContext & threadContext )
```

Get constraint on memory item value at the beginning of the basic block

Parameters

mi	memory item for which the constraint needs to be evaluated
bb	basic block for which the pre-constraint is requested
func	currently analyzed function
threadContext	reference to ThreadContext object maintained by analysis engine Returned constraint should not be freed

9.27.2.3 constraint_containsAllValues()

```
bool kpa::constraint_containsAllValues (
            constraint_t cons )
```

Check if the constraint does not restrict the range of values, i.e. the constraint contains all possible values.

Parameters

cons	constraint to check

Returns

a boolean stating if all possible values are allowed by the constraint.

9.27.2.4 constraint_containsNoValues()

```
bool kpa::constraint_containsNoValues (
            constraint_t cons )
```

Check if the constraint does not allow any values at all, i.e. the constraint contains no values.

Parameters

cons	constraint to check

Returns

a boolean stating if no values are allowed by the constraint.

9.27.2.5 constraint_containsValue()

```
bool kpa::constraint_containsValue (
              constraint_t cons,
             const  integer_t & value )
```

Check if the constraint contains the value within its bound.

Parameters

cons	constraint to check
value	the value to check if it is contained in `cons`

Returns

a boolean stating if the value is contained in the constraint.

9.27.2.6 constraint_delete()

```
void kpa::constraint_delete (
              constraint_t cons )
```

Free memory taken by a constraint

9.27.2.7 constraint_getEQValue()

```
integer_t kpa::constraint_getEQValue (
              constraint_t cons )
```

Check if constraint includes just a single integer value. I.e. it represents '== a' or {a} in math notation.

Parameters

cons	constraint to check

Returns

the only integer value included in the constraint if it is a '==' constraint. Otherwise, an invalid integer is returned (can be detected by calling **kpa::integer_t::isValid()** (p. 35)).

9.27.2.8 constraint_getMaxValue() [1/2]

```
int kpa::constraint_getMaxValue (
              constraint_t cons,
             long int * a )
```

Check if constraint has a maximum value and if it has, get it.

Returns

1 if constraint has a maximum value or 0 if higher bound is plus infinity

Parameters

cons	constraint to check
a	pointer to long integer that will contain higher bound upon exit

Deprecated This function is only dealing with 32 bit signed integers.
It is replaced by **constraint_getMaxValue(constraint_t cons)** (p. 118).

```
// Example of the proposed migration.
// Old code:
void f(constraint_t cons)
{
    long int max;
    if (constraint_getMaxValue(cons, &max)) {
        ...
    }
}

// New code:
void f(constraint_t cons)
{
    integer_t max = constraint_getMaxValue(cons);
    if (max.isValid()) {
        ...
    }
}
```

9.27.2.9 constraint_getMaxValue() [2/2]

```
integer_t kpa::constraint_getMaxValue (
              constraint_t cons )
```

Get the maximum value of the constraint.

Parameters

cons	constraint to get its maximum value.

Returns

an integer representing the maximum value of the constraint. An invalid integer is returned if the constraint does not have a maximum value.

9.27.2.10 constraint_getMinValue() [1/2]

```
int kpa::constraint_getMinValue (
              constraint_t cons,
             long int * a )
```

Check if constraint has a minimum value and if it has, get it.

Returns

1 if constraint has a minimum value or 0 if lower bound is minus infinity

Parameters

cons	constraint to check
a	pointer to long integer that will contain lower bound upon exit

Deprecated This function is only dealing with 32 bit signed integers.
It is replaced by **constraint_getMinValue(constraint_t cons)** (p. 119).

```
// Example of the proposed migration.
// Old code:
void f(constraint_t cons)
{
    long int min;
    if (constraint_getMinValue(cons, &min)) {
        ...
    }
}

// New code:
void f(constraint_t cons)
{
    integer_t min = constraint_getMinValue(cons);
    if (min.isValid()) {
        ...
    }
}
```

9.27.2.11 constraint_getMinValue() [2/2]

```
integer_t kpa::constraint_getMinValue (
              constraint_t cons )
```

Get the minimum value of the constraint.

Parameters

cons	constraint to get its minimum value.

Returns

an integer representing the minimum value of the constraint. An invalid integer is returned if the constraint does not have a minimum value.

9.27.2.12 constraint_getNEValue()

```
integer_t kpa::constraint_getNEValue (
            constraint_t cons )
```

Check if constraint includes all values except for a single value 'a'. I.e. it represents '!= a' or '(-oo, a-1] U [a+1, +oo)' in math notation.

Parameters

cons	constraint to check

Returns

the single integer value that is not included in the constraint ('a' in the description) if it is a '!=' constraint. Otherwise, an invalid integer is returned (can be detected by calling **kpa::integer_t::isValid()** (p. 35)).

9.27.2.13 constraint_getValue()

```
int kpa::constraint_getValue (
            constraint_t cons )
```

Get constraint value. (Get x from constraint {x})

See also

constraint_isValue (p. 125)

Deprecated This function is only dealing with 32 bit signed integers.
It is replaced by **constraint_getEQValue(constraint_t cons)** (p. 117).

```
// Example of the proposed migration.
// Old code:
void f(constraint_t cons)
{
    if (constraint_isValue(cons)) {
        long int val = constraint_getValue(cons);
        ...
    }
}

// New code:
void f(constraint_t cons)
{
    integer_t x = constraint_getEQValue(cons);
    if (x.isValid()) {
        ...
    }
}
```

9.27.2.14 constraint_hasMaxValue()

```
bool kpa::constraint_hasMaxValue (
              constraint_t cons )
```

Check if constraint has a maximum value.

Parameters

cons	constraint to check

Returns

true if constraint has a maximum value or false if upper bound is plus infinity

9.27.2.15 constraint_hasMinValue()

```
bool kpa::constraint_hasMinValue (
              constraint_t cons )
```

Check if constraint has a minimum value.

Parameters

cons	constraint to check

Returns

true if constraint has a minimum value or false if lower bound is minus infinity

9.27.2.16 constraint_isEQ()

```
int kpa::constraint_isEQ (
              constraint_t cons,
             long int * a )
```

Check if constraint is just one integer value x==a ({a} in math notation)

Returns

0 if constraint is not a 'equal to', non-zero otherwise

Parameters

cons	constraint to check
a	pointer to long integer that will contain value of 'a' upon exit

Deprecated This function is only dealing with 32 bit signed integers.
It is replaced by **constraint_getEQValue(constraint_t cons)** (p. 117).

```
// Example of the proposed migration.
// Old code:
void f(constraint_t cons)
{
    long int val;
    if (constraint_isEQ(cons, &val)) {
        ...
    }
}

// New code:
void f(constraint_t cons)
{
    integer_t x = constraint_getEQValue(cons);
    if (x.isValid()) {
        ...
    }
}
```

9.27.2.17 constraint_isGE()

```
int kpa::constraint_isGE (
              constraint_t cons,
             long int * a )
```

Check if constraint is of form x>=a (or [a, +oo) in math notation)

Returns

0 if constraint is not a 'greater or equal than', non-zero otherwise

Parameters

cons	constraint to check
a	pointer to long integer that will contain value of 'a' upon exit

Deprecated This function is only dealing with 32 bit signed integers.
It is replaced by a combination of **constraint_getMinValue(constraint_t cons)** (p. 119) and **constraint_hasMaxValue(constraint_t cons)** (p. 120).

```
// Example of the proposed migration.
// Old code:
void f(constraint_t cons)
{
    long int min;
    if (constraint_isGE(cons, &min)) {
        ...
    }
}

// New code:
void f(constraint_t cons)
{
    if (constraint_hasMinValue(cons) && !constraint_hasMaxValue(cons)) {
        integer_t min = constraint_getMinValue(cons);
        ...
    }
}
```

9.27.2.18 constraint_isInterval()

```
int kpa::constraint_isInterval (
              constraint_t cons,
             long int * a,
             long int * b )
```

Check if constraint is an interval beginValue=<x<=endValue ([a,b] in math notation)

Returns

0 if constraint is not a bound interval, non-zero otherwise

Parameters

cons	constraint to check
a	pointer to long integer that will contain minimal value of an interval upon exit
b	pointer to long integer that will contain maximal value of an interval upon exit

Deprecated This function is only dealing with 32 bit signed integers.
It is replaced by a combination of **constraint_getMinValue(constraint_t cons)** (p. 119) and **constraint_getMaxValue(constraint_t cons)** (p. 118).

```
// Example of the proposed migration.
// Old code:
void f(constraint_t cons)
{
    long int min;
    long int max;
    if (constraint_isInterval(cons, &min, &max)) {
        ...
    }
}

// New code:
void f(constraint_t cons)
{
    if (constraint_hasMinValue(cons) && constraint_hasMaxValue(cons)) {
        integer_t min = constraint_getMinValue(cons);
        integer_t max = constraint_getMaxValue(cons);
        ...
    }
}
```

9.27.2.19 constraint_isLE()

```
int kpa::constraint_isLE (
              constraint_t cons,
             long int * a )
```

Check if constraint is of form x<=a (or (-oo, a] in math notation)

Returns

0 if constraint is not a 'lesser or equal than', non-zero otherwise

Parameters

cons	constraint to check
a	pointer to long integer that will contain value of 'a' upon exit

Deprecated This function is only dealing with 32 bit signed integers.
It is replaced by a combination of **constraint_hasMinValue(constraint_t cons)** (p. 121) and **constraint_getMaxValue(constraint_t cons)** (p. 118).

```
// Example of the proposed migration.
// Old code:
void f(constraint_t cons)
{
    long int max;
    if (constraint_isLE(cons, &max)) {
        ...
    }
}

// New code:
void f(constraint_t cons)
{
    if (!constraint_hasMinValue(cons) && constraint_hasMaxValue(cons)) {
        integer_t max = constraint_getMaxValue(cons);
        ...
    }
}
```

9.27.2.20 constraint_isNE()

```
int kpa::constraint_isNE (
              constraint_t cons,
             long int * a )
```

Check if constraint excludes just one value x!=a ((-oo, a-1] U [a+1, +oo) in math notation)

Returns

0 if constraint is not a 'not equal to', non-zero otherwise

Parameters

cons	constraint to check
a	pointer to long integer that will contain value of 'a' upon exit

Deprecated This function is only dealing with 32 bit signed integers.
It is replaced by **constraint_getNEValue(constraint_t cons)** (p. 120).

```
// Example of the proposed migration.
// Old code:
void f(constraint_t cons)
{
    long int val;
    if (constraint_isNE(cons, &val)) {
```

```
        ...
    }
}

// New code:
void f(constraint_t cons)
{
    integer_t x = constraint_getNEValue(cons);
    if (x.isValid()) {
        ...
    }
}
```

9.27.2.21 constraint_isValue()

```
int kpa::constraint_isValue (
              constraint_t cons )
```

Check if constraint is just one integer value {x}

See also

constraint_getValue (p. 120)

Deprecated This function was partially duplicated by **constraint_isEQ()** (p. 121).
It is replaced by **constraint_getEQValue(constraint_t cons)** (p. 117).

```
// Example of the proposed migration.
// Old code:
void f(constraint_t cons)
{
    if (constraint_isValue(cons)) {
        long int val = constraint_getValue(cons);
        ...
    }
}

// New code:
void f(constraint_t cons)
{
    integer_t x = constraint_getEQValue(cons);
    if (x.isValid()) {
        ...
    }
}
```

9.27.2.22 constraint_toString()

```
void kpa::constraint_toString (
              constraint_t cons,
             char * buf,
             size_t bufsize )
```

Render constraint in string form

Parameters

cons	constraint to render into string
buf	character buffer for constraint's string representation
bufsize	character buffer size

9.27.2.23 mi_getNodePreConstraint()

```
constraint_t kpa::mi_getNodePreConstraint (
            memitem_t mi,
            node_t node,
            function_t func,
           ThreadContext & threadContext )
```

Get constraint on memory item effective before the given node

Parameters

mi	memory item for which the constraint needs to be evaluated
node	MIR node for which the pre-constraint is requested
func	currently analyzed function
threadContext	reference to ThreadContext object maintained by analysis engine Returns newly allocated copy of a constraint which must be later freed by a call to 'constraint_delete'

9.28 Positions in MIR

Typedefs

- typedef struct tLEFPosition * **kpa::position_t**

Functions

- **position_t kpa::node_getPosition** (**node_t** node)
- int **kpa::position_getLine** (**position_t** pos)
- int **kpa::position_getColumn** (**position_t** pos)

9.28.1 Detailed Description

9.28.2 Typedef Documentation

9.28.2.1 position_t

```
typedef struct tLEFPosition*  kpa::position_t
```

Opaque type for source position description attached to MIR nodes

9.28.3 Function Documentation

9.28.3.1 node_getPosition()

```
position_t kpa::node_getPosition (
            node_t node )
```

Get node's source position

9.28.3.2 position_getColumn()

```
int kpa::position_getColumn (
            position_t pos )
```

Get column number from position

9.28.3.3 position_getLine()

```
int kpa::position_getLine (
            position_t pos )
```

Get line number from position

9.29 Trace and events

Typedefs

- typedef void * **kpa::trace_t**

Functions

- **trace_t kpa::trace_new** (**function_t** func)
- void * **kpa::event_new** (void *func, void *pos, const char *event)
- void **kpa::event_setParameter** (void *event, const char *name, const char *value)
- void **kpa::trace_addEvent** (**trace_t** trace, void *func, **position_t** pos, const char *event)
- void **kpa::trace_addEventEx** (**trace_t** trace, void *event)
- void **kpa::trace_delete** (**trace_t** trace)

9.29.1 Detailed Description

9.29.2 Typedef Documentation

9.29.2.1 trace_t

```
typedef void*  kpa::trace_t
```

9.29.3 Function Documentation

9.29.3.1 event_new()

```
void* kpa::event_new (
             void * func,
             void * pos,
             const char * event )
```

9.29.3.2 event_setParameter()

```
void kpa::event_setParameter (
             void * event,
             const char * name,
             const char * value )
```

9.29.3.3 trace_addEvent()

```
void kpa::trace_addEvent (
              trace_t trace,
             void * func,
              position_t pos,
             const char * event )
```

9.29.3.4 trace_addEventEx()

```
void kpa::trace_addEventEx (
              trace_t trace,
             void * event )
```

9.29.3.5 trace_delete()

```
void kpa::trace_delete (
              trace_t trace )
```

9.29.3.6 trace_new()

```
trace_t kpa::trace_new (
              function_t func )
```

Create new message

9.30 Issue reporting functions

Typedefs

- typedef void * **kpa::message_t**

Functions

- **message_t kpa::message_new** (const char *error_id)
- void **kpa::message_setPosition** (**message_t** msg, **position_t** pos)
- void **kpa::message_setRecommendationFactor** (**message_t** msg, const char *factor, int value)
- void **kpa::message_addAttribute** (**message_t** msg, const char *attr_string)
- void **kpa::message_addAnchorAttribute** (**message_t** msg, const char *attr_string)
- void **kpa::message_addTrace** (**message_t** msg, **trace_t** trace)
- void **kpa::message_render** (**message_t** msg)
- void **kpa::message_delete** (**message_t** msg)

9.30.1 Detailed Description

Usually user doesn't have to use these functions manually. Checkers provided by Klocwork will do it automatically

9.30.2 Typedef Documentation

9.30.2.1 message_t

```
typedef void*  kpa::message_t
```

Opaque type for storing issue message information

9.30.3 Function Documentation

9.30.3.1 message_addAnchorAttribute()

```
void kpa::message_addAnchorAttribute (
              message_t msg,
             const char * attr_string )
```

add attribute to the issue message. This attribute will be a part of anchor which is used to propagate defects. If defect doesn't use anchor attributes, then all attributes will be a part of anchor.

9.30.3.2 message_addAttribute()

```
void kpa::message_addAttribute (
              message_t msg,
             const char * attr_string )
```

add attribute to the issue message

9.30.3.3 message_addTrace()

```
void kpa::message_addTrace (
              message_t msg,
              trace_t trace )
```

add trace to the issue message

9.30.3.4 message_delete()

```
void kpa::message_delete (
              message_t msg )
```

Free memory occupied by issue message

9.30.3.5 message_new()

```
 message_t kpa::message_new (
             const char * error_id )
```

Create new issue message

9.30.3.6 message_render()

```
void kpa::message_render (
              message_t msg )
```

Register issue message

9.30.3.7 message_setPosition()

```
void kpa::message_setPosition (
              message_t msg,
              position_t pos )
```

Set position for the issue message

9.30.3.8 message_setRecommendationFactor()

```
void kpa::message_setRecommendationFactor (
              message_t msg,
             const char * factor,
             int value )
```

Set a recommendation factor for the issue message

9.31 Semantic information in MIR

Modules

- **Numerical codes of declaration type qualifiers (empty, 'const' or 'volatile'). Actual values are bit-or'ed superpositions of these flags.**
- **Numerical codes of builtin types**

Functions

- **sema_t kpa::expr_getSemanticInfo** (**expr_t** expr)
- **sema_t kpa::func_getSemanticInfo** (**function_t** func)
- const char * **kpa::sema_getName** (**sema_t** si)
- const char * **kpa::sema_getQualifiedName** (**sema_t** si)
- int **kpa::sema_isFunction** (**sema_t** si)
- **sema_t kpa::sema_getFunctionType** (**sema_t** si)
- int **kpa::sema_getNumberOfArguments** (**sema_t** si)
- **sema_t kpa::sema_getFormalArgument** (**sema_t** si, int argnum)
- **sema_t kpa::sema_getFunctionReturnType** (**sema_t** si)
- int **kpa::sema_isPointer** (**sema_t** si)
- int **kpa::sema_isReference** (**sema_t** si)
- int **kpa::sema_isVariable** (**sema_t** si)
- int **kpa::sema_isType** (**sema_t** si)
- **sema_t kpa::sema_getVariableType** (**sema_t** si)
- int **kpa::sema_isBuiltin** (**sema_t** si)
- int **kpa::sema_isClass** (**sema_t** si)
- int **kpa::sema_isUnion** (**sema_t** si)
- int **kpa::sema_isEnum** (**sema_t** si)
- int **kpa::sema_getBuiltin** (**sema_t** si)
- int **kpa::sema_getCVQualifiers** (**sema_t** si)
- **sema_t kpa::sema_getPointedType** (**sema_t** si)
- bool **kpa::sema_isBaseClass** (**sema_t** base, **sema_t** derived)
- **sema_t kpa::sema_getNextBaseClass** (**sema_t** cl, **sema_t** base)
- **sema_t kpa::sema_getParent** (**sema_t** si)
- int **kpa::memoryChangedInCall** (**expr_t** call_expr, int argnum)

Variables

- const int **kpa::MEMCHANGE_NOT_A_CALL**
- const int **kpa::MEMCHANGE_NO_SEMANTIC_INFO**
- const int **kpa::MEMCHANGE_NOT_A_FUNCTION**
- const int **kpa::MEMCHANGE_NOT_A_POINTER**
- const int **kpa::MEMCHANGE_INVALID_ARGUMENT**
- const int **kpa::MEMCHANGE_NOT_CHANGED**
- const int **kpa::MEMCHANGE_MAY_BE_CHANGED**
- const int **kpa::MEMCHANGE_CHANGED**

9.31.1 Detailed Description

9.31.2 Function Documentation

9.31.2.1 expr_getSemanticInfo()

```
sema_t kpa::expr_getSemanticInfo (
              expr_t expr )
```

Get semantic description of an MIR expression

9.31.2.2 func_getSemanticInfo()

```
sema_t kpa::func_getSemanticInfo (
              function_t func )
```

Get semantic description of analyzed function

9.31.2.3 memoryChangedInCall()

```
int kpa::memoryChangedInCall (
              expr_t call_expr,
             int argnum )
```

Checks if data in buffer pointed by some argument are changed by a call to function or method

Parameters

call_expr	call expression to check
argnum	actual parameter number

9.31.2.4 sema_getBuiltin()

```
int kpa::sema_getBuiltin (
              sema_t si )
```

Get builtin tyype code (

See also

Numerical codes of builtin types (p. 140)). If semantic information does not represent a buitin type, BUIL↩ TIN_NODE is returned

9.31.2.5 sema_getCVQualifiers()

```
int kpa::sema_getCVQualifiers (
              sema_t si )
```

Get type qualifier flags (see **Numerical codes of declaration type qualifiers (empty, 'const' or 'volatile'). Actual values are bit-or'ed superpositions of these flags.** (p. 139)) for type description

9.31.2.6 sema_getFormalArgument()

```
sema_t kpa::sema_getFormalArgument (
            sema_t si,
           int argnum )
```

Get description of a function formal argument

Returns

0 if 'si' is not a semantic description of a function, semantic description of formal argument otherwise

9.31.2.7 sema_getFunctionReturnType()

```
sema_t kpa::sema_getFunctionReturnType (
            sema_t si )
```

Get description of a function return type.

Returns

0 if si is not a function or a function type.

9.31.2.8 sema_getFunctionType()

```
sema_t kpa::sema_getFunctionType (
            sema_t si )
```

Get description of a function type of a function

Returns

0 if 'si' is not a semantic description of function, function type description otherwise

9.31.2.9 sema_getName()

```
const char* kpa::sema_getName (
            sema_t si )
```

Get symbol name from its semantic description

9.31.2.10 sema_getNextBaseClass()

```
sema_t kpa::sema_getNextBaseClass (
              sema_t cl,
              sema_t base )
```

Iterate immediate base classes. Order of base classes may be different from the declaration order

Returns

first base class if 'base' is equal to 0, otherwise next base class is returned or 0 if there are no more base classes

9.31.2.11 sema_getNumberOfArguments()

```
int kpa::sema_getNumberOfArguments (
              sema_t si )
```

Get number of formal arguments for function semantic description. Must be applied to semantic information of a function, if pointer to semantic information is null, function will return -1.

Arguments are counted from 0, but argument #0 in Path API always stands for 'this' pointer, even if function is not a mehtod of any class ('this' pointer will be 0 in that case) actual arguments start from 1. Make sure you use '<=' operation in iteration: `for (i=1;i<=sema_getNumberOfArguments(function_sema);i++) { ... }`

9.31.2.12 sema_getParent()

```
sema_t kpa::sema_getParent (
              sema_t si )
```

Get parent semantic descriptor. Defines scope of the object. For example methods and fields will have class as a parent.

9.31.2.13 sema_getPointedType()

```
sema_t kpa::sema_getPointedType (
              sema_t si )
```

For pointer type, get pointed type

Returns

0 if 'si' is not a description of pointer type, description of pointed type otherwise

9.31.2.14 sema_getQualifiedName()

```
const char* kpa::sema_getQualifiedName (
              sema_t si )
```

Get qualified symbol name from its semantic description

9.31.2.15 sema_getVariableType()

```
 sema_t kpa::sema_getVariableType (
              sema_t si )
```

Get semantic description of variable type

9.31.2.16 sema_isBaseClass()

```
bool kpa::sema_isBaseClass (
              sema_t base,
              sema_t derived )
```

Check if class 'derived' inherits 'base' either directly or indirectly

9.31.2.17 sema_isBuiltin()

```
int kpa::sema_isBuiltin (
              sema_t si )
```

Check if semantic information describes builtin type

9.31.2.18 sema_isClass()

```
int kpa::sema_isClass (
              sema_t si )
```

Check if semantic information describes class

9.31.2.19 sema_isEnum()

```
int kpa::sema_isEnum (
              sema_t si )
```

Check if semantic information describes enumerated type

9.31.2.20 sema_isFunction()

```
int kpa::sema_isFunction (
              sema_t si )
```

Check if semantic information describes function

9.31.2.21 sema_isPointer()

```
int kpa::sema_isPointer (
              sema_t si )
```

Check if semantic information describes pointer type

9.31.2.22 sema_isReference()

```
int kpa::sema_isReference (
              sema_t si )
```

Check if semantic information describes a reference type

9.31.2.23 sema_isType()

```
int kpa::sema_isType (
              sema_t si )
```

Check if semantic information describes some type To elaborate, what kind of type is that, use sema_isClass, sema_isUnion etc

9.31.2.24 sema_isUnion()

```
int kpa::sema_isUnion (
              sema_t si )
```

Check if semantic information describes union

9.31.2.25 sema_isVariable()

```
int kpa::sema_isVariable (
              sema_t si )
```

Check if semantic information describes some variable (local or global, object or scalar)

9.31.3 Variable Documentation

9.31.3.1 MEMCHANGE_CHANGED

```
const int kpa::MEMCHANGE_CHANGED
```

Result returned by memoryChangedInACall. Memory pointed by pointer is changed in a call

9.31.3.2 MEMCHANGE_INVALID_ARGUMENT

```
const int kpa::MEMCHANGE_INVALID_ARGUMENT
```

Error code of function memoryChangedInACall. There is no actual number with given number in a call

9.31.3.3 MEMCHANGE_MAY_BE_CHANGED

```
const int kpa::MEMCHANGE_MAY_BE_CHANGED
```

Result returned by memoryChangedInACall. Memory pointed by pointer may be changed in a call

9.31.3.4 MEMCHANGE_NO_SEMANTIC_INFO

```
const int kpa::MEMCHANGE_NO_SEMANTIC_INFO
```

Error code of function memoryChangedInACall. Semantic information about called function cannot be found

9.31.3.5 MEMCHANGE_NOT_A_CALL

```
const int kpa::MEMCHANGE_NOT_A_CALL
```

Error code of function memoryChangedInACall. Provided expression is not a call

9.31.3.6 MEMCHANGE_NOT_A_FUNCTION

```
const int kpa::MEMCHANGE_NOT_A_FUNCTION
```

Error code of function memoryChangedInACall. Called object is not a function or class method

9.31.3.7 MEMCHANGE_NOT_A_POINTER

```
const int kpa::MEMCHANGE_NOT_A_POINTER
```

Error code of function memoryChangedInACall. Argument type is not a pointer type

9.31.3.8 MEMCHANGE_NOT_CHANGED

```
const int kpa::MEMCHANGE_NOT_CHANGED
```

Result returned by memoryChangedInACall. Memory pointed by pointer is not changed by a call

9.32 Numerical codes of declaration type qualifiers (empty, 'const' or 'volatile'). Actual values are bit-or'ed superpositions of these flags.

Variables

- const int **kpa::CVQUALIFIER_NONE**
- const int **kpa::CVQUALIFIER_CONST**
- const int **kpa::CVQUALIFIER_VOLATILE**

9.32.1 Detailed Description

9.32.2 Variable Documentation

9.32.2.1 CVQUALIFIER_CONST

```
const int kpa::CVQUALIFIER_CONST
```

9.32.2.2 CVQUALIFIER_NONE

```
const int kpa::CVQUALIFIER_NONE
```

9.32.2.3 CVQUALIFIER_VOLATILE

```
const int kpa::CVQUALIFIER_VOLATILE
```

9.33 Numerical codes of builtin types

Variables

- const int **kpa::BUILTIN_VOID**
- const int **kpa::BUILTIN_BOOL**
- const int **kpa::BUILTIN_WCHAR_T**
- const int **kpa::BUILTIN_CHAR**
- const int **kpa::BUILTIN_SIGNED_CHAR**
- const int **kpa::BUILTIN_UNSIGNED_CHAR**
- const int **kpa::BUILTIN_SHORT_INT**
- const int **kpa::BUILTIN_SIGHNED_SHORT_INT**
- const int **kpa::BUILTIN_UNSIGNED_SHORT_INT**
- const int **kpa::BUILTIN_INT**
- const int **kpa::BUILTIN_SIGNED_INT**
- const int **kpa::BUILTIN_UNSIGNED_INT**
- const int **kpa::BUILTIN_LONG_INT**
- const int **kpa::BUILTIN_SIGNED_LONG_INT**
- const int **kpa::BUILTIN_UNSIGNED_LONG_INT**
- const int **kpa::BUILTIN_LONG_LONG_INT**
- const int **kpa::BUILTIN_SIGNED_LONG_LONG_INT**
- const int **kpa::BUILTIN_UNSIGNED_LONG_LONG_INT**
- const int **kpa::BUILTIN_FLOAT**
- const int **kpa::BUILTIN_DOUBLE**
- const int **kpa::BUILTIN_LONG_DOUBLE**

9.33.1 Detailed Description

9.33.2 Variable Documentation

9.33.2.1 BUILTIN_BOOL

```
const int kpa::BUILTIN_BOOL
```

9.33.2.2 BUILTIN_CHAR

```
const int kpa::BUILTIN_CHAR
```

9.33.2.3 BUILTIN_DOUBLE

```
const int kpa::BUILTIN_DOUBLE
```

9.33.2.4 BUILTIN_FLOAT

```
const int kpa::BUILTIN_FLOAT
```

9.33.2.5 BUILTIN_INT

```
const int kpa::BUILTIN_INT
```

9.33.2.6 BUILTIN_LONG_DOUBLE

```
const int kpa::BUILTIN_LONG_DOUBLE
```

9.33.2.7 BUILTIN_LONG_INT

```
const int kpa::BUILTIN_LONG_INT
```

9.33.2.8 BUILTIN_LONG_LONG_INT

```
const int kpa::BUILTIN_LONG_LONG_INT
```

9.33.2.9 BUILTIN_SHORT_INT

```
const int kpa::BUILTIN_SHORT_INT
```

9.33.2.10 BUILTIN_SIGHNED_SHORT_INT

```
const int kpa::BUILTIN_SIGHNED_SHORT_INT
```

9.33.2.11 BUILTIN_SIGNED_CHAR

```
const int kpa::BUILTIN_SIGNED_CHAR
```

9.33.2.12 BUILTIN_SIGNED_INT

```
const int kpa::BUILTIN_SIGNED_INT
```

9.33.2.13 BUILTIN_SIGNED_LONG_INT

```
const int kpa::BUILTIN_SIGNED_LONG_INT
```

9.33.2.14 BUILTIN_SIGNED_LONG_LONG_INT

```
const int kpa::BUILTIN_SIGNED_LONG_LONG_INT
```

9.33.2.15 BUILTIN_UNSIGNED_CHAR

```
const int kpa::BUILTIN_UNSIGNED_CHAR
```

9.33.2.16 BUILTIN_UNSIGNED_INT

```
const int kpa::BUILTIN_UNSIGNED_INT
```

9.33.2.17 BUILTIN_UNSIGNED_LONG_INT

```
const int kpa::BUILTIN_UNSIGNED_LONG_INT
```

9.33.2.18 BUILTIN_UNSIGNED_LONG_LONG_INT

```
const int kpa::BUILTIN_UNSIGNED_LONG_LONG_INT
```

9.33.2.19 BUILTIN_UNSIGNED_SHORT_INT

```
const int kpa::BUILTIN_UNSIGNED_SHORT_INT
```

9.33.2.20 BUILTIN_VOID

```
const int kpa::BUILTIN_VOID
```

9.33.2.21 BUILTIN_WCHAR_T

```
const int kpa::BUILTIN_WCHAR_T
```

9.34 Frontend information

Modules

- **Numerical codes for the compilation unit language**
- **Information about compilation unit**

9.34.1 Detailed Description

9.35 Numerical codes for the compilation unit language

Variables

- const int **kpa::LANGUAGE_C**
- const int **kpa::LANGUAGE_CXX**
- const int **kpa::LANGUAGE_CSHARP**

9.35.1 Detailed Description

9.35.2 Variable Documentation

9.35.2.1 LANGUAGE_C

```
const int kpa::LANGUAGE_C
```

Result returned by getFrontendLanguage. Denotes C language and its dialects.

9.35.2.2 LANGUAGE_CSHARP

```
const int kpa::LANGUAGE_CSHARP
```

Result returned by getFrontendLanguage. Denotes C# language.

9.35.2.3 LANGUAGE_CXX

```
const int kpa::LANGUAGE_CXX
```

Result returned by getFrontendLanguage. Denotes C++ language and its dialects.

9.36 Information about compilation unit

Functions

- int **kpa::getFrontendLanguage** ()

9.36.1 Detailed Description

9.36.2 Function Documentation

9.36.2.1 getFrontendLanguage()

```
int kpa::getFrontendLanguage ( )
```

Returns a numerical code for the language used for translating the current compilation unit.

9.37 Source-sink path

Classes

- class **kpa::Hit**
- class **kpa::SourceSinkPath**
- class **kpa::SourceSinkProcessor**

Typedefs

- typedef **Ptr**< **Hit** > **kpa::HitPtr**
- typedef **Ptr**< **SourceSinkPath** > **kpa::SourceSinkPathPtr**
- typedef **Ptr**< **SourceSinkProcessor** > **kpa::SourceSinkProcessorPtr**

9.37.1 Detailed Description

Source-sink path is a path from source to sink dataflow event occurrence points

9.37.2 Typedef Documentation

9.37.2.1 HitPtr

```
typedef Ptr< Hit> kpa::HitPtr
```

smart pointer to a dataflow point

9.37.2.2 SourceSinkPathPtr

```
typedef Ptr< SourceSinkPath> kpa::SourceSinkPathPtr
```

smart pointer to source-sink path

9.37.2.3 SourceSinkProcessorPtr

```
typedef Ptr< SourceSinkProcessor> kpa::SourceSinkProcessorPtr
```

smart pointer to source-sink processor

9.38 Source-sink analyzers

Classes

- class **kpa::SourceSinkAnalyzer**
- class **kpa::SimpleCondition**

Typedefs

- typedef **Ptr**< **SourceSinkAnalyzer** > **kpa::SourceSinkAnalyzerPtr**

Functions

- virtual **node_t kpa::Hit::getNode** ()=0
- virtual **function_t kpa::Hit::getFunction** ()=0
- virtual **memitem_t kpa::Hit::getMemoryItem** ()=0
- virtual **HitPtr kpa::SourceSinkPath::getSource** ()=0
- virtual **HitPtr kpa::SourceSinkPath::getSink** ()=0
- virtual void **kpa::SourceSinkProcessor::process** (**SourceSinkPathPtr** path)=0
- virtual const char * **kpa::SourceSinkAnalyzer::getName** () const =0
- virtual void **kpa::SourceSinkAnalyzer::analyze** (**function_t**)=0
- virtual void **kpa::SourceSinkAnalyzer::addSourceTrigger** (const **TriggerPtr** &t, DataUpdater *u=NULL, unsigned kind=3)=0
- virtual void **kpa::SourceSinkAnalyzer::addKBSource** (const char *name, DataUpdater *u=NULL, unsigned kind=2)=0
- virtual void **kpa::SourceSinkAnalyzer::addSinkTrigger** (const **TriggerPtr** &t, DataUpdater *u=NULL, unsigned kind=3)=0
- virtual void **kpa::SourceSinkAnalyzer::addKBSink** (const char *name, DataUpdater *u=NULL, unsigned kind=2)=0
- virtual void **kpa::SourceSinkAnalyzer::addCheckTrigger** (const **TriggerPtr** &t)=0
- virtual void **kpa::SourceSinkAnalyzer::addKBCheck** (const char *name)=0
- virtual void **kpa::SourceSinkAnalyzer::addRejectTrigger** (const **TriggerPtr** &t)=0
- virtual void **kpa::SourceSinkAnalyzer::addKBReject** (const char *name)=0
- virtual void **kpa::SourceSinkAnalyzer::setProcessor** (**SourceSinkProcessorPtr**)=0
- virtual void **kpa::SourceSinkAnalyzer::addPropTriggers** (const **TriggerPtr** &t_in, const **TriggerPtr** &t_↩ out, DataUpdater *u=NULL)=0
- **SourceSinkAnalyzerPtr kpa::getConditionalSourceSinkChecker** (const char *name, ThreadContext &threadContext)
- **SourceSinkAnalyzerPtr kpa::getConditionalSourceFBKBGenerator** (const char *name, ThreadContext &threadContext)
- **SourceSinkAnalyzerPtr kpa::getConditionalSinkFBKBGenerator** (const char *name, ThreadContext &threadContext)
- **SourceSinkAnalyzerPtr kpa::getForwardSourceSinkChecker** (const char *name, ThreadContext &threadContext)
- **SourceSinkAnalyzerPtr kpa::getForwardReverseSourceSinkChecker** (const char *name, Thread↩ Context &threadContext)
- **SourceSinkAnalyzerPtr kpa::getForwardSourceFBKBGenerator** (const char *name, ThreadContext &threadContext)
- **SourceSinkAnalyzerPtr kpa::getForwardSinkFBKBGenerator** (const char *name, ThreadContext &threadContext)
- **SourceSinkAnalyzerPtr kpa::getBackwardSourceSinkChecker** (const char *name, ThreadContext &threadContext)
- **SourceSinkAnalyzerPtr kpa::getDirectChecker** (const char *name, ThreadContext &threadContext)

- DataUpdater * **kpa::getEvent** (const char *str, ThreadContext &threadContext, DataUpdater *updater=0)
- virtual **constraint_t kpa::SimpleCondition::createConstraint** ()=0

- DataUpdater * **kpa::getFmtEvent** (const char *str, ThreadContext &threadContext, DataUpdater *updater=0)
- **SimpleCondition** * **kpa::getEQNullConstraint** ()
- DataUpdater * **kpa::getSimpleCondition** (**SimpleCondition** *cnd, ThreadContext &threadContext, DataUpdater *updater)

9.38.1 Detailed Description

Source-sink analyzer runs dataflow analysis from source to sink. If analysis finds sink reachable from source, it will stop the analysis and depending on analyzer kind it will either report an issue or generate an FBKB record.

9.38.2 Typedef Documentation

9.38.2.1 SourceSinkAnalyzerPtr

```
typedef Ptr< SourceSinkAnalyzer> kpa::SourceSinkAnalyzerPtr
```

smart pointer to source-sink analyzer

9.38.3 Function Documentation

9.38.3.1 addCheckTrigger()

```
virtual void kpa::SourceSinkAnalyzer::addCheckTrigger (
            const TriggerPtr & t ) [pure virtual]
```

add check trigger. Check trigger stops the analysis.

9.38.3.2 addKBCheck()

```
virtual void kpa::SourceSinkAnalyzer::addKBCheck (
            const char * name ) [pure virtual]
```

add check using FBKB record.

Parameters

name	specifies FBKB record kind. **registerKBKindForConditionalSocket()** (p. 169) should be called during checker initialization with the same name. `SourceSinkAnalyzerPtr analyzer = ...; ... analyzer->addKBCheck("MY.CHECK");`

9.38.3.3 addKBReject()

```
virtual void kpa::SourceSinkAnalyzer::addKBReject (
            const char * name )  [pure virtual]
```

add reject using FBKB record.

Parameters

name	specifies FBKB record kind. **registerKBKindForConditionalSocket()** (p. 169) should be called during checker initialization with the same name. `SourceSinkAnalyzerPtr analyzer = ...; ... analyzer->addKBCheck("MY.REJECT");`

9.38.3.4 addKBSink()

```
virtual void kpa::SourceSinkAnalyzer::addKBSink (
            const char * name,
            DataUpdater * u = NULL,
            unsigned kind = 2 )  [pure virtual]
```

add sink using FBKB record.

Parameters

name	specifies FBKB record kind. **registerKBKindForConditionalSocket()** (p. 169) should be called during checker initialization with the same name.
u	is optional use it to specify trace events or/and constraints for tracked data
kind	is reserved for future and internal use. `SourceSinkAnalyzerPtr analyzer = ...; ... analyzer->addKBSink("MY.SINK", getFmtEvent("function {f} will process data"));` See also **getFmtEvent** (p. 153)

9.38.3.5 addKBSource()

```
virtual void kpa::SourceSinkAnalyzer::addKBSource (
```

```
            const char * name,
            DataUpdater * u = NULL,
            unsigned kind = 2 )  [pure virtual]
```

add source using FBKB record.

Parameters

name	specifies FBKB record kind. **registerKBKindForConditionalSocket()** (p. 169) should be called during checker initialization with the same name.
u	is optional use it to specify trace events or/and constraints for tracked data
kind	is reserved for future and internal use. `SourceSinkAnalyzerPtr analyzer = ...; ... analyzer->addKBSource("MY.SRC", getFmtEvent("data comes from call to function '{f}'"));` See also **getFmtEvent** (p. 153)

9.38.3.6 addPropTriggers()

```
virtual void kpa::SourceSinkAnalyzer::addPropTriggers (
            const  TriggerPtr & t_in,
            const  TriggerPtr & t_out,
            DataUpdater * u = NULL )  [pure virtual]
```

add prop using triggers.

Parameters

t_in	specifies incoming dataflow which is propagated in the node into outcoming dataflow.
t_out	specifies corresponding outcoming dataflow which is propagated in the node from incoming dataflow.
u	is optional use it to specify trace events or/and constraints for tracked data

9.38.3.7 addRejectTrigger()

```
virtual void kpa::SourceSinkAnalyzer::addRejectTrigger (
            const  TriggerPtr & t )  [pure virtual]
```

add reject trigger. Reject trigger filters out specific data.

9.38.3.8 addSinkTrigger()

```
virtual void kpa::SourceSinkAnalyzer::addSinkTrigger (
            const  TriggerPtr & t,
            DataUpdater * u = NULL,
            unsigned kind = 3 )  [pure virtual]
```

add sink trigger to analyzer.

Parameters

t	trigger to add
u	is optional use it to specify trace events or/and constraints for tracked data
kind	is reserved for future and internal use. `SourceSinkAnalyzerPtr analyzer = ...; ... analyzer->addSinkTrigger(new MyTrigger, getFmtEvent("my message"));` See also **getFmtEvent** (p. 153)

9.38.3.9 addSourceTrigger()

```
virtual void kpa::SourceSinkAnalyzer::addSourceTrigger (
            const  TriggerPtr & t,
            DataUpdater * u = NULL,
            unsigned kind = 3 )  [pure virtual]
```

add source trigger to analyzer.

Parameters

t	trigger to add
u	is optional use it to specify trace events or/and constraints for tracked data
kind	is reserved for future and internal use. `SourceSinkAnalyzerPtr analyzer = ...; ... analyzer->addSourceTrigger(new MyTrigger, getFmtEvent("my message"));` See also **getFmtEvent** (p. 153)

9.38.3.10 analyze()

```
virtual void kpa::SourceSinkAnalyzer::analyze (
             function_t  )  [pure virtual]
```

runs analysis on function

9.38.3.11 createConstraint()

```
virtual  constraint_t kpa::SimpleCondition::createConstraint ( )  [pure virtual]
```

9.38.3.12 getBackwardSourceSinkChecker()

```
SourceSinkAnalyzerPtr kpa::getBackwardSourceSinkChecker (
            const char * name,
            ThreadContext & threadContext )
```

returns backward source-sink analyzer for issue detection.

Parameters

name	will be used as issue kind. Returns lightweight checker that will not eliminate infeasible paths.

9.38.3.13 getConditionalSinkFBKBGenerator()

```
SourceSinkAnalyzerPtr kpa::getConditionalSinkFBKBGenerator (
            const char * name,
            ThreadContext & threadContext )
```

returns source-sink analyzer for Sink FBKB records.

Parameters

name	will be used as FBKB record kind. Returned analyzer will eliminate infeasible paths, but may work slower than ForwardSinkFBKBGenerator Usually is configured with same sinks as checker and uses InputTrigger as a source. See **getInputTrigger()** (p. 166)

9.38.3.14 getConditionalSourceFBKBGenerator()

```
SourceSinkAnalyzerPtr kpa::getConditionalSourceFBKBGenerator (
            const char * name,
            ThreadContext & threadContext )
```

returns source-sink analyzer for Source FBKB records.

Parameters

name	will be used as FBKB record kind. Returned analyzer will eliminate infeasible paths, but may work slower than ForwardSourceFBKBGenerator Usually is configured with same sources as checker and uses OutputTrigger as a sink. See **getOutputTrigger()** (p. 167) and **getReturnTrigger()** (p. 168)

9.38.3.15 getConditionalSourceSinkChecker()

```
SourceSinkAnalyzerPtr kpa::getConditionalSourceSinkChecker (
            const char * name,
            ThreadContext & threadContext )
```

returns source-sink analyzer for issue detection.

Parameters

name	will be used as issue kind. Returned analyzer will eliminate infeasible paths, but may work slower than ForwardSourceSinkChecker

9.38.3.16 getDirectChecker()

```
SourceSinkAnalyzerPtr kpa::getDirectChecker (
            const char * name,
            ThreadContext & threadContext )
```

returns direct analyzer for issue detection.

Parameters

name	will be used as issue kind. Returned lightweight checker traverses nodes and detects sources and sinks, making no difference between them. Each triggered source or sink is treated as both source and sink.

9.38.3.17 getEQNullConstraint()

```
SimpleCondition* kpa::getEQNullConstraint ( )
```

returns constraint provider for 'not equal to 0' constraint

9.38.3.18 getEvent()

```
DataUpdater* kpa::getEvent (
            const char * str,
            ThreadContext & threadContext,
            DataUpdater * updater = 0 )
```

adds event to issue traceback

Parameters

str	string for event message

9.38.3.19 getFmtEvent()

```
DataUpdater* kpa::getFmtEvent (
            const char * str,
```

```
            ThreadContext & threadContext,
            DataUpdater * updater = 0 )
```

adds event to issue traceback

Parameters

str	format string for event message Supports specifiers: {F} - function name {L} - line number {C} - column number {v} - variable name {f} - called function name {n} - call argument number \ - adds \ • adds {

9.38.3.20 getForwardReverseSourceSinkChecker()

```
SourceSinkAnalyzerPtr kpa::getForwardReverseSourceSinkChecker (
            const char * name,
            ThreadContext & threadContext )
```

returns forward source-sink analyzer for reverse issue detection.

Parameters

name	will be used as issue kind. Returns lightweight checker that will not eliminate infeasible paths.

9.38.3.21 getForwardSinkFBKBGenerator()

```
SourceSinkAnalyzerPtr kpa::getForwardSinkFBKBGenerator (
            const char * name,
            ThreadContext & threadContext )
```

returns forward source-sink analyzer for Sink FBKB records.

Parameters

name	will be used as FBKB record kind. Returns lightweight FBKB generator that will not eliminate infeasible paths.

9.38.3.22 getForwardSourceFBKBGenerator()

```
SourceSinkAnalyzerPtr kpa::getForwardSourceFBKBGenerator (
            const char * name,
            ThreadContext & threadContext )
```

returns forward source-sink analyzer for Source FBKB records.

Parameters

name	will be used as FBKB record kind. Returns lightweight FBKB generator that will not eliminate infeasible paths.

9.38.3.23 getForwardSourceSinkChecker()

```
SourceSinkAnalyzerPtr kpa::getForwardSourceSinkChecker (
            const char * name,
            ThreadContext & threadContext )
```

returns forward source-sink analyzer for issue detection.

Parameters

name	will be used as issue kind. Returns lightweight checker that will not eliminate infeasible paths.

9.38.3.24 getFunction()

```
virtual  function_t kpa::Hit::getFunction ( )  [pure virtual]
```

9.38.3.25 getMemoryItem()

```
virtual  memitem_t kpa::Hit::getMemoryItem ( )  [pure virtual]
```

9.38.3.26 getName()

```
virtual const char* kpa::SourceSinkAnalyzer::getName ( ) const  [pure virtual]
```

returns name of source-sink analyzer instance.

9.38.3.27 getNode()

```
virtual  node_t kpa::Hit::getNode ( )  [pure virtual]
```

9.38.3.28 getSimpleCondition()

```
DataUpdater* kpa::getSimpleCondition (
              SimpleCondition * cnd,
             ThreadContext & threadContext,
             DataUpdater * updater )
```

data updater that adds constraint data tracked by source-sink checker. May be used to configure sources for conditional source-sink analyzers

9.38.3.29 getSink()

```
virtual  HitPtr kpa::SourceSinkPath::getSink ( )  [pure virtual]
```

9.38.3.30 getSource()

```
virtual  HitPtr kpa::SourceSinkPath::getSource ( )  [pure virtual]
```

9.38.3.31 process()

```
virtual void kpa::SourceSinkProcessor::process (
              SourceSinkPathPtr path )  [pure virtual]
```

processes a source-sink path

9.38.3.32 setProcessor()

```
virtual void kpa::SourceSinkAnalyzer::setProcessor (
              SourceSinkProcessorPtr  )  [pure virtual]
```

set custom processor to process detected source-sink paths, It will overwrite default processor which reports found paths as issues

9.39 Triggers

Classes

- class **kpa::TriggerResult**
- class **kpa::Trigger**

Typedefs

- typedef **Ptr**< **Trigger** > **kpa::TriggerPtr**

9.39.1 Detailed Description

9.39.2 Typedef Documentation

9.39.2.1 TriggerPtr

`typedef` **Ptr**< **Trigger**> **kpa::TriggerPtr**

smart pointer for that manages trigger

Chapter 10

Namespace Documentation

10.1 kpa Namespace Reference

Classes

- class **DescriptorAccepter**
- struct **edgeIterator_tag**
- class **Hit**
- class **integer_t**
- class **NodeCollection**
- class **Ptr**
- class **RefCnt**
- class **RefCounter**
- class **SimpleCondition**
- class **SourceSinkAnalyzer**
- class **SourceSinkPath**
- class **SourceSinkProcessor**
- class **Trigger**
- class **TriggerResult**

Typedefs

- typedef struct mir_tFunction * **function_t**
- typedef size_t **sema_t**
- typedef struct mir_tNode * **node_t**
- typedef struct mir_tEdge * **edge_t**
- typedef union mir_tExpression * **expr_t**
- typedef struct mir_tBasicBlock * **bb_t**
- typedef const struct MemoryItem * **memitem_t**
- typedef struct NumericRange * **constraint_t**
- typedef void(* **functionHook_t**) (**function_t**, ThreadContext &)
- typedef struct **kpa::edgeIterator_tag** **edgeIterator_t**
- typedef **Ptr**< **NodeCollection** > **NodeCollectionPtr**
- typedef struct tLEFPosition * **position_t**
- typedef void * **trace_t**
- typedef void * **message_t**
- typedef **Ptr**< **Hit** > **HitPtr**
- typedef **Ptr**< **SourceSinkPath** > **SourceSinkPathPtr**
- typedef **Ptr**< **SourceSinkProcessor** > **SourceSinkProcessorPtr**
- typedef **Ptr**< **SourceSinkAnalyzer** > **SourceSinkAnalyzerPtr**
- typedef **Ptr**< **Trigger** > **TriggerPtr**
- typedef **Ptr**< **DescriptorAccepter** > **DescriptorAccepterPtr**

Functions

- const char * **func_getName** (**function_t** func)
- **integer_t** **operator+** (const signed long long lhs_const, const **integer_t** &rhs)
- **integer_t** **operator+** (const signed int lhs_const, const **integer_t** &rhs)
- **integer_t** **operator+** (const unsigned long long lhs_const, const **integer_t** &rhs)
- **integer_t** **operator+** (const unsigned int lhs_const, const **integer_t** &rhs)
- **integer_t** **operator-** (const signed long long lhs_const, const **integer_t** &rhs)
- **integer_t** **operator-** (const signed int lhs_const, const **integer_t** &rhs)
- **integer_t** **operator-** (const unsigned long long lhs_const, const **integer_t** &rhs)
- **integer_t** **operator-** (const unsigned int lhs_const, const **integer_t** &rhs)
- **integer_t** **operator*** (const signed long long lhs_const, const **integer_t** &rhs)
- **integer_t** **operator*** (const signed int lhs_const, const **integer_t** &rhs)
- **integer_t** **operator*** (const unsigned long long lhs_const, const **integer_t** &rhs)
- **integer_t** **operator*** (const unsigned int lhs_const, const **integer_t** &rhs)
- **integer_t** **operator/** (const signed long long lhs_const, const **integer_t** &rhs)
- **integer_t** **operator/** (const signed int lhs_const, const **integer_t** &rhs)
- **integer_t** **operator/** (const unsigned long long lhs_const, const **integer_t** &rhs)
- **integer_t** **operator/** (const unsigned int lhs_const, const **integer_t** &rhs)
- **integer_t** **operator%** (const signed long long lhs_const, const **integer_t** &rhs)
- **integer_t** **operator%** (const signed int lhs_const, const **integer_t** &rhs)
- **integer_t** **operator%** (const unsigned long long lhs_const, const **integer_t** &rhs)
- **integer_t** **operator%** (const unsigned int lhs_const, const **integer_t** &rhs)
- **integer_t** **operator|** (const signed long long lhs_const, const **integer_t** &rhs)
- **integer_t** **operator|** (const signed int lhs_const, const **integer_t** &rhs)
- **integer_t** **operator|** (const unsigned long long lhs_const, const **integer_t** &rhs)
- **integer_t** **operator|** (const unsigned int lhs_const, const **integer_t** &rhs)
- **integer_t** **operator&** (const signed long long lhs_const, const **integer_t** &rhs)
- **integer_t** **operator&** (const signed int lhs_const, const **integer_t** &rhs)
- **integer_t** **operator&** (const unsigned long long lhs_const, const **integer_t** &rhs)
- **integer_t** **operator&** (const unsigned int lhs_const, const **integer_t** &rhs)
- **integer_t** **operator^** (const signed long long lhs_const, const **integer_t** &rhs)
- **integer_t** **operator^** (const signed int lhs_const, const **integer_t** &rhs)
- **integer_t** **operator^** (const unsigned long long lhs_const, const **integer_t** &rhs)
- **integer_t** **operator^** (const unsigned int lhs_const, const **integer_t** &rhs)
- bool **operator==** (const signed long long lhs_const, const **integer_t** &rhs)
- bool **operator==** (const signed int lhs_const, const **integer_t** &rhs)
- bool **operator==** (const unsigned long long lhs_const, const **integer_t** &rhs)
- bool **operator==** (const unsigned int lhs_const, const **integer_t** &rhs)
- bool **operator!=** (const signed long long lhs_const, const **integer_t** &rhs)
- bool **operator!=** (const signed int lhs_const, const **integer_t** &rhs)
- bool **operator!=** (const unsigned long long lhs_const, const **integer_t** &rhs)
- bool **operator!=** (const unsigned int lhs_const, const **integer_t** &rhs)
- bool **operator<** (const signed long long lhs_const, const **integer_t** &rhs)
- bool **operator<** (const signed int lhs_const, const **integer_t** &rhs)
- bool **operator<** (const unsigned long long lhs_const, const **integer_t** &rhs)
- bool **operator<** (const unsigned int lhs_const, const **integer_t** &rhs)
- bool **operator>** (const signed long long lhs_const, const **integer_t** &rhs)
- bool **operator>** (const signed int lhs_const, const **integer_t** &rhs)
- bool **operator>** (const unsigned long long lhs_const, const **integer_t** &rhs)
- bool **operator>** (const unsigned int lhs_const, const **integer_t** &rhs)
- bool **operator<=** (const signed long long lhs_const, const **integer_t** &rhs)
- bool **operator<=** (const signed int lhs_const, const **integer_t** &rhs)
- bool **operator<=** (const unsigned long long lhs_const, const **integer_t** &rhs)
- bool **operator<=** (const unsigned int lhs_const, const **integer_t** &rhs)

- bool **operator>=** (const signed long long lhs_const, const **integer_t** &rhs)
- bool **operator>=** (const signed int lhs_const, const **integer_t** &rhs)
- bool **operator>=** (const unsigned long long lhs_const, const **integer_t** &rhs)
- bool **operator>=** (const unsigned int lhs_const, const **integer_t** &rhs)
- void **registerFunctionHook** (**functionHook_t** function_hook)
- void **registerKBGeneratorFunctionHook** (**functionHook_t** function_hook)
- **expr_t** **node_getReadExpression** (**node_t** n)
- **expr_t** **node_getWrittenExpression** (**node_t** n)
- int **node_isExpression** (**node_t** node)
- int **node_isSwitch** (**node_t** node)
- int **node_isConditionalBranch** (**node_t** node)
- int **node_isLeaf** (**node_t** node)
- int **node_isReturn** (**node_t** node)
- int **node_isBreak** (**node_t** node)
- int **node_isContinue** (**node_t** node)
- int **node_isInitialization** (**node_t** node)
- int **node_isThrow** (**node_t** node)
- int **node_getOutDegree** (**node_t** node)
- int **node_getInDegree** (**node_t** node)
- **edgeIterator_t** **node_getInEdgeSet** (**node_t** node)
- **edgeIterator_t** **node_getOutEdgeSet** (**node_t** node)
- int **edgeIterator_valid** (**edgeIterator_t** it)
- void **edgeIterator_next** (**edgeIterator_t** *it)
- **edge_t** **edgeIterator_value** (**edgeIterator_t** it)
- int **edge_getKind** (**edge_t** edge)
- **node_t** **edge_getStartNode** (**edge_t** edge)
- **node_t** **edge_getEndNode** (**edge_t** edge)
- int **expr_isCallTo** (**expr_t** mir_expr, const char *func_name)
- int **expr_isCallToQualified** (**expr_t** mir_expr, const char *func_name)
- const char * **expr_getCallName** (**expr_t** expr, ThreadContext &threadContext)
- const char * **expr_getCallQualifiedName** (**expr_t** expr, ThreadContext &threadContext)
- const char * **expr_getCallFBKBName** (**expr_t** expr)
- int **expr_getNumberOfArguments** (**expr_t** call_expr)
- **expr_t** **expr_getCallArgument** (**expr_t** call_expr, int argnum)
- **memitem_t** **expr_getMemitem** (**expr_t** mir_expr, ThreadContext &threadContext)
- int **expr_isVariable** (**expr_t** expr)
- int **expr_isFunction** (**expr_t** expr)
- int **expr_isConstantValue** (**expr_t** expr)
- int **expr_isIntegerConstant** (**expr_t** expr)
- long long **expr_getIntegerConstantValue** (**expr_t** expr, int *error_flag)
- **integer_t** **expr_getIntegerConstantValue** (**expr_t** expr)
- int **expr_isStringConstant** (**expr_t** expr)
- char * **expr_getStringConstantValue** (**expr_t** expr)
- int **expr_isFloatConstant** (**expr_t** expr)
- long double **expr_getFloatConstantValue** (**expr_t** expr, int *error_flag)
- int **expr_isSizeofConstant** (**expr_t** expr)
- int **expr_getSizeofConstantValue** (**expr_t** expr, int *error_flag)
- int **expr_isAddress** (**expr_t** expr)
- int **expr_isIndex** (**expr_t** expr)
- int **expr_isDereference** (**expr_t** expr)
- int **expr_isField** (**expr_t** expr)
- int **expr_isMember** (**expr_t** expr)
- int **expr_isCall** (**expr_t** expr)
- int **expr_isBinaryOperation** (**expr_t** expr)
- int **expr_isUnaryOperation** (**expr_t** expr)

- int **expr_isTemporaryRegister** (**expr_t** expr)
- **NodeCollectionPtr getDefinitionNodeForTemporary** (**expr_t** temp, **node_t** n, ThreadContext &thread↩ Context)
- int **expr_isParameter** (**expr_t** expr)
- **expr_t expr_getUnaryOperand** (**expr_t** unary_expr)
- **expr_t expr_getBinaryOperand1** (**expr_t** binary_expr)
- **expr_t expr_getBinaryOperand2** (**expr_t** binary_expr)
- **expr_t expr_getAddressed** (**expr_t** addr_expr)
- int **expr_getParameterNumber** (**expr_t** param_expr)
- **expr_t expr_getDereferenced** (**expr_t** deref_expr)
- **expr_t expr_getIndexBase** (**expr_t** index_expr)
- **expr_t expr_getIndexOffset** (**expr_t** index_expr)
- **constraint_t expr_getOffsetValue** (**node_t** node, **expr_t** index_expr, int ∗error_flag, ThreadContext &threadContext)
- **expr_t expr_getFieldBase** (**expr_t** field_expr)
- **expr_t expr_getFieldMember** (**expr_t** field_expr)
- **expr_t expr_getCalled** (**expr_t** call_expr)
- int **expr_getOperationCode** (**expr_t** binary_or_unary_expr)
- **memitem_t extractMemoryItem** (**expr_t** expr, ThreadContext &threadContext)
- **memitem_t memitem_getPointed** (**memitem_t** mi)
- **memitem_t memitem_getPointer** (**memitem_t** mi)
- **memitem_t memitem_getParent** (**memitem_t** mi)
- const char ∗ **memitem_getName** (**memitem_t** mi)
- int **memitem_isGlobal** (**memitem_t** mi)
- int **memitem_isStatic** (**memitem_t** mi)
- int **memitem_isLocal** (**memitem_t** mi)
- int **memitem_isTemporary** (**memitem_t** mi)
- int **memitem_isFunctionArgument** (**memitem_t** mi)
- int **memitem_isAddress** (**memitem_t** mi)
- int **memitem_isPointer** (**memitem_t** mi)
- int **memitem_isPointerToConst** (**memitem_t** mi)
- int **memitem_isClass** (**memitem_t** mi)
- int **memitem_isBuiltin** (**memitem_t** mi)
- int **memitem_isUnion** (**memitem_t** mi)
- int **memitem_isInstantiation** (**memitem_t** mi)
- int **memitem_isArray** (**memitem_t** mi)
- int **memitem_isUnknown** (**memitem_t** mi)
- int **memitem_isArrowField** (**memitem_t** mi)
- **sema_t memitem_getSemanticInfo** (**memitem_t** mi)
- **sema_t memitem_getTypeSemanticInfo** (**memitem_t** mi)
- int **memitemUsage** (**node_t** node, **memitem_t** mi, ThreadContext &threadContext)
- **memitem_t memitemGetAliased** (**node_t** node, **memitem_t** mi, ThreadContext &threadContext)
- **constraint_t bb_getPreConstraint** (**memitem_t** mi, **bb_t** bb, **function_t** func, ThreadContext &thread↩ Context)
- **constraint_t bb_getPostConstraint** (**memitem_t** mi, **bb_t** bb, **function_t** func, ThreadContext &thread↩ Context)
- **constraint_t mi_getNodePreConstraint** (**memitem_t** mi, **node_t** node, **function_t** func, ThreadContext &threadContext)
- int **constraint_isValue** (**constraint_t** cons)
- int **constraint_getValue** (**constraint_t** cons)
- int **constraint_getMinValue** (**constraint_t** cons, long int ∗a)
- bool **constraint_hasMinValue** (**constraint_t** cons)
- **integer_t constraint_getMinValue** (**constraint_t** cons)
- int **constraint_getMaxValue** (**constraint_t** cons, long int ∗a)
- bool **constraint_hasMaxValue** (**constraint_t** cons)

- **integer_t constraint_getMaxValue** (**constraint_t** cons)
- int **constraint_isGE** (**constraint_t** cons, long int ∗a)
- int **constraint_isLE** (**constraint_t** cons, long int ∗a)
- int **constraint_isInterval** (**constraint_t** cons, long int ∗a, long int ∗b)
- int **constraint_isNE** (**constraint_t** cons, long int ∗a)
- **integer_t constraint_getNEValue** (**constraint_t** cons)
- int **constraint_isEQ** (**constraint_t** cons, long int ∗a)
- **integer_t constraint_getEQValue** (**constraint_t** cons)
- bool **constraint_containsValue** (**constraint_t** cons, const **integer_t** &value)
- bool **constraint_containsNoValues** (**constraint_t** cons)
- bool **constraint_containsAllValues** (**constraint_t** cons)
- void **constraint_toString** (**constraint_t** cons, char ∗buf, size_t bufsize)
- void **constraint_delete** (**constraint_t** cons)
- **position_t node_getPosition** (**node_t** node)
- int **position_getLine** (**position_t** pos)
- int **position_getColumn** (**position_t** pos)
- **trace_t trace_new** (**function_t** func)
- void ∗ **event_new** (void ∗func, void ∗pos, const char ∗event)
- void **event_setParameter** (void ∗event, const char ∗name, const char ∗value)
- void **trace_addEvent** (**trace_t** trace, void ∗func, **position_t** pos, const char ∗event)
- void **trace_addEventEx** (**trace_t** trace, void ∗event)
- void **trace_delete** (**trace_t** trace)
- **message_t message_new** (const char ∗error_id)
- void **message_setPosition** (**message_t** msg, **position_t** pos)
- void **message_setRecommendationFactor** (**message_t** msg, const char ∗factor, int value)
- void **message_addAttribute** (**message_t** msg, const char ∗attr_string)
- void **message_addAnchorAttribute** (**message_t** msg, const char ∗attr_string)
- void **message_addTrace** (**message_t** msg, **trace_t** trace)
- void **message_render** (**message_t** msg)
- void **message_delete** (**message_t** msg)
- **sema_t expr_getSemanticInfo** (**expr_t** expr)
- **sema_t func_getSemanticInfo** (**function_t** func)
- const char ∗ **sema_getName** (**sema_t** si)
- const char ∗ **sema_getQualifiedName** (**sema_t** si)
- int **sema_isFunction** (**sema_t** si)
- **sema_t sema_getFunctionType** (**sema_t** si)
- int **sema_getNumberOfArguments** (**sema_t** si)
- **sema_t sema_getFormalArgument** (**sema_t** si, int argnum)
- **sema_t sema_getFunctionReturnType** (**sema_t** si)
- int **sema_isPointer** (**sema_t** si)
- int **sema_isReference** (**sema_t** si)
- int **sema_isVariable** (**sema_t** si)
- int **sema_isType** (**sema_t** si)
- **sema_t sema_getVariableType** (**sema_t** si)
- int **sema_isBuiltin** (**sema_t** si)
- int **sema_isClass** (**sema_t** si)
- int **sema_isUnion** (**sema_t** si)
- int **sema_isEnum** (**sema_t** si)
- int **sema_getBuiltin** (**sema_t** si)
- int **sema_getCVQualifiers** (**sema_t** si)
- **sema_t sema_getPointedType** (**sema_t** si)
- bool **sema_isBaseClass** (**sema_t** base, **sema_t** derived)
- **sema_t sema_getNextBaseClass** (**sema_t** cl, **sema_t** base)
- **sema_t sema_getParent** (**sema_t** si)
- int **memoryChangedInCall** (**expr_t** call_expr, int argnum)

- int **getFrontendLanguage** ()
- **SourceSinkAnalyzerPtr getConditionalSourceSinkChecker** (const char *name, ThreadContext &threadContext)
- **SourceSinkAnalyzerPtr getConditionalSourceFBKBGenerator** (const char *name, ThreadContext &threadContext)
- **SourceSinkAnalyzerPtr getConditionalSinkFBKBGenerator** (const char *name, ThreadContext &threadContext)
- **SourceSinkAnalyzerPtr getForwardSourceSinkChecker** (const char *name, ThreadContext &threadContext)
- **SourceSinkAnalyzerPtr getForwardReverseSourceSinkChecker** (const char *name, ThreadContext &threadContext)
- **SourceSinkAnalyzerPtr getForwardSourceFBKBGenerator** (const char *name, ThreadContext &threadContext)
- **SourceSinkAnalyzerPtr getForwardSinkFBKBGenerator** (const char *name, ThreadContext &threadContext)
- **SourceSinkAnalyzerPtr getBackwardSourceSinkChecker** (const char *name, ThreadContext &threadContext)
- **SourceSinkAnalyzerPtr getDirectChecker** (const char *name, ThreadContext &threadContext)
- DataUpdater * **getEvent** (const char *str, ThreadContext &threadContext, DataUpdater *updater=0)
- **TriggerPtr getInputTrigger** (ThreadContext &threadContext, const **DescriptorAccepterPtr** &mi_accepter)
- **TriggerPtr getInputTrigger** (ThreadContext &threadContext)
- **TriggerPtr getOutputTrigger** (ThreadContext &threadContext, const **DescriptorAccepterPtr** &mi_accepter)
- **TriggerPtr getOutputTrigger** (ThreadContext &threadContext)
- **TriggerPtr getReturnTrigger** (ThreadContext &threadContext, const **DescriptorAccepterPtr** &mi_accepter)
- **TriggerPtr getReturnTrigger** (ThreadContext &threadContext)
- **DescriptorAccepterPtr getPointedAccepter** ()
- **DescriptorAccepterPtr getPrimarilyPointedAccepter** ()
- **DescriptorAccepterPtr getPrimaryWithFieldsAccepter** ()
- bool **memitem_traverseFields** (**memitem_t** memitem, **DescriptorAccepterPtr** accepter, ThreadContext &threadContext)
- bool **function_traverseLocals** (**function_t** function, **DescriptorAccepterPtr** accepter, ThreadContext &threadContext)
- bool **registerKBKindForConditionalSocket** (const char *kind)

- DataUpdater * **getFmtEvent** (const char *str, ThreadContext &threadContext, DataUpdater *updater=0)
- **SimpleCondition** * **getEQNullConstraint** ()
- DataUpdater * **getSimpleCondition** (**SimpleCondition** *cnd, ThreadContext &threadContext, DataUpdater *updater)

Variables

- const int **OPCODE_NONE**
- const int **OPCODE_ADD**
- const int **OPCODE_ADDRESS**
- const int **OPCODE_ASL**
- const int **OPCODE_ASR**
- const int **OPCODE_BITAND**
- const int **OPCODE_BITNOT**
- const int **OPCODE_BITOR**

- const int **OPCODE_BITXOR**
- const int **OPCODE_CAST**
- const int **OPCODE_DEREF**
- const int **OPCODE_DIV**
- const int **OPCODE_EQ**
- const int **OPCODE_GE**
- const int **OPCODE_GT**
- const int **OPCODE_IDIV**
- const int **OPCODE_LE**
- const int **OPCODE_LOGAND**
- const int **OPCODE_LOGNOT**
- const int **OPCODE_LOGOR**
- const int **OPCODE_LT**
- const int **OPCODE_MAX**
- const int **OPCODE_MIN**
- const int **OPCODE_MOD**
- const int **OPCODE_UMOD**
- const int **OPCODE_MUL**
- const int **OPCODE_NE**
- const int **OPCODE_SIZEOF**
- const int **OPCODE_SUB**
- const int **OPCODE_THROW**
- const int **MI_NO_ACTION**
- const int **MI_MIGHT_BE_READ**
- const int **MI_IS_READ**
- const int **MI_MIGHT_BE_CHANGED**
- const int **MI_IS_CHANGED**
- const int **MI_ALIASED**
- const int **MI_IS_READ_PARTIALLY**
- const int **MI_IS_READ_INDIRECTLY**
- const int **MI_IS_OVERWRITTEN**
- const int **CVQUALIFIER_NONE**
- const int **CVQUALIFIER_CONST**
- const int **CVQUALIFIER_VOLATILE**
- const int **BUILTIN_VOID**
- const int **BUILTIN_BOOL**
- const int **BUILTIN_WCHAR_T**
- const int **BUILTIN_CHAR**
- const int **BUILTIN_SIGNED_CHAR**
- const int **BUILTIN_UNSIGNED_CHAR**
- const int **BUILTIN_SHORT_INT**
- const int **BUILTIN_SIGHNED_SHORT_INT**
- const int **BUILTIN_UNSIGNED_SHORT_INT**
- const int **BUILTIN_INT**
- const int **BUILTIN_SIGNED_INT**
- const int **BUILTIN_UNSIGNED_INT**
- const int **BUILTIN_LONG_INT**
- const int **BUILTIN_SIGNED_LONG_INT**
- const int **BUILTIN_UNSIGNED_LONG_INT**
- const int **BUILTIN_LONG_LONG_INT**
- const int **BUILTIN_SIGNED_LONG_LONG_INT**
- const int **BUILTIN_UNSIGNED_LONG_LONG_INT**
- const int **BUILTIN_FLOAT**
- const int **BUILTIN_DOUBLE**
- const int **BUILTIN_LONG_DOUBLE**

- const int **MEMCHANGE_NOT_A_CALL**
- const int **MEMCHANGE_NO_SEMANTIC_INFO**
- const int **MEMCHANGE_NOT_A_FUNCTION**
- const int **MEMCHANGE_NOT_A_POINTER**
- const int **MEMCHANGE_INVALID_ARGUMENT**
- const int **MEMCHANGE_NOT_CHANGED**
- const int **MEMCHANGE_MAY_BE_CHANGED**
- const int **MEMCHANGE_CHANGED**
- const int **LANGUAGE_C**
- const int **LANGUAGE_CXX**
- const int **LANGUAGE_CSHARP**

- const int **EDGE_TRUE**
- const int **EDGE_FALSE**
- const int **EDGE_CONDITIONAL**
- const int **EDGE_UNCONDITIONAL**

10.1.1 Typedef Documentation

10.1.1.1 DescriptorAccepterPtr

```
typedef Ptr< DescriptorAccepter> kpa::DescriptorAccepterPtr
```

10.1.2 Function Documentation

10.1.2.1 function_traverseLocals()

```
bool kpa::function_traverseLocals (
            function_t function,
            DescriptorAccepterPtr accepter,
           ThreadContext & threadContext )
```

Traverse memory items for local variables declared in the function

Parameters

accepter	is an accepter which is called for each declared variable Traverses variables while accepter returns 'true', stops otherwise. Returns 'true' if traverse hasn't been stopped by accepter

10.1.2.2 getInputTrigger() [1/2]

```
TriggerPtr kpa::getInputTrigger (
            ThreadContext & threadContext,
            const  DescriptorAccepterPtr & mi_accepter )
```

returns trigger that will extract function arguments and their fields from function entry node

Parameters

mi_accepter	filters InputTrigger result. See **DescriptorAccepter** (p. 171)

10.1.2.3 getInputTrigger() [2/2]

```
TriggerPtr kpa::getInputTrigger (
            ThreadContext & threadContext )
```

returns trigger that will extract function arguments and their fields from function entry node

10.1.2.4 getOutputTrigger() [1/2]

```
TriggerPtr kpa::getOutputTrigger (
            ThreadContext & threadContext,
            const  DescriptorAccepterPtr & mi_accepter )
```

returns trigger that will extract function arguments passed by reference and their fields from function exit node

Parameters

mi_accepter	filters OutputTrigger result. See **DescriptorAccepter** (p. 171)

10.1.2.5 getOutputTrigger() [2/2]

```
TriggerPtr kpa::getOutputTrigger (
            ThreadContext & threadContext )
```

returns trigger that will extract function arguments passed by reference and their fields from function exit node

10.1.2.6 getPointedAccepter()

```
DescriptorAccepterPtr kpa::getPointedAccepter ( )
```

10.1.2.7 getPrimarilyPointedAccepter()

```
DescriptorAccepterPtr kpa::getPrimarilyPointedAccepter ( )
```

10.1.2.8 getPrimaryWithFieldsAccepter()

```
DescriptorAccepterPtr kpa::getPrimaryWithFieldsAccepter ( )
```

10.1.2.9 getReturnTrigger() [1/2]

```
TriggerPtr kpa::getReturnTrigger (
            ThreadContext & threadContext,
            const  DescriptorAccepterPtr & mi_accepter )
```

returns trigger that will extract function return value and it's fields from return node

Parameters

mi_accepter	filters ReturnTrigger result. See **DescriptorAccepter** (p. 171)

10.1.2.10 getReturnTrigger() [2/2]

```
TriggerPtr kpa::getReturnTrigger (
            ThreadContext & threadContext )
```

returns trigger that will extract function return value and it's fields from return node

10.1.2.11 memitem_traverseFields()

```
bool kpa::memitem_traverseFields (
             memitem_t memitem,
             DescriptorAccepterPtr accepter,
            ThreadContext & threadContext )
```

Traverse field memory items of a given memory item

Parameters

memitem	is a memory item to traverse its fields
accepter	is an accepter which is called for each field memory item Traverses fields while accepter returns 'true', stops otherwise. Returns 'true' if traverse hasn't been stopped by accepter

10.1.2.12 registerKBKindForConditionalSocket()

```
bool kpa::registerKBKindForConditionalSocket (
            const char * kind )
```

registers FBKB record kind

Parameters

kind	names user FBKB record kind

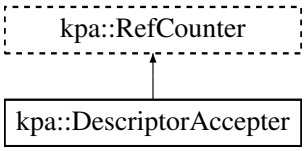
Chapter 11

Class Documentation

11.1 kpa::DescriptorAccepter Class Reference

`#include <kpaUtil.hh>`

Inheritance diagram for kpa::DescriptorAccepter:



Public Member Functions

- virtual bool **accepts** (**memitem_t**)=0

Protected Member Functions

- virtual ~**DescriptorAccepter** ()

11.1.1 Detailed Description

interface to filter results of some builtin triggers

11.1.2 Constructor & Destructor Documentation

11.1.2.1 ~DescriptorAccepter()

`virtual kpa::DescriptorAccepter::~DescriptorAccepter ( ) [inline], [protected], [virtual]`

11.1.3 Member Function Documentation

11.1.3.1 accepts()

```
virtual bool kpa::DescriptorAccepter::accepts (
              memitem_t  ) [pure virtual]
```

should return false if memory item should be filtered out

The documentation for this class was generated from the following file:

- **kpaUtil.hh**

11.2 kpa::edgeIterator_tag Struct Reference

```
#include <kpaMirUtil.hh>
```

Public Attributes

- size_t **n**
- void * **data**

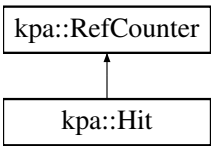
The documentation for this struct was generated from the following file:

- **kpaMirUtil.hh**

11.3 kpa::Hit Class Reference

```
#include <kpaSourceSinkAnalyzer.hh>
```

Inheritance diagram for kpa::Hit:



Public Member Functions

- virtual **node_t getNode** ()=0
- virtual **function_t getFunction** ()=0
- virtual **memitem_t getMemoryItem** ()=0

11.3.1 Detailed Description

Interface to a dataflow event occurrence point

The documentation for this class was generated from the following file:

- **kpaSourceSinkAnalyzer.hh**

11.4 kpa::integer_t Class Reference

`#include <kpaMirUtil.hh>`

Public Member Functions

- **integer_t** ()
- **integer_t** (signed char i)
- **integer_t** (unsigned char i)
- **integer_t** (signed short i)
- **integer_t** (unsigned short i)
- **integer_t** (signed int i)
- **integer_t** (unsigned int i)
- **integer_t** (signed long long i)
- **integer_t** (unsigned long long i)
- **integer_t** (const **integer_t** &other)
- ∼**integer_t** ()
  Destructor of an integer.
- **integer_t** & **operator=** (signed char i)
- **integer_t** & **operator=** (unsigned char i)
- **integer_t** & **operator=** (signed short i)
- **integer_t** & **operator=** (unsigned short i)
- **integer_t** & **operator=** (signed int i)
- **integer_t** & **operator=** (unsigned int i)
- **integer_t** & **operator=** (signed long long i)
- **integer_t** & **operator=** (unsigned long long i)
- **integer_t** & **operator=** (const **integer_t** &other)
- bool **isValid** () const
- long long **getInt64** () const
- unsigned long long **getUint64** () const
- char ∗ **toCharPtr** () const
- **integer_t** **castToType** (**sema_t** si) const
- **integer_t** **castToType** (**memitem_t** mi) const
- **integer_t** **castToType** (**expr_t** expr) const
- **integer_t** **operator+** (const **integer_t** &rhs) const
- **integer_t** **operator-** (const **integer_t** &rhs) const
- **integer_t** **operator**∗ (const **integer_t** &rhs) const
- **integer_t** **operator**/ (const **integer_t** &rhs) const
- **integer_t** **operator%** (const **integer_t** &rhs) const
- **integer_t** **operator-** () const
- **integer_t** **operator+** () const
- **integer_t** **operator**∼ () const
- **integer_t** & **operator++** ()

- **integer_t operator++** (int)
- **integer_t** & **operator--** ()
- **integer_t operator--** (int)
- void **operator+=** (const **integer_t** &rhs)
- void **operator-=** (const **integer_t** &rhs)
- void **operator*=** (const **integer_t** &rhs)
- void **operator/=** (const **integer_t** &rhs)
- void **operator%=** (const **integer_t** &rhs)
- bool **operator>** (const **integer_t** &rhs) const
- bool **operator<** (const **integer_t** &rhs) const
- bool **operator>=** (const **integer_t** &rhs) const
- bool **operator<=** (const **integer_t** &rhs) const
- bool **operator==** (const **integer_t** &rhs) const
- bool **operator!=** (const **integer_t** &rhs) const
- **integer_t operator<<** (int shift) const
- **integer_t operator>>** (int shift) const
- **integer_t operator<<** (const **integer_t** &rhs) const
- **integer_t operator>>** (const **integer_t** &rhs) const
- **integer_t operator &** (const **integer_t** &rhs) const
- **integer_t operator|** (const **integer_t** &rhs) const
- **integer_t operator^** (const **integer_t** &rhs) const
- void **operator<<=** (int shift)
- void **operator>>=** (int shift)
- void **operator &=** (const **integer_t** &rhs)
- void **operator|=** (const **integer_t** &rhs)
- void **operator^=** (const **integer_t** &rhs)
- bool **operator!** () const
- void **setPimpl** (void *x)
- const void * **getPimpl** () const

11.4.1 Detailed Description

Fixed precision integer class used in Klocwork Path Analysis. The purpose of the class is to handle arithmetic operations on both signed and unsigned 64 bit integers.

In the future, the precision might be increased without modifying the interface.

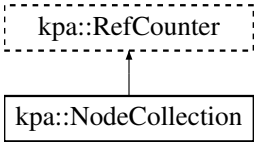
The documentation for this class was generated from the following file:

- **kpaMirUtil.hh**

11.5 kpa::NodeCollection Class Reference

`#include <kpaMirUtil.hh>`

Inheritance diagram for kpa::NodeCollection:



Public Member Functions

- virtual size_t **size** () const =0
- virtual **node_t get** (unsigned index) const =0
- virtual ~**NodeCollection** ()

11.5.1 Detailed Description

Get set of MIR nodes that assign value temporary variable temp, used in node n

The documentation for this class was generated from the following file:

- **kpaMirUtil.hh**

11.6 kpa::Ptr< T > Class Template Reference

```
#include <kpaRefCounting.hh>
```

Public Member Functions

- **Ptr** (T *p)
- **Ptr** ()
- **Ptr** (const **Ptr**< T > &src)
- ~**Ptr** ()
- T * **operator=** (T *p)
- T * **getPtr** () const
- T & **operator*** () const
- T * **operator->** () const
- **Ptr**< T > & **operator=** (const **Ptr**< T > &src)
- template<class X >
  T * **operator=** (const **Ptr**< X > &src)
- **operator T*** () const
- bool **operator<** (T *p) const
- bool **operator==** (T *p) const
- bool **operator!=** (T *p) const
- bool **operator!** () const
- **operator bool** () const
- template<class X >
  operator Ptr< **X** > () const

11.6.1 Detailed Description

template<class T>
class kpa::Ptr< T >

Smart pointer that works with **RefCounter** (p. 180) interface.

11.6.2 Constructor & Destructor Documentation

11.6.2.1 Ptr() [1/3]

```
template<class T>
kpa::Ptr< T >::  Ptr (
            T * p )  [inline]
```

constructs smart pointer object that owns pointer 'p'

11.6.2.2 Ptr() [2/3]

```
template<class T>
kpa::Ptr< T >::  Ptr ( )  [inline]
```

11.6.2.3 Ptr() [3/3]

```
template<class T>
kpa::Ptr< T >::  Ptr (
            const  Ptr< T > & src )  [inline]
```

copy constructor

11.6.2.4 ~Ptr()

```
template<class T>
kpa::Ptr< T >::~ Ptr ( )  [inline]
```

11.6.3 Member Function Documentation

11.6.3.1 getPtr()

```
template<class T>
T*  kpa::Ptr< T >::getPtr ( ) const  [inline]
```

returns pointer. Reference number is not increased. User has to call AddRef/Release manually or use **Ptr** (p. 175)

Referenced by kpa::Ptr< T >::operator=().

11.6.3.2 operator bool()

```
template<class T>
kpa::Ptr< T >::operator bool ( ) const  [inline]
```

11.6.3.3 operator Ptr< X >()

```
template<class T>
template<class X >
kpa::Ptr< T >::operator  Ptr< X > ( ) const  [inline]
```

11.6.3.4 operator T*()

```
template<class T>
kpa::Ptr< T >::operator T* ( ) const  [inline]
```

11.6.3.5 operator"!()

```
template<class T>
bool  kpa::Ptr< T >::operator!  ( ) const  [inline]
```

11.6.3.6 operator"!=()

```
template<class T>
bool  kpa::Ptr< T >::  operator!= (
            T * p ) const  [inline]
```

11.6.3.7 operator*()

```
template<class T>
T&  kpa::Ptr< T >::operator* ( ) const  [inline]
```

operator to dereference pointer

11.6.3.8 operator->()

```
template<class T>
T*  kpa::Ptr< T >::operator-> ( ) const  [inline]
```

operator to dereference pointer

11.6.3.9 operator<()

```
template<class T>
bool  kpa::Ptr< T >::operator< (
            T * p ) const  [inline]
```

11.6.3.10 operator=() [1/3]

```
template<class T>
T*  kpa::Ptr< T >::operator= (
            T * p )  [inline]
```

11.6.3.11 operator=() [2/3]

```
template<class T>
Ptr<T>&  kpa::Ptr< T >::operator= (
            const  Ptr< T > & src )  [inline]
```

11.6.3.12 operator=() [3/3]

```
template<class T>
template<class X >
T*  kpa::Ptr< T >::operator= (
            const  Ptr< X > & src )  [inline]
```

References kpa::Ptr< T >::getPtr().

11.6.3.13 operator==()

```
template<class T>
bool  kpa::Ptr< T >::operator== (
            T * p ) const  [inline]
```

The documentation for this class was generated from the following file:

- **kpaRefCounting.hh**

11.7 kpa::RefCnt Class Reference

```
#include <kpaRefCounting.hh>
```

Public Member Functions

- void **inc** ()
- unsigned **dec** ()
- **RefCnt** ()
- **RefCnt** (const **RefCnt** &)
- **RefCnt** & **operator=** (const **RefCnt** &)

11.7.1 Constructor & Destructor Documentation

11.7.1.1 RefCnt() [1/2]

```
kpa::RefCnt::RefCnt ( )  [inline]
```

11.7.1.2 RefCnt() [2/2]

```
kpa::RefCnt::RefCnt (
            const  RefCnt &  )  [inline]
```

11.7.2 Member Function Documentation

11.7.2.1 dec()

```
unsigned kpa::RefCnt::dec ( )  [inline]
```

11.7.2.2 inc()

```
void kpa::RefCnt::inc ( )  [inline]
```

11.7.2.3 operator=()

```
RefCnt& kpa::RefCnt::operator= (
            const  RefCnt &  )  [inline]
```

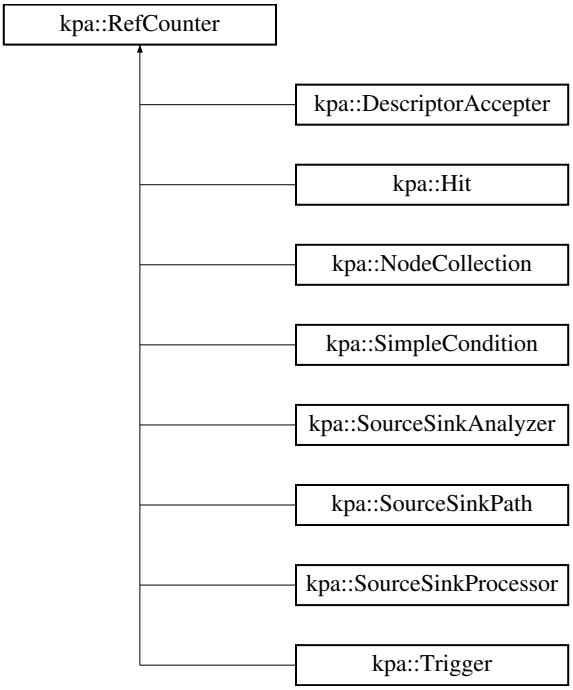
The documentation for this class was generated from the following file:

- **kpaRefCounting.hh**

11.8 kpa::RefCounter Class Reference

`#include <kpaRefCounting.hh>`

Inheritance diagram for kpa::RefCounter:



Public Member Functions

- virtual ∼**RefCounter** ()
- virtual void **AddRef** ()=0
- virtual void **Release** ()=0

11.8.1 Detailed Description

Interface used by Path API to count references to objects and autorelease objects without references. Use macro **REF_COUNTING_IMPL** (p. 195) to add implementation of this interface to your class.

11.8.2 Constructor & Destructor Documentation

11.8.2.1 ∼RefCounter()

`virtual kpa::RefCounter::~RefCounter ( ) [inline], [virtual]`

11.8.3 Member Function Documentation

11.8.3.1 AddRef()

```
virtual void kpa::RefCounter::AddRef ( )  [pure virtual]
```

Called to indicate that reference to object is added

11.8.3.2 Release()

```
virtual void kpa::RefCounter::Release ( )  [pure virtual]
```

Called to indicate that reference to object is removed

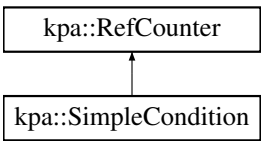
The documentation for this class was generated from the following file:

- **kpaRefCounting.hh**

11.9 kpa::SimpleCondition Class Reference

```
#include <kpaSourceSinkAnalyzer.hh>
```

Inheritance diagram for kpa::SimpleCondition:



Public Member Functions

- virtual **constraint_t createConstraint** ()=0

11.9.1 Detailed Description

constraint provider interface

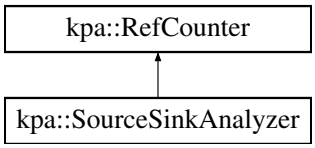
The documentation for this class was generated from the following file:

- **kpaSourceSinkAnalyzer.hh**

11.10 kpa::SourceSinkAnalyzer Class Reference

`#include <kpaSourceSinkAnalyzer.hh>`

Inheritance diagram for kpa::SourceSinkAnalyzer:



Public Member Functions

- virtual const char * **getName** () const =0
- virtual void **analyze** (**function_t**)=0
- virtual void **addSourceTrigger** (const **TriggerPtr** &t, DataUpdater *u=NULL, unsigned kind=3)=0
- virtual void **addKBSource** (const char *name, DataUpdater *u=NULL, unsigned kind=2)=0
- virtual void **addSinkTrigger** (const **TriggerPtr** &t, DataUpdater *u=NULL, unsigned kind=3)=0
- virtual void **addKBSink** (const char *name, DataUpdater *u=NULL, unsigned kind=2)=0
- virtual void **addCheckTrigger** (const **TriggerPtr** &t)=0
- virtual void **addKBCheck** (const char *name)=0
- virtual void **addRejectTrigger** (const **TriggerPtr** &t)=0
- virtual void **addKBReject** (const char *name)=0
- virtual void **setProcessor** (**SourceSinkProcessorPtr**)=0
- virtual void **addPropTriggers** (const **TriggerPtr** &t_in, const **TriggerPtr** &t_out, DataUpdater *u=NULL)=0

11.10.1 Detailed Description

Interface to source-sink analyzer.

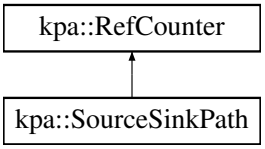
The documentation for this class was generated from the following file:

- **kpaSourceSinkAnalyzer.hh**

11.11 kpa::SourceSinkPath Class Reference

`#include <kpaSourceSinkAnalyzer.hh>`

Inheritance diagram for kpa::SourceSinkPath:



Public Member Functions

- virtual **HitPtr getSource** ()=0
- virtual **HitPtr getSink** ()=0

11.11.1 Detailed Description

Interface to source-sink path

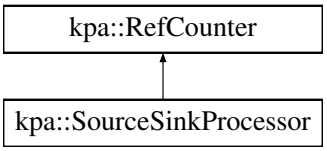
The documentation for this class was generated from the following file:

- **kpaSourceSinkAnalyzer.hh**

11.12 kpa::SourceSinkProcessor Class Reference

`#include <kpaSourceSinkAnalyzer.hh>`

Inheritance diagram for kpa::SourceSinkProcessor:



Public Member Functions

- virtual void **process** (**SourceSinkPathPtr** path)=0

11.12.1 Detailed Description

Interface to source-sink processor

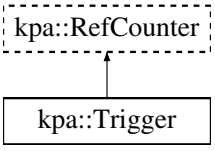
The documentation for this class was generated from the following file:

- **kpaSourceSinkAnalyzer.hh**

11.13 kpa::Trigger Class Reference

`#include <kpaTrigger.hh>`

Inheritance diagram for kpa::Trigger:



Public Member Functions

- **Trigger** (ThreadContext &ctx)
- virtual void **extract** (**node_t** node, **TriggerResult** *res)=0
- virtual ~**Trigger** ()

Protected Attributes

- ThreadContext & **threadContext**

11.13.1 Detailed Description

trigger interface

11.13.2 Constructor & Destructor Documentation

11.13.2.1 Trigger()

```
kpa::Trigger::Trigger (
            ThreadContext & ctx )  [inline]
```

11.13.2.2 ~Trigger()

```
virtual kpa::Trigger::~Trigger ( )  [inline], [virtual]
```

11.13.3 Member Function Documentation

11.13.3.1 extract()

```
virtual void kpa::Trigger::extract (
            node_t node,
            TriggerResult * res )  [pure virtual]
```

frame work calls this method to get information from the trigger

Parameters

node	- node that trigger should analyze
res	- trigger output.

11.13.4 Member Data Documentation

11.13.4.1 threadContext

```
ThreadContext& kpa::Trigger::threadContext  [protected]
```

The documentation for this class was generated from the following file:

- **kpaTrigger.hh**

11.14 kpa::TriggerResult Class Reference

```
#include <kpaTrigger.hh>
```

Public Member Functions

- virtual void **add** (**expr_t**, ThreadContext &)=0
- virtual void **add** (**memitem_t**, ThreadContext &)=0

Protected Member Functions

- virtual ~**TriggerResult** ()

11.14.1 Detailed Description

interface to collect result from a trigger

11.14.2 Constructor & Destructor Documentation

11.14.2.1 ~TriggerResult()

```
virtual kpa::TriggerResult::~TriggerResult ( )  [inline], [protected], [virtual]
```

11.14.3 Member Function Documentation

11.14.3.1 add() [1/2]

```
virtual void kpa::TriggerResult::add (
              expr_t ,
             ThreadContext &  )  [pure virtual]
```

custom triggers should call this method to add expressions extracted from a node

11.14.3.2 add() [2/2]

```
virtual void kpa::TriggerResult::add (
              memitem_t ,
             ThreadContext &  )  [pure virtual]
```

call this method if expression that should be extracted is not available in the current node.

The documentation for this class was generated from the following file:

- **kpaTrigger.hh**

Chapter 12

File Documentation

12.1 kpaAPI.h File Reference

Macros

- #define **KPA_API_VERSION_MAJOR** 3
- #define **KPA_API_VERSION_MINOR** 0
- #define **KPA_API_VERSION_PATCHLEVEL** 0

12.1.1 Macro Definition Documentation

12.1.1.1 KPA_API_VERSION_MAJOR

```
#define KPA_API_VERSION_MAJOR 3
```

12.1.1.2 KPA_API_VERSION_MINOR

```
#define KPA_API_VERSION_MINOR 0
```

12.1.1.3 KPA_API_VERSION_PATCHLEVEL

```
#define KPA_API_VERSION_PATCHLEVEL 0
```

12.2 kpaAPI_MainDoc.h File Reference

12.3 kpaMirUtil.hh File Reference

```
#include <stdlib.h>
#include "kwapi.h"
#include "kpaRefCounting.hh"
```

Classes

- class **kpa::integer_t**
- struct **kpa::edgeIterator_tag**
- class **kpa::NodeCollection**

Namespaces

- **kpa**

Macros

- #define **DECLARE_INT_OP_INTEGER_T**(__X_OP__, __RETURN_TYPE__)

Typedefs

- typedef struct mir_tFunction * **kpa::function_t**
- typedef size_t **kpa::sema_t**
- typedef struct mir_tNode * **kpa::node_t**
- typedef struct mir_tEdge * **kpa::edge_t**
- typedef union mir_tExpression * **kpa::expr_t**
- typedef struct mir_tBasicBlock * **kpa::bb_t**
- typedef const struct MemoryItem * **kpa::memitem_t**
- typedef struct NumericRange * **kpa::constraint_t**
- typedef void(* **kpa::functionHook_t**) (function_t, ThreadContext &)
- typedef struct **kpa::edgeIterator_tag** **kpa::edgeIterator_t**
- typedef Ptr< NodeCollection > **kpa::NodeCollectionPtr**
- typedef struct tLEFPosition * **kpa::position_t**
- typedef void * **kpa::trace_t**
- typedef void * **kpa::message_t**

Functions

- const char * **kpa::func_getName** (function_t func)
- integer_t **kpa::operator+** (const signed long long lhs_const, const integer_t &rhs)
- integer_t **kpa::operator+** (const signed int lhs_const, const integer_t &rhs)
- integer_t **kpa::operator+** (const unsigned long long lhs_const, const integer_t &rhs)
- integer_t **kpa::operator+** (const unsigned int lhs_const, const integer_t &rhs)
- integer_t **kpa::operator-** (const signed long long lhs_const, const integer_t &rhs)
- integer_t **kpa::operator-** (const signed int lhs_const, const integer_t &rhs)
- integer_t **kpa::operator-** (const unsigned long long lhs_const, const integer_t &rhs)
- integer_t **kpa::operator-** (const unsigned int lhs_const, const integer_t &rhs)
- integer_t **kpa::operator*** (const signed long long lhs_const, const integer_t &rhs)
- integer_t **kpa::operator*** (const signed int lhs_const, const integer_t &rhs)
- integer_t **kpa::operator*** (const unsigned long long lhs_const, const integer_t &rhs)
- integer_t **kpa::operator*** (const unsigned int lhs_const, const integer_t &rhs)
- integer_t **kpa::operator/** (const signed long long lhs_const, const integer_t &rhs)
- integer_t **kpa::operator/** (const signed int lhs_const, const integer_t &rhs)
- integer_t **kpa::operator/** (const unsigned long long lhs_const, const integer_t &rhs)
- integer_t **kpa::operator/** (const unsigned int lhs_const, const integer_t &rhs)
- integer_t **kpa::operator%** (const signed long long lhs_const, const integer_t &rhs)
- integer_t **kpa::operator%** (const signed int lhs_const, const integer_t &rhs)
- integer_t **kpa::operator%** (const unsigned long long lhs_const, const integer_t &rhs)
- integer_t **kpa::operator%** (const unsigned int lhs_const, const integer_t &rhs)
- integer_t **kpa::operator|** (const signed long long lhs_const, const integer_t &rhs)
- integer_t **kpa::operator|** (const signed int lhs_const, const integer_t &rhs)
- integer_t **kpa::operator|** (const unsigned long long lhs_const, const integer_t &rhs)
- integer_t **kpa::operator|** (const unsigned int lhs_const, const integer_t &rhs)
- integer_t **kpa::operator&** (const signed long long lhs_const, const integer_t &rhs)
- integer_t **kpa::operator&** (const signed int lhs_const, const integer_t &rhs)
- integer_t **kpa::operator&** (const unsigned long long lhs_const, const integer_t &rhs)
- integer_t **kpa::operator&** (const unsigned int lhs_const, const integer_t &rhs)
- integer_t **kpa::operator^** (const signed long long lhs_const, const integer_t &rhs)
- integer_t **kpa::operator^** (const signed int lhs_const, const integer_t &rhs)
- integer_t **kpa::operator^** (const unsigned long long lhs_const, const integer_t &rhs)
- integer_t **kpa::operator^** (const unsigned int lhs_const, const integer_t &rhs)
- bool **kpa::operator==** (const signed long long lhs_const, const integer_t &rhs)
- bool **kpa::operator==** (const signed int lhs_const, const integer_t &rhs)
- bool **kpa::operator==** (const unsigned long long lhs_const, const integer_t &rhs)
- bool **kpa::operator==** (const unsigned int lhs_const, const integer_t &rhs)
- bool **kpa::operator!=** (const signed long long lhs_const, const integer_t &rhs)
- bool **kpa::operator!=** (const signed int lhs_const, const integer_t &rhs)
- bool **kpa::operator!=** (const unsigned long long lhs_const, const integer_t &rhs)
- bool **kpa::operator!=** (const unsigned int lhs_const, const integer_t &rhs)
- bool **kpa::operator<** (const signed long long lhs_const, const integer_t &rhs)
- bool **kpa::operator<** (const signed int lhs_const, const integer_t &rhs)
- bool **kpa::operator<** (const unsigned long long lhs_const, const integer_t &rhs)
- bool **kpa::operator<** (const unsigned int lhs_const, const integer_t &rhs)
- bool **kpa::operator>** (const signed long long lhs_const, const integer_t &rhs)
- bool **kpa::operator>** (const signed int lhs_const, const integer_t &rhs)
- bool **kpa::operator>** (const unsigned long long lhs_const, const integer_t &rhs)
- bool **kpa::operator>** (const unsigned int lhs_const, const integer_t &rhs)
- bool **kpa::operator<=** (const signed long long lhs_const, const integer_t &rhs)
- bool **kpa::operator<=** (const signed int lhs_const, const integer_t &rhs)
- bool **kpa::operator<=** (const unsigned long long lhs_const, const integer_t &rhs)
- bool **kpa::operator<=** (const unsigned int lhs_const, const integer_t &rhs)

- bool **kpa::operator>=** (const signed long long lhs_const, const integer_t &rhs)
- bool **kpa::operator>=** (const signed int lhs_const, const integer_t &rhs)
- bool **kpa::operator>=** (const unsigned long long lhs_const, const integer_t &rhs)
- bool **kpa::operator>=** (const unsigned int lhs_const, const integer_t &rhs)
- void **kpa::registerFunctionHook** (functionHook_t function_hook)
- void **kpa::registerKBGeneratorFunctionHook** (functionHook_t function_hook)
- expr_t **kpa::node_getReadExpression** (node_t n)
- expr_t **kpa::node_getWrittenExpression** (node_t n)
- int **kpa::node_isExpression** (node_t node)
- int **kpa::node_isSwitch** (node_t node)
- int **kpa::node_isConditionalBranch** (node_t node)
- int **kpa::node_isLeaf** (node_t node)
- int **kpa::node_isReturn** (node_t node)
- int **kpa::node_isBreak** (node_t node)
- int **kpa::node_isContinue** (node_t node)
- int **kpa::node_isInitialization** (node_t node)
- int **kpa::node_isThrow** (node_t node)
- int **kpa::node_getOutDegree** (node_t node)
- int **kpa::node_getInDegree** (node_t node)
- edgeIterator_t **kpa::node_getInEdgeSet** (node_t node)
- edgeIterator_t **kpa::node_getOutEdgeSet** (node_t node)
- int **kpa::edgeIterator_valid** (edgeIterator_t it)
- void **kpa::edgeIterator_next** (edgeIterator_t *it)
- edge_t **kpa::edgeIterator_value** (edgeIterator_t it)
- int **kpa::edge_getKind** (edge_t edge)
- node_t **kpa::edge_getStartNode** (edge_t edge)
- node_t **kpa::edge_getEndNode** (edge_t edge)
- int **kpa::expr_isCallTo** (expr_t mir_expr, const char *func_name)
- int **kpa::expr_isCallToQualified** (expr_t mir_expr, const char *func_name)
- const char * **kpa::expr_getCallName** (expr_t expr, ThreadContext &threadContext)
- const char * **kpa::expr_getCallQualifiedName** (expr_t expr, ThreadContext &threadContext)
- const char * **kpa::expr_getCallFBKBName** (expr_t expr)
- int **kpa::expr_getNumberOfArguments** (expr_t call_expr)
- expr_t **kpa::expr_getCallArgument** (expr_t call_expr, int argnum)
- memitem_t **kpa::expr_getMemitem** (expr_t mir_expr, ThreadContext &threadContext)
- int **kpa::expr_isVariable** (expr_t expr)
- int **kpa::expr_isFunction** (expr_t expr)
- int **kpa::expr_isConstantValue** (expr_t expr)
- int **kpa::expr_isIntegerConstant** (expr_t expr)
- long long **kpa::expr_getIntegerConstantValue** (expr_t expr, int *error_flag)
- integer_t **kpa::expr_getIntegerConstantValue** (expr_t expr)
- int **kpa::expr_isStringConstant** (expr_t expr)
- char * **kpa::expr_getStringConstantValue** (expr_t expr)
- int **kpa::expr_isFloatConstant** (expr_t expr)
- long double **kpa::expr_getFloatConstantValue** (expr_t expr, int *error_flag)
- int **kpa::expr_isSizeofConstant** (expr_t expr)
- int **kpa::expr_getSizeofConstantValue** (expr_t expr, int *error_flag)
- int **kpa::expr_isAddress** (expr_t expr)
- int **kpa::expr_isIndex** (expr_t expr)
- int **kpa::expr_isDereference** (expr_t expr)
- int **kpa::expr_isField** (expr_t expr)
- int **kpa::expr_isMember** (expr_t expr)
- int **kpa::expr_isCall** (expr_t expr)
- int **kpa::expr_isBinaryOperation** (expr_t expr)
- int **kpa::expr_isUnaryOperation** (expr_t expr)

- int **kpa::expr_isTemporaryRegister** (expr_t expr)
- NodeCollectionPtr **kpa::getDefinitionNodeForTemporary** (expr_t temp, node_t n, ThreadContext &threadContext)
- int **kpa::expr_isParameter** (expr_t expr)
- expr_t **kpa::expr_getUnaryOperand** (expr_t unary_expr)
- expr_t **kpa::expr_getBinaryOperand1** (expr_t binary_expr)
- expr_t **kpa::expr_getBinaryOperand2** (expr_t binary_expr)
- expr_t **kpa::expr_getAddressed** (expr_t addr_expr)
- int **kpa::expr_getParameterNumber** (expr_t param_expr)
- expr_t **kpa::expr_getDereferenced** (expr_t deref_expr)
- expr_t **kpa::expr_getIndexBase** (expr_t index_expr)
- expr_t **kpa::expr_getIndexOffset** (expr_t index_expr)
- constraint_t **kpa::expr_getOffsetValue** (node_t node, expr_t index_expr, int *error_flag, ThreadContext &threadContext)
- expr_t **kpa::expr_getFieldBase** (expr_t field_expr)
- expr_t **kpa::expr_getFieldMember** (expr_t field_expr)
- expr_t **kpa::expr_getCalled** (expr_t call_expr)
- int **kpa::expr_getOperationCode** (expr_t binary_or_unary_expr)
- memitem_t **kpa::extractMemoryItem** (expr_t expr, ThreadContext &threadContext)
- memitem_t **kpa::memitem_getPointed** (memitem_t mi)
- memitem_t **kpa::memitem_getPointer** (memitem_t mi)
- memitem_t **kpa::memitem_getParent** (memitem_t mi)
- const char * **kpa::memitem_getName** (memitem_t mi)
- int **kpa::memitem_isGlobal** (memitem_t mi)
- int **kpa::memitem_isStatic** (memitem_t mi)
- int **kpa::memitem_isLocal** (memitem_t mi)
- int **kpa::memitem_isTemporary** (memitem_t mi)
- int **kpa::memitem_isFunctionArgument** (memitem_t mi)
- int **kpa::memitem_isAddress** (memitem_t mi)
- int **kpa::memitem_isPointer** (memitem_t mi)
- int **kpa::memitem_isPointerToConst** (memitem_t mi)
- int **kpa::memitem_isClass** (memitem_t mi)
- int **kpa::memitem_isBuiltin** (memitem_t mi)
- int **kpa::memitem_isUnion** (memitem_t mi)
- int **kpa::memitem_isInstantiation** (memitem_t mi)
- int **kpa::memitem_isArray** (memitem_t mi)
- int **kpa::memitem_isUnknown** (memitem_t mi)
- int **kpa::memitem_isArrowField** (memitem_t mi)
- sema_t **kpa::memitem_getSemanticInfo** (memitem_t mi)
- sema_t **kpa::memitem_getTypeSemanticInfo** (memitem_t mi)
- int **kpa::memitemUsage** (node_t node, memitem_t mi, ThreadContext &threadContext)
- memitem_t **kpa::memitemGetAliased** (node_t node, memitem_t mi, ThreadContext &threadContext)
- constraint_t **kpa::bb_getPreConstraint** (memitem_t mi, bb_t bb, function_t func, ThreadContext &thread↩ Context)
- constraint_t **kpa::bb_getPostConstraint** (memitem_t mi, bb_t bb, function_t func, ThreadContext &thread↩ Context)
- constraint_t **kpa::mi_getNodePreConstraint** (memitem_t mi, node_t node, function_t func, ThreadContext &threadContext)
- int **kpa::constraint_isValue** (constraint_t cons)
- int **kpa::constraint_getValue** (constraint_t cons)
- int **kpa::constraint_getMinValue** (constraint_t cons, long int *a)
- bool **kpa::constraint_hasMinValue** (constraint_t cons)
- integer_t **kpa::constraint_getMinValue** (constraint_t cons)
- int **kpa::constraint_getMaxValue** (constraint_t cons, long int *a)
- bool **kpa::constraint_hasMaxValue** (constraint_t cons)

- integer_t **kpa::constraint_getMaxValue** (constraint_t cons)
- int **kpa::constraint_isGE** (constraint_t cons, long int *a)
- int **kpa::constraint_isLE** (constraint_t cons, long int *a)
- int **kpa::constraint_isInterval** (constraint_t cons, long int *a, long int *b)
- int **kpa::constraint_isNE** (constraint_t cons, long int *a)
- integer_t **kpa::constraint_getNEValue** (constraint_t cons)
- int **kpa::constraint_isEQ** (constraint_t cons, long int *a)
- integer_t **kpa::constraint_getEQValue** (constraint_t cons)
- bool **kpa::constraint_containsValue** (constraint_t cons, const integer_t &value)
- bool **kpa::constraint_containsNoValues** (constraint_t cons)
- bool **kpa::constraint_containsAllValues** (constraint_t cons)
- void **kpa::constraint_toString** (constraint_t cons, char *buf, size_t bufsize)
- void **kpa::constraint_delete** (constraint_t cons)
- position_t **kpa::node_getPosition** (node_t node)
- int **kpa::position_getLine** (position_t pos)
- int **kpa::position_getColumn** (position_t pos)
- trace_t **kpa::trace_new** (function_t func)
- void * **kpa::event_new** (void *func, void *pos, const char *event)
- void **kpa::event_setParameter** (void *event, const char *name, const char *value)
- void **kpa::trace_addEvent** (trace_t trace, void *func, position_t pos, const char *event)
- void **kpa::trace_addEventEx** (trace_t trace, void *event)
- void **kpa::trace_delete** (trace_t trace)
- message_t **kpa::message_new** (const char *error_id)
- void **kpa::message_setPosition** (message_t msg, position_t pos)
- void **kpa::message_setRecommendationFactor** (message_t msg, const char *factor, int value)
- void **kpa::message_addAttribute** (message_t msg, const char *attr_string)
- void **kpa::message_addAnchorAttribute** (message_t msg, const char *attr_string)
- void **kpa::message_addTrace** (message_t msg, trace_t trace)
- void **kpa::message_render** (message_t msg)
- void **kpa::message_delete** (message_t msg)
- sema_t **kpa::expr_getSemanticInfo** (expr_t expr)
- sema_t **kpa::func_getSemanticInfo** (function_t func)
- const char * **kpa::sema_getName** (sema_t si)
- const char * **kpa::sema_getQualifiedName** (sema_t si)
- int **kpa::sema_isFunction** (sema_t si)
- sema_t **kpa::sema_getFunctionType** (sema_t si)
- int **kpa::sema_getNumberOfArguments** (sema_t si)
- sema_t **kpa::sema_getFormalArgument** (sema_t si, int argnum)
- sema_t **kpa::sema_getFunctionReturnType** (sema_t si)
- int **kpa::sema_isPointer** (sema_t si)
- int **kpa::sema_isReference** (sema_t si)
- int **kpa::sema_isVariable** (sema_t si)
- int **kpa::sema_isType** (sema_t si)
- sema_t **kpa::sema_getVariableType** (sema_t si)
- int **kpa::sema_isBuiltin** (sema_t si)
- int **kpa::sema_isClass** (sema_t si)
- int **kpa::sema_isUnion** (sema_t si)
- int **kpa::sema_isEnum** (sema_t si)
- int **kpa::sema_getBuiltin** (sema_t si)
- int **kpa::sema_getCVQualifiers** (sema_t si)
- sema_t **kpa::sema_getPointedType** (sema_t si)
- bool **kpa::sema_isBaseClass** (sema_t base, sema_t derived)
- sema_t **kpa::sema_getNextBaseClass** (sema_t cl, sema_t base)
- sema_t **kpa::sema_getParent** (sema_t si)
- int **kpa::memoryChangedInCall** (expr_t call_expr, int argnum)
- int **kpa::getFrontendLanguage** ()

Variables

- const int **kpa::OPCODE_NONE**
- const int **kpa::OPCODE_ADD**
- const int **kpa::OPCODE_ADDRESS**
- const int **kpa::OPCODE_ASL**
- const int **kpa::OPCODE_ASR**
- const int **kpa::OPCODE_BITAND**
- const int **kpa::OPCODE_BITNOT**
- const int **kpa::OPCODE_BITOR**
- const int **kpa::OPCODE_BITXOR**
- const int **kpa::OPCODE_CAST**
- const int **kpa::OPCODE_DEREF**
- const int **kpa::OPCODE_DIV**
- const int **kpa::OPCODE_EQ**
- const int **kpa::OPCODE_GE**
- const int **kpa::OPCODE_GT**
- const int **kpa::OPCODE_IDIV**
- const int **kpa::OPCODE_LE**
- const int **kpa::OPCODE_LOGAND**
- const int **kpa::OPCODE_LOGNOT**
- const int **kpa::OPCODE_LOGOR**
- const int **kpa::OPCODE_LT**
- const int **kpa::OPCODE_MAX**
- const int **kpa::OPCODE_MIN**
- const int **kpa::OPCODE_MOD**
- const int **kpa::OPCODE_UMOD**
- const int **kpa::OPCODE_MUL**
- const int **kpa::OPCODE_NE**
- const int **kpa::OPCODE_SIZEOF**
- const int **kpa::OPCODE_SUB**
- const int **kpa::OPCODE_THROW**
- const int **kpa::MI_NO_ACTION**
- const int **kpa::MI_MIGHT_BE_READ**
- const int **kpa::MI_IS_READ**
- const int **kpa::MI_MIGHT_BE_CHANGED**
- const int **kpa::MI_IS_CHANGED**
- const int **kpa::MI_ALIASED**
- const int **kpa::MI_IS_READ_PARTIALLY**
- const int **kpa::MI_IS_READ_INDIRECTLY**
- const int **kpa::MI_IS_OVERWRITTEN**
- const int **kpa::CVQUALIFIER_NONE**
- const int **kpa::CVQUALIFIER_CONST**
- const int **kpa::CVQUALIFIER_VOLATILE**
- const int **kpa::BUILTIN_VOID**
- const int **kpa::BUILTIN_BOOL**
- const int **kpa::BUILTIN_WCHAR_T**
- const int **kpa::BUILTIN_CHAR**
- const int **kpa::BUILTIN_SIGNED_CHAR**
- const int **kpa::BUILTIN_UNSIGNED_CHAR**
- const int **kpa::BUILTIN_SHORT_INT**
- const int **kpa::BUILTIN_SIGHNED_SHORT_INT**
- const int **kpa::BUILTIN_UNSIGNED_SHORT_INT**
- const int **kpa::BUILTIN_INT**
- const int **kpa::BUILTIN_SIGNED_INT**

- const int **kpa::BUILTIN_UNSIGNED_INT**
- const int **kpa::BUILTIN_LONG_INT**
- const int **kpa::BUILTIN_SIGNED_LONG_INT**
- const int **kpa::BUILTIN_UNSIGNED_LONG_INT**
- const int **kpa::BUILTIN_LONG_LONG_INT**
- const int **kpa::BUILTIN_SIGNED_LONG_LONG_INT**
- const int **kpa::BUILTIN_UNSIGNED_LONG_LONG_INT**
- const int **kpa::BUILTIN_FLOAT**
- const int **kpa::BUILTIN_DOUBLE**
- const int **kpa::BUILTIN_LONG_DOUBLE**
- const int **kpa::MEMCHANGE_NOT_A_CALL**
- const int **kpa::MEMCHANGE_NO_SEMANTIC_INFO**
- const int **kpa::MEMCHANGE_NOT_A_FUNCTION**
- const int **kpa::MEMCHANGE_NOT_A_POINTER**
- const int **kpa::MEMCHANGE_INVALID_ARGUMENT**
- const int **kpa::MEMCHANGE_NOT_CHANGED**
- const int **kpa::MEMCHANGE_MAY_BE_CHANGED**
- const int **kpa::MEMCHANGE_CHANGED**
- const int **kpa::LANGUAGE_C**
- const int **kpa::LANGUAGE_CXX**
- const int **kpa::LANGUAGE_CSHARP**

- const int **kpa::EDGE_TRUE**
- const int **kpa::EDGE_FALSE**
- const int **kpa::EDGE_CONDITIONAL**
- const int **kpa::EDGE_UNCONDITIONAL**

12.4 kpaRefCounting.hh File Reference

```
#include "kwapi.h"
#include <atomic>
```

Classes

- class **kpa::RefCnt**
- class **kpa::RefCounter**
- class **kpa::Ptr**< **T** >

Namespaces

- **kpa**

Macros

- #define **REF_COUNTING_IMPL**

12.4.1 Macro Definition Documentation

12.4.1.1 REF_COUNTING_IMPL

```
#define REF_COUNTING_IMPL
```

Value:

```
private: \
    kpa::RefCnt cnt; \
    public: \
    virtual void AddRef() { cnt.inc(); } \
    virtual void Release() { if (cnt.dec() == 0) { delete this; } }
```

Should be used inside class to add implementation to RefCounter interface

```
class C : public RefCounter {
  REF_COUNTING_IMPL
  //Other class declarations
};
```

12.5 kpaSourceSinkAnalyzer.hh File Reference

```
#include "kpaMirUtil.hh"
#include "kpaTrigger.hh"
#include "kpaRefCounting.hh"
```

Classes

- class **kpa::Hit**
- class **kpa::SourceSinkPath**
- class **kpa::SourceSinkProcessor**
- class **kpa::SourceSinkAnalyzer**
- class **kpa::SimpleCondition**

Namespaces

- **kpa**

Typedefs

- typedef Ptr< Hit > **kpa::HitPtr**
- typedef Ptr< SourceSinkPath > **kpa::SourceSinkPathPtr**
- typedef Ptr< SourceSinkProcessor > **kpa::SourceSinkProcessorPtr**
- typedef Ptr< SourceSinkAnalyzer > **kpa::SourceSinkAnalyzerPtr**

Functions

- SourceSinkAnalyzerPtr **kpa::getConditionalSourceSinkChecker** (const char *name, ThreadContext &threadContext)
- SourceSinkAnalyzerPtr **kpa::getConditionalSourceFBKBGenerator** (const char *name, ThreadContext &threadContext)
- SourceSinkAnalyzerPtr **kpa::getConditionalSinkFBKBGenerator** (const char *name, ThreadContext &threadContext)
- SourceSinkAnalyzerPtr **kpa::getForwardSourceSinkChecker** (const char *name, ThreadContext &threadContext)
- SourceSinkAnalyzerPtr **kpa::getForwardReverseSourceSinkChecker** (const char *name, ThreadContext &threadContext)
- SourceSinkAnalyzerPtr **kpa::getForwardSourceFBKBGenerator** (const char *name, ThreadContext &threadContext)
- SourceSinkAnalyzerPtr **kpa::getForwardSinkFBKBGenerator** (const char *name, ThreadContext &threadContext)
- SourceSinkAnalyzerPtr **kpa::getBackwardSourceSinkChecker** (const char *name, ThreadContext &threadContext)
- SourceSinkAnalyzerPtr **kpa::getDirectChecker** (const char *name, ThreadContext &threadContext)
- DataUpdater * **kpa::getEvent** (const char *str, ThreadContext &threadContext, DataUpdater *updater=0)
- DataUpdater * **kpa::getFmtEvent** (const char *str, ThreadContext &threadContext, DataUpdater *updater=0)
- SimpleCondition * **kpa::getEQNullConstraint** ()
- DataUpdater * **kpa::getSimpleCondition** (SimpleCondition *cnd, ThreadContext &threadContext, DataUpdater *updater)

12.6 kpaTrigger.hh File Reference

```
#include "kpaMirUtil.hh"
#include "kpaRefCounting.hh"
```

Classes

- class **kpa::TriggerResult**
- class **kpa::Trigger**

Namespaces

- **kpa**

Typedefs

- typedef Ptr< Trigger > **kpa::TriggerPtr**

12.7 kpaTriggerUtil.hh File Reference

```
#include "kpaTrigger.hh"
#include "kpaUtil.hh"
```

Namespaces

- **kpa**

Functions

- TriggerPtr **kpa::getInputTrigger** (ThreadContext &threadContext, const DescriptorAccepterPtr &mi_↩ accepter)
- TriggerPtr **kpa::getInputTrigger** (ThreadContext &threadContext)
- TriggerPtr **kpa::getOutputTrigger** (ThreadContext &threadContext, const DescriptorAccepterPtr &mi_↩ accepter)
- TriggerPtr **kpa::getOutputTrigger** (ThreadContext &threadContext)
- TriggerPtr **kpa::getReturnTrigger** (ThreadContext &threadContext, const DescriptorAccepterPtr &mi_↩ accepter)
- TriggerPtr **kpa::getReturnTrigger** (ThreadContext &threadContext)

12.8 kpaUtil.hh File Reference

```
#include "kpaMirUtil.hh"
#include "kpaRefCounting.hh"
```

Classes

- class **kpa::DescriptorAccepter**

Namespaces

- **kpa**

Typedefs

- typedef Ptr< DescriptorAccepter > **kpa::DescriptorAccepterPtr**

Functions

- DescriptorAccepterPtr **kpa::getPointedAccepter** ()
- DescriptorAccepterPtr **kpa::getPrimarilyPointedAccepter** ()
- DescriptorAccepterPtr **kpa::getPrimaryWithFieldsAccepter** ()
- bool **kpa::memitem_traverseFields** (memitem_t memitem, DescriptorAccepterPtr accepter, ThreadContext &threadContext)
- bool **kpa::function_traverseLocals** (function_t function, DescriptorAccepterPtr accepter, ThreadContext &threadContext)
- bool **kpa::registerKBKindForConditionalSocket** (const char *kind)

12.9 kwapi.h File Reference

```
#include <stddef.h>
```

Macros

- #define **KWAPI_DECLARE**(type) type
- #define **KWAPI_DECLARE_CPP**(type) type
- #define **KWAPI_DECLARE_NONSTD**(type) type
- #define **KWAPI_DECLARE_DATA**

Typedefs

- typedef size_t **kw_size_t**
- typedef struct ParameterNode * **kwapi_cfgparam_t**
- typedef struct **kw_array_t** **kw_array_t**

Enumerations

- enum **kwapi_apitypes_t** {
  KWAPI_NONE = 0x0, **KWAPI_TREE** = 0x1, **KWAPI_PATH** = 0x2, **KWAPI_PATTERN** = 0x4,
  KWAPI_PREP = 0x8, **KWAPI_LINKER** = 0x10 }
- enum **kwapi_langtypes_t** { **KWAPI_NOLANG**, **KWAPI_JAVA**, **KWAPI_CXX**, **KWAPI_CSHARP** }

Functions

- **kwapi_cfgparam_t** **kwapi_cfgparam_getRootParameterList** (const char *error)
- **kwapi_cfgparam_t** **kwapi_cfgparam_getListNodeByName** (**kwapi_cfgparam_t**, const char *name)
- **kwapi_cfgparam_t** **kwapi_cfgparam_getListNodeByRegexMatchingName** (**kwapi_cfgparam_t**, const char *name)
- const char * **kwapi_cfgparam_getName** (**kwapi_cfgparam_t**)
- const char * **kwapi_cfgparam_getType** (**kwapi_cfgparam_t**)
- **kw_size_t** **kwapi_cfgparam_getListLength** (**kwapi_cfgparam_t**)
- **kwapi_cfgparam_t** **kwapi_cfgparam_getListNodeByIndex** (**kwapi_cfgparam_t**, **kw_size_t** idx)
- int **kwapi_cfgparam_isParameter** (**kwapi_cfgparam_t**)
- const char * **kwapi_cfgparam_getParameterValue** (**kwapi_cfgparam_t**)
- const char * **kwapi_cfgparam_getParameterValueFromList** (**kwapi_cfgparam_t** parent, const char *paramName)
- const char * **kwapi_cfgparam_getConfigurationParameter** (const char *errorId, const char *paramName)
- const char *const * **kwapi_cfgparam_getCheckerErrors** (const char *checker_id)
- const char * **ktc_error_getConfigurationParameter** (const char *errorId, const char *paramName)
- int **kwapi_cfgparam_errorIsEnabled** (const char *error_id)
- **kw_size_t** **kw_array_size** (**kw_array_t** *array)
- void * **kw_array_get** (**kw_array_t** *array, **kw_size_t** index)
- void **kw_array_delete** (**kw_array_t** *array)

12.9.1 Macro Definition Documentation

12.9.1.1 KWAPI_DECLARE

```
#define KWAPI_DECLARE(
              type ) type
```

12.9.1.2 KWAPI_DECLARE_CPP

```
#define KWAPI_DECLARE_CPP(
              type ) type
```

12.9.1.3 KWAPI_DECLARE_DATA

```
#define KWAPI_DECLARE_DATA
```

12.9.1.4 KWAPI_DECLARE_NONSTD

```
#define KWAPI_DECLARE_NONSTD(
              type ) type
```

12.9.2 Typedef Documentation

12.9.2.1 kw_array_t

```
typedef struct  kw_array_t  kw_array_t
```

12.9.2.2 kw_size_t

```
typedef size_t  kw_size_t
```

12.9.3 Enumeration Type Documentation

12.9.3.1 kwapi_apitypes_t

```
enum  kwapi_apitypes_t
```

Constants for Klocwork APIs

Enumerator

KWAPI_NONE	
KWAPI_TREE	
KWAPI_PATH	
KWAPI_PATTERN	
KWAPI_PREP	
KWAPI_LINKER	

12.9.3.2 kwapi_langtypes_t

```
enum kwapi_langtypes_t
```

Constants for Klocwork supported languages

Enumerator

KWAPI_NOLANG	
KWAPI_JAVA	
KWAPI_CXX	
KWAPI_CSHARP	

12.9.4 Function Documentation

12.9.4.1 kw_array_delete()

```
void kw_array_delete (
            kw_array_t * array )
```

12.9.4.2 kw_array_get()

```
void* kw_array_get (
            kw_array_t * array,
            kw_size_t index )
```

12.9.4.3 kw_array_size()

```
kw_size_t kw_array_size (
            kw_array_t * array )
```

Index