



Software Analyzers

The Mthread plug-in

For Frama-C 31.0 (Gallium)

Boris Yakobowski
with Richard Bonichon, David Bühler and Basile Desloges



Work licensed under Creative Commons BY-SA licence
<https://creativecommons.org/licenses/by-sa/4.0/>

CONTENTS


	Foreword	3
1	An introduction to Mthread	4
1.1	What is Mthread?	4
1.2	How Mthread works	4
1.3	Running Mthread	5
2	Mthread theory	6
2.1	Calling contexts	6
2.2	Concurrent control-flow graphs	7
2.2.1	General idea	7
2.2.2	Example	7
2.2.3	Understanding loops in concurrent control flow graphs	8
2.3	Shared zones	9
2.3.1	Protecting shared zones through mutexes	10
2.4	Related works	11
3	Instrumenting the C concurrent primitives	13
3.1	First steps	13
3.2	Stubbing the header (.h) files	13
3.3	Stubbing the source (.c) files	15
3.3.1	pthread library	15
4	Analyzing a full project without warnings	17
4.1	The philosophers example	17
4.1.1	A first try	18
4.1.2	Unrolling loops	20
4.2	Other Mthread warnings	21
5	Reading Mthread results	24
5.1	Reading the results of the philosophers examples	24
5.2	HTML	26
6	Command-line options	30
A	Mthread functions for stubbing	32

FOREWORD

This is the documentation of the **Frama-C** plug-in **Mthread**.

The content of this document corresponds to the version 31.0 (Gallium) of **Frama-C**. However the development of the **Mthread** plug-in is still ongoing: features described here may evolve in the future.

Acknowledgements

 This project has received funding from the ANR project Veridyc (ANR-09-SEGI-016).
--

1.1 What is Mthread?

Mthread is a Frama-C plug-in dedicated to the analysis of concurrent C programs. It finds and displays multithreaded events, such as thread creation, mutex locking, access to shared variables, *etc.*... Mthread then gives a very simplified view of the source code, in which only source statements relevant to the concurrent behavior of the program are left. It also displays variables that are shared between threads, as well as data sent by threads on messages queues. For each shared memory zone, the mutexes that may protect it are automatically inferred, and possible race conditions are reported.

1.2 How Mthread works

Mthread performs sound and precise analyzes of concurrent programs. It is built on top of the Eva analysis of Frama-C, and uses the latter to derive sound values (hence sometimes over-approximated) for all the variables of the program, including those that are shared between multiple threads. Schematically, Mthread's behavior can be summarized as follows:

- Do a symbolic execution of the main thread; find the threads it launches.
- Do a symbolic execution of the new threads, possibly discovering other new threads, which are then also executed symbolically.
- From each thread, compute the set of variables it reads and writes, as well as the messages it tries to receive and send.
- Compute the shared variables of the program, by detecting variables that are accessed concurrently (*ie.* by at least two threads that are live at the same time). On such concurrent accesses, record which mutexes are being hold by the various threads.
- Restart the whole process, reinjecting the results obtained so far:
 - threads receiving messages from a message queue are given the values sent to this queue by the other threads;
 - threads reading shared variables “see” the values they write in those variables, but also those written by the other threads.
- Iterate the process above until all threads agree on the information sent and exchanged during the execution of the program.

Reaching a fixpoint of the above process means that a sound approximation of the behavior of the program has been obtained, by construction. More details on how Mthread works are given in Chapter 2.

1.3 Running Mthread

Mthread is a Frama-C plug-in, and is activated when launching Frama-C by turning on the `-mthread` switch and enabling the Mthread domain with `-eva-domains mthread` at your shell prompt, as follows:

```
| % frama-c <C files> -mthread -eva-domains mthread <mthread options>
```

The various options that configure the behavior of Mthread will be given throughout this document, and are summarized in Appendix 6. In order to be really useful, Mthread however requires you to instrument the thread library used in your C project, as explained in §3.

The schema given in §1.2 already gives a faithful representation of **Mthread** fixpoint-based approach. Through the **Eva** analysis, we obtain information about a thread; we then reinject those information into (future) analyzes of the other threads. Reaching a fixpoint guarantees that all threads agree on the concurrent part of the program, and that we have found an over-approximation of their behavior. The sections below detail some of the computations **Mthread** does during the iterations.

2.1 Calling contexts

Although the approach outlined above is simple, obtaining precise results is not. Indeed, we must be careful not to compute too general behaviors for the various threads, the risk being to get unusable results. **Mthread** uses some callbacks made available by the **Eva** analysis to record the state of each function at the end of its evaluation. In order to avoid losing precision, **Mthread** fuses those states only when the *calling contexts* are the same. Formally, a calling context is the callstack that lead to the execution of f , taking the statements at which the calls originated into account.

```

1  main () {
2      g ();
3  }
4  void g () {
5      int x, y;
6      f(&x, 1);
7      f(&y, 2);
8  }
9  void f (int *p, int v) {
10     *p = v;
11 }
```

In the example above, there are two distinct calling contexts for the function f , namely $\langle \text{main}, 2 \rangle : \langle g, 6 \rangle$ and $\langle \text{main}, 2 \rangle : \langle g, 7 \rangle$. By making a distinction between those two calls, **Mthread** is able to detect that x (*resp.* y) is always affected the value 1 (*resp.* 2). This is much more precise than the information available by only inspecting the state of the **Eva** analysis at the end of the execution, which merges together all the calls to a function. (This can be easily verified by querying the possible values for p and v in the GUI of **Frama-C**, which would lead to conclude that x and y are affected either 1 or 2, without the possibility to know which one).

2.2 Concurrent control-flow graphs

2.2.1 General idea

One result of the analyzes done by `Mthread` is the *concurrent control-flow graph* of each thread. Those graphs aim at displaying all the following *events*:

- calls to a `mthread.h` primitive;
- accesses to a shared memory zone (see §2.3).

Basically, we build a very high-level view of the functions called by a thread, with the following characteristics:

1. only function calls that contain an event, or that transitively lead (through another call) to such an event, appear in the graph;
2. functions are duplicated for each calling context they appear in;
3. inside the body of a function, only events and high-level control-flow statements such as `if` and `loop` appear. Control-flow statements that do not lead to an event are also removed.

Points 1 and 3 guarantee that the graph keeps a reasonable size, even with very big programs. Indeed, most of the code is typically not related to its concurrent structure. Conversely, point 2 expands the size of the graph, but increases its precision. Indeed, the statements executed by a function can be very different from one call to another, and this is captured by our use of calling contexts.

Notice that the concurrent control-flow graph of a thread is very different from what would be obtained with the `Slicing` plugin of `Frama-C`. In particular, our control-flow graph does not represent executable code at all. (For example, incrementations of loop indices are generally removed from the graph.) Conversely, our graph can be more precise when a function is called multiple times, and roughly corresponds to the specialization obtained by `-slicing-level 3`.

2.2.2 Example

The concurrent control-flow graph for the main thread of the example `doc/eva/examples/mthread/ccfg.c`, reprinted below, is given in Figure 2.1. How to generate this graph is explained in §5.2.

```

1  #include "mthread_pthread.h"
2  #define NULL ((void*)0)
3
4  pthread_t  jobs[4];
5  int x, global1, global2[2];
6
7  void *fjob(void *) {
8      int r = global1 + global2[0] + global2[1];
9      return NULL;
10 }
11
12 void gl(int* v, int i) {
13     if (i<4)
14         pthread_create(&jobs[i], NULL, fjob, NULL );
15     else
16         *v = 1;
17 }
18
19 //@ assigns *v, *(v+1) \from \nothing;
20 void g2(int* v);

```

2.2. CONCURRENT CONTROL-FLOW GRAPHS

```
21
22 void main(void) {
23     int i, arr[2];
24     void (*pf)(int*, int) = &g1;
25
26     g1(NULL, 0);
27     g2(arr);
28     for (i=1; i<5; i++)
29         if (!x) {
30             (*pf)(&global1, i);
31             g2(global2);
32         }
33 }
```

Let us illustrate through our example the characteristics of concurrent control-flow graphs that have been mentioned above. The options we hint at are documented in Appendix 6.

- The topmost node contains the name of the function the thread starts with, here `main`.
- Calls to other functions are inlined within the graph, as can be seen *eg.* for `g1`. A dotted grey edge links `Call` nodes to the corresponding `return` ones. (Option `-return-edges`.)
- Functions called twice in two different calling contexts, *eg.* `g1`, are inlined twice. Each body shown represents precisely the execution of the corresponding call. For `g1`, the first call creates the first thread, while the second call has a behavior that depends on `i`.
- For function calls occurring through pointers *eg.* the second call to `g1`, the real name of the function is printed between the `Call` node and the body of the function.
- Calls to functions that do not lead to an event are removed. For example, the call `g2(arr)` does not appear anywhere.
- Nodes with a red border represent immediate calls to one or more `pthread.h` primitives, that are listed in the node. (In this case, only thread creations occur.)
- Nodes with a blue or green background represent accesses to a shared zone of the memory, and are discussed in §2.3.
- Loop nodes represent `while(1)` loops; `for` loops are automatically desugared into `while` ones by Frama-C.
- Diamond nodes appear for a function without definition, *eg.* `g2`. They represent all the events that occur during the call to the function inside a single node. Functions without definition use their ACSL prototype to specify the data they read and write.
- `if` constructs for which the condition is either completely true or completely false in the given context are removed. This is the case for the `if (!x)`, as `x` is always equal to 0 in the program. Similarly, loops whose body do not contain an event are removed. Those simplifications can be deactivated with `-mt-full-cfg`.
- The `exit` node represent the end of the thread, hence the outgoing edge that goes nowhere.
- Although this is not shown here, the `Mthread` graph simplifier is sometimes forced to leave some nodes that do not really contribute to the concurrent structure of the program. This is typically the case for functions that use `gotos`. Those nodes will not have any border or background.

2.2.3 Understanding loops in concurrent control flow graphs

A word must be said on the composite node `Create thread jobs[1..3]` of our example. It must *not* be understood as “at each iteration of the loop, three threads are created”. This is indeed impossible:

- each node corresponds to a single statement, and no `pthread.h` primitive can create three threads in a single statement;
- `Mthread` does not allow the same thread to be launched more than once, as indicated in §4.2.

The correct way to read the graph is the following: at each iteration of the loop, a different thread is

created, with some iterations possibly spawning none. (In fact, in our case the iteration for $i = 4$ does not create a thread.)

2.3 Shared zones

In this section, we call *shared zone* a region of the memory on which a race condition between at least two threads can occur. **Mthread** performs a fine-grained analysis to detect those regions. It proceeds as follows:

1. Once a thread is evaluated by the **Eva** analysis, we compute the global variables it reads and writes, using the **InOut** plugin of **Frama-C**. This plugin uses the results of the **Eva** analysis, thus giving us a sound but quite imprecise over-approximation of the shared zones accessed by this thread. Let us call $Ri(j)$ (*resp.* $Wi(j)$) the zones that are read (*resp.* written) by the thread j .
2. After a full iteration (once all threads have been computed), we find all the zones that are read by at least one thread and written by at least one another.

$$RWi = \bigcup_{j,k,j \neq k} (Ri(j) \cap Wi(k))$$

This set over-approximates the shared zones, and we call it *potential shared zones*.

3. For each thread that accesses a zone in RWi we start another **Eva** analysis, and *watch* the zones above: at the end of the execution of any function, if it reads or writes a zone in RWi , we record a **Mthread** event for this access. This event will thus appear in the control-flow graph for the current thread.
Let $Rp^z(j)$ (*resp.* $Wp^z(j)$) be the set of those precise read (*resp.* write) events relative to the zone z , for the thread j .
4. Once all the needed threads have been recomputed, we compute the threads that are live on each point of the control-flow graphs. Let us note $live(j@e)$ the fact that the thread j is live at the node containing the event e .

By definition, there is potentially a race condition on a zone z if it is written by one thread and read by another, both threads being live at the same time. Thus, for each zone z of RWi , we define the fact it is shared by:

$$shared(z) = \exists j, k, j \neq k \wedge (\exists e_j \in Rp^z(j), \exists e_k \in Wp^z(k), live(j@e_k) \wedge live(k@e_j))$$

(Of course, this is only the mathematical definition of the **shared** predicate. The computations themselves are done efficiently, to avoid the cubic complexity of the formulas above.)

The definition of **shared** is as precise as possible given the information available to **Mthread**, while remaining sound. In particular, it avoids flagging as shared an important set of variables, those that are only initialized (*ie.* written) by the main thread, and used (*ie.* read) afterward by the various threads. As long as the initialization occurs before the creation of any of the threads that access the variable, this variable is *not* shared.¹

Once all the analyzes are finished, **Mthread** classifies the events representing accesses to potential shared zones in three categories. Let us consider an event e for an access to a zone z .

Non-shared access This means that z is in fact *not* a shared zone. Although z was in RWi , there is never any race condition when accessing this zone. Since those zones are not important, e is not shown in the control-flow graph by default. This can be overridden by the option `-mt-non-shared-accesses` if desired.

¹ In fact, to reduce the time spent computing shared zones, **Mthread** completely ignores all the accesses that occur before the creation of the first thread.

Shared, non-concurrent access *Mthread* has determined that z is indeed a shared zone. However, the particular event represented by e is not concurrent, because all the other threads that access z are either not created yet, or canceled. This is typically the case for most initializations of shared zones by the main thread. In the control-flow graph, those events are shown in green. The option `-mt-no-non-concurrent-accesses` can be used to hide them if desired.

Concurrent access The zone z is indeed a shared zone, and the access is concurrent. That is, a race condition is possible at e . It is shown in blue in the control-flow graph.

An example of the various cases above can be found in the file `src/plugins/eva/tests/mthread/sharedvars.c`, which we do not reproduce below for space consideration. Of the 6 variables of the programs, 3 are shared (those starting by 's') and 3 are not (those starting by 'u'). Running *Mthread* on it with the option `-mt-verbose 3` is concluded by

```
[mt] Imprecise locations to watch: u3; s4; s5; s6
[mt] Possible read/write data races:
s6:
  read by jobs4 at sharedvars.c:56, unprotected
  read by jobs6 at sharedvars.c:75, unprotected
  write by jobs4 at sharedvars.c:57, unprotected
  write by jobs6 at sharedvars.c:76, unprotected
s5:
  read by jobs51 at sharedvars.c:62, unprotected
  write by jobs51 at sharedvars.c:63, unprotected
  write by jobs5 at sharedvars.c:69, unprotected
s4:
  read by jobs4 at sharedvars.c:53, unprotected
  write by <main> at sharedvars.c:100, unprotected
  write by jobs4 at sharedvars.c:54, unprotected
[mt] Shared memory: s4; s5; s6
```

The line “Imprecise locations to watch” indicates that the potential shared zones are the variables `u3`, `s4`, `s5` and `s6`. The section “Possible read/write data races” and the line “Shared memory” however indicates that `u3` is not really shared.

Also, examining the control-flow graph of the main thread shows a non-concurrent access to `s4` before the creation of `&jobs4`. This access is not listed above, as it is not concurrent —and thus must not be taken into account when examining the mutexes that protect `s4`.

2.3.1 Protecting shared zones through mutexes

The race conditions evoked in the previous section are theoretical. That is, they can be prevented using an appropriate use of mutexes. However, once all the shared zones have been found, *Mthread* needs to do very little more to have this information.

Indeed, for each access to a shared zone (*ie.* an event in the control-flow graph), we know which mutexes are locked, and which are not. Thus, in its final output, when *Mthread* lists all the shared zones it has detected, it adds the information it possesses about mutexes. Mutexes that are guaranteed to be locked are written directly. Mutexes that may or may not be locked (*eg.* that are locked in one branch of the program, but not in another) are prefixed by (?).

In a second time, *Mthread* combines those information together and list for each zone the mutexes that are either possibly or systematically locked when the zone is accessed. A shared zone that is protected by at least one guaranteed mutex will not be subject to a race condition.

Since `sharedvars.c` does not use mutexes at all, it is not very pertinent here. Some examples of protection outputs are given in §5.1.

2.4 Related works

Ferrara [Fer09] uses a fixpoint-based approach very similar to ours to analyze Java bytecode. The static analyzer Locksmith [HFP06], which is dedicated to finding data races in multithreaded C programs, possess some similarities with our shared zones detection algorithm. The Goblint [VV07] is race-detection tool using some fixpoint computation (resolved by a constraint solver): it offers a path-sensitive analysis of data-races, based upon conditional constraint propagation and points-to analysis. Miné [Min12] builds an analyzer for concurrent code on top of the Astree abstract interpreter. Apart from the use of two distinct base analysers, our approach and his are very similar.

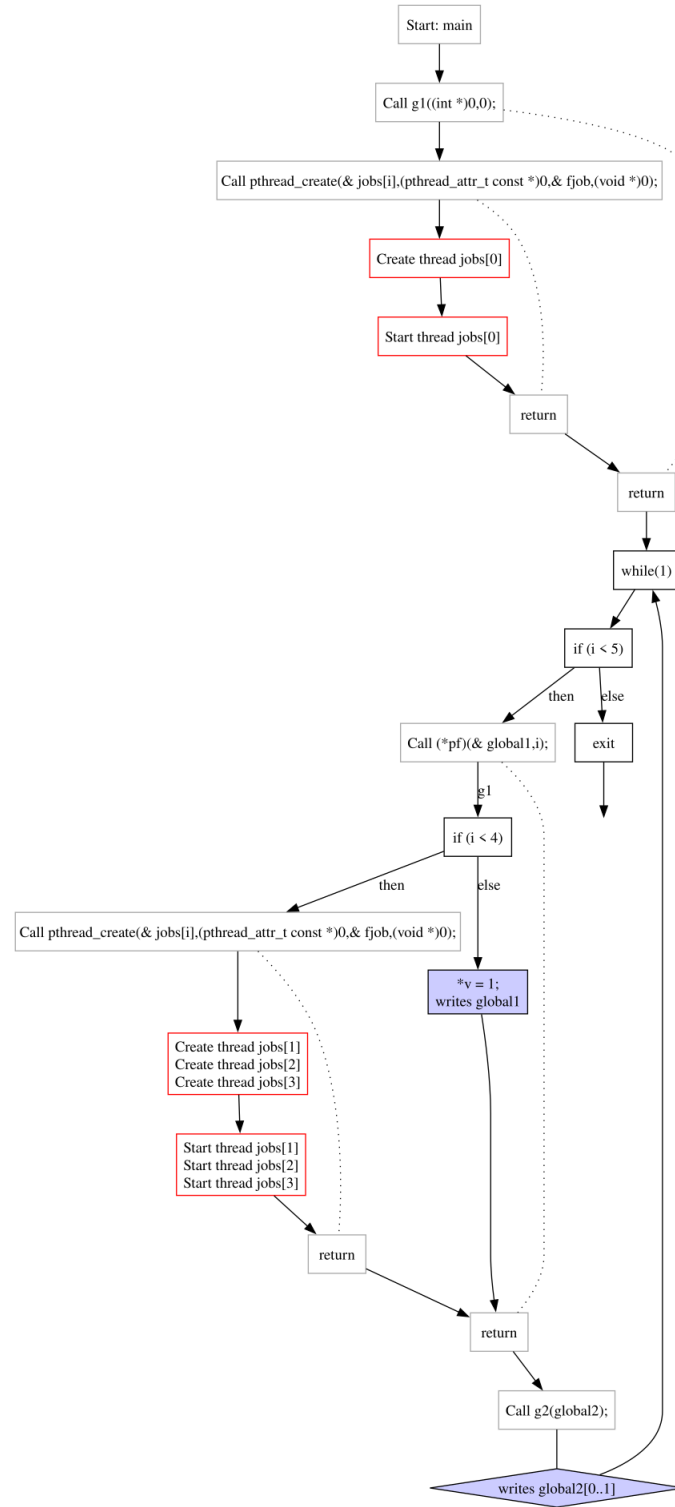


Figure 2.1: Concurrent control-flow graph for the main thread of our example

INSTRUMENTING THE C CONCURRENT PRIMITIVES

To precisely detect calls to concurrent primitives during the symbolic execution of the program, **Mthread** makes the hypothesis that those primitives invoke low-level **Mthread** functions. Hence, the first step in using **Mthread** consists in properly stubbing the thread library of the program. This work has already been done for parts of the **pthread**s library. The functions currently supported by **Mthread** are detailed in the next sections.

In this chapter, and unless stated otherwise, the files we refer to are located in the `src/plugins/eva/share/mthread/` directory of **Frama-C** sources, or alternatively in the `share/mt` of the **Frama-C** installation. After installation, this directory can be found by running the command `echo $(frama-c-config -print-share-path)/mt`. This directory will be abbreviated as `$MTSHARE` in the rest of this document.

3.1 First steps

For a new project, the first step consists in finding within the C sources the `.h` file containing the declarations for the various multithreaded primitives used in the code. In general, those functions include at least

- thread creation (and possibly cancellation);
- mutex locking and release;
- emission and reception on/from a message queue.

Other interesting primitives are those initializing the structures used to refer to the objects above (threads, mutexes, queues), functions using more evolved concurrency primitives (spinlocks,...) *etc...*. A detailed status of which functions are currently handled by **Mthread** is given in [Appendix A](#).

Once the prototypes of the functions above have been found, any potential implementation must be removed from the source, for example by using well-placed `#ifdef 0` lines. This step is however typically not needed, as these functions usually belong to the OS implementation, the source code of which is rarely available.

3.2 Stubbing the header (.h) files

The next steps consists in stubbing the existing concurrency `.h` files. The primary responsibility of this step is to define all the types in the prototypes of the functions in terms of either `__fc_mthread_id` or `__fc_mthread_name`. Both types are defined in the **Mthread** header `mthread.h` as

```
typedef void *__fc_mthread_name;
```

3.2. STUBBING THE HEADER (.h) FILES

```
typedef int __fc_mthread_id;
```

In general, `__fc_mthread_id` is the return type of the initialization functions, and also the type used by functions that *use* an object. During execution, they are simply sequential non-null offsets to an array allocated by `Mthread`, that itself holds the state of the object. The non-null information is important, as some code uses the convention `v == 0` to test whether an object is initialized. Also, we cannot return a pointer, as some concurrent library assume that thread ids are no bigger than the short type; returning short integers (unless the code allocates an inordinate amount of *eg.* mutexes) ensures that our ids can safely be cast to short, or even char.

By contrast, `__fc_mthread_name` is the input type used by initialization functions. It is used as a hint to name the mutex, thread, or queue. It can be either NULL, in which case `Mthread` will use an internal name, a constant string, or the address of a global variable, with possibly an offset (if the variable is an array cell).

As an example, let us show how those two types are used in the prototypes of the primitive `Mthread` functions. The lines below are also an excerpt of `mthread.h`. (The entire file is given in Appendix A.)

```
__fc_mthread_id Framac_C_thread_create(__fc_mthread_name,
                                       void *(*)(void *),
                                       ...);
int Framac_C_thread_cancel(__fc_mthread_id);

__fc_mthread_id Framac_C_mutex_init(__fc_mthread_name);
int Framac_C_mutex_lock(__fc_mthread_id);
```

To conclude this section, let us consider excerpts of the stubbing that has been done for the `pthread` library. The prototypes can be found in the file `mthread_pthread.h`, and are reprinted below.¹

```
#include <mthread.h>

typedef __fc_mthread_id pthread_t;
typedef __fc_mthread_id pthread_attr_t;
typedef __fc_mthread_id pthread_mutex_t;
typedef __fc_mthread_id pthread_mutexattr_t;

#define PTHREAD_MUTEX_INITIALIZER 1

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                  void *(*start_routine)(void *), void *arg);
int pthread_cancel(pthread_t thread);
int pthread_join(pthread_t thread, void **thread_return);
void pthread_exit(void *thread_return) __attribute__((noreturn));
pthread_t pthread_self(void);

int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                      const pthread_mutexattr_t *restrict attr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);

int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
void pthread_testcancel(void);
```

¹ Prototypes for queue-related functions are not actually part of `pthread`s, and are in `mthread_queue.c`.

Except for the typedef declarations, everything can be taken verbatim from a system header for `pthread`s. Thus, since the included file `pthread.h` is a generic header the user should not modify, writing new stubs consists essentially in writing the `pthread.c` file, as explained in the next section.

Both `pthread_t` and `pthread_mutex_t` are defined as type aliases to `__fc_mthread_id`. Indeed, the interface of the `pthread`s library does not lend itself to the separation we use in `Mthread`. Instead, as the next section will show, the initialization primitives use the address of the object they create when naming them.

3.3 Stubbing the source (.c) files

The bulk of the stubbing work consists in implementing the concurrent C primitives in terms of the low-level `Mthread` ones. Stubs are generally very easy to write, as most of the time they consist in:

- disregarding useless arguments (such as initialization options `Mthread` may not handle yet), or swapping some arguments around;
- dereferencing pointers, if a pointer is supplied while `Mthread` needs the value it points to;
- for initialization functions, storing or returning the result of the call to the low-level `Mthread` primitive (which will be the id of the thread, mutex or queue for `Mthread`);
- translating the `Mthread` return codes into those of the OS library.

3.3.1 pthreads library

The stubbing for the `pthread`s library can be found in `mthread_pthread.c`. Its interesting parts are reprinted below.

```
#include "mthread_pthread.h"

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                  void *(*start_routine)(void *), void *arg) {
    int result = Frama_C_thread_create(thread, start_routine, arg);
    if (result > 0) {
        *thread = result;
        Frama_C_thread_start(result);
        return 0;
    } else {
        return 11; /* EAGAIN */
    }
}

int pthread_cancel(pthread_t thread) {
    int result = Frama_C_thread_cancel(thread);
    return (result != -1 ? 0 : 3 /* ESRCH */);
}

pthread_t pthread_self(void) { return Frama_C_thread_id(); }

int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                      const pthread_mutexattr_t *restrict attr) {
    int result = Frama_C_mutex_init(mutex);
    if (result > 0) {
        *mutex = result;
        return 0;
    }
}
```

3.3. STUBBING THE SOURCE (.c) FILES

```
    } else {  
        return 22; /* EINVAL */  
    }  
}  
  
int pthread_mutex_lock(pthread_mutex_t *mutex) {  
    int result = Framac_mutex_lock(*mutex);  
    return (result != -1 ? 0 : 22 /* EINVAL */);  
}  
  
int pthread_mutex_trylock(pthread_mutex_t *mutex) {  
    int result = Framac_mutex_lock(*mutex);  
    return (result != -1 ? 0 : 22 /* EINVAL */);  
}  
  
int pthread_mutex_unlock(pthread_mutex_t *mutex) {  
    int result = Framac_mutex_unlock(*mutex);  
    return (result != -1 ? 0 : 22 /* EINVAL */);  
}
```

Notice the recurring pattern `*obj = obj_init(&obj, ...)` (with proper error-handling) used in both functions `pthread_create` and `pthread_mutex_init`. The address holding the object is used as name hint during the creation. Then the `Mthread` initialization function returns the id of the object, which is stored at the given address. The functions that use this id either dereference their argument if the id is passed as a pointer (`pthread_mutex_lock`) or use it directly otherwise (`pthread_cancel`), depending on their POSIX prototype.

As hinted by the comments, not all functions are properly stubbed. The `*setcancel` functions, which are related to the cancelability of a thread, are not given a body; for now, we implicitly assume that `pthread_cancel` always succeed in stopping a thread. For functions that do not initialize values, there is little differences between not stubbing a function, and giving it a trivial body; the latter approach however silences a few `Framac` warnings. For functions that initialize a structure used later, a stub is however mandatory.

Also, thread return codes are not stored yet, which means that `pthread_join` is not modeled as precisely as possible.

ANALYZING A FULL PROJECT WITHOUT WARNINGS

This chapter explains the warning or error messages emitted by **Mthread** during its analysis. **Mthread**'s own analysis can only be as precise as the one done by **Eva** for each thread. Thus, setting up the latter correctly is important; relevant information can be found in its own manual:

<https://frama-c.com/eva.html>

During the analysis, the first hint that something might have gone awry resides in the warnings sent back to the user. As a general rule of thumb, it is good to eliminate those messages. How to read the *results* of **Mthread** will be explained in Chapter 5.

4.1 The philosophers example

In the remainder of this document, we will use the source code below to exemplify some uses of **Mthread**. It is taken from the file `doc/eva/examples/mthread/philos.c`, and contains a modified version of the classic philosophers problem.

```
1
2 #include "mthread_pthread.h"
3 #include "mthread_queue.h"
4 #define NULL ((void*) 0)
5 #define N 5
6
7
8 int end2 = 0;
9 pthread_mutex_t locks[N];
10 pthread_t jobs[N];
11 msgqueue_t queue;
12
13
14 int random(void);
15
16 void aux (int l, int r, int mess) {
17     pthread_mutex_lock(locks+l);
18     pthread_mutex_lock(locks+r);
19     if (random() && mess != 2) {
20         char buf[2];
21         buf[0]=mess;
22         end2 = 1;
```

4.1. THE PHILOSOPHERS EXAMPLE

```
23     msgsnd(queue, buf, 2);
24 }
25 pthread_mutex_unlock(locks+r);
26 pthread_mutex_unlock(locks+l);
27 }
28
29 void * job( void * k ) {
30     int p = (int) k ;
31     int l = p>0 ? p-1 : N-1 ;
32     int r = p<N-1 ? p+1 : 0 ;
33
34     while(1)
35         aux(l, r, p+1);
36 }
37
38 int main() {
39     int i ;
40     char end[2];
41     end[0]=0;
42
43     for(i=0;i<N;i++)
44         pthread_mutex_init( &locks[i] , NULL);
45
46     queuecreate(&queue, 5);
47
48     for(i=0;i<N;i++)
49         pthread_create( &jobs[i], NULL, job, (void *) i );
50
51     while(!(end[0] && __MTHREAD_SYNC(end2)))
52         msgrcv(queue, 2, end);
53
54     return 0;
55 }
```

This code presents some interesting challenges. First, the ids for the threads and mutexes of the program are stored in two arrays, `jobs` and `locks` respectively. Both arrays are initialized through loops — a challenge for any analyzer. Moreover, the behavior of the various threads is governed by a unique function: the only difference between them lies in the argument they initially receive. Finally, the various threads write in the global variable `end2`, and send a partially initialized message on the queue `queue`. The termination of the main thread is influenced by those two objects.

4.1.1 A first try

Let us start `Mthread` on this program. We need to start `Frama-C` on both `philos.c` and on our stubbed `pthread` library. In order to use our own headers, a proper `-I` directive must be given to our `C` preprocessor. Additionally, the `Mthread` domain for `Eva` should be enabled. Finally, the `Eva` analysis is by default very verbose, and it is a good idea to partially silence it using `-eva-verbose 0`. Thus, a complete invocation of `Mthread` would be

```
% frama-c -cpp-extra-args="-I$MTSHARE" \
-mthread -eva-domains mthread \
$MTSHARE/mthread_pthread.c $MTSHARE/mthread_queue.c philos.c \
-eva-verbose 0
```

(`Frama-C` assumes the main function is called `main`, which is the case here)

4.1. THE PHILOSOPHERS EXAMPLE

While **Mthread**'s output is also rapidly verbose (we will only reproduce snippets below), it is quite apparent that something has gone awry. Many lines start by `phil.c:11[mt] warning:`, where 11 is a line number. The prefix `[mt]` being a short-name for **Mthread**, let us examine a few of those warning lines.

The first one is

```
[mt] phil.c:46: Initializing mutex #2
[mt] phil.c:46: Warning:
    Unable to check that mutex #2 has not been already initialized;
    {0; 1} should be 0
```

By contrast, the previous line was

```
[mt] phil.c:46: Initializing mutex locks[0]
```

Since line 46 is a call to `pthread_mutex_init`, it is clear that the mutex initialization did not succeed in all cases. This is of course due to the presence of the loop, which is executed symbolically, but imprecisely. While the first iteration of the body proceeds as expected (and initializes `locks[0]`), the next ones do not. Instead, during the analysis of the loop, the value `i` ranges over the sets `{0} ... {0, 1, 2, 3, 4}`. While in the first iteration, **Eva** can reduce the value of `locks[i]` to a single value, this is not the case for the following iterations where the expression ranges over the locations `&locks[1 ... 4]`. **Mthread** cannot distinguish one mutex from the other and uses a generic name #2 for the set of locks. Moreover, **Mthread** cannot know which mutex is really being initialized at each iteration and accordingly warns that it is unable to check if the mutex has not already been initialized.

In this particular case, since the initialization is in a loop, increasing the `-eva-slevel` option of the **Eva** analysis should help. Indeed this option controls, among others, the symbolic execution of loops. By default, no `-eva-slevel` is used, and variables modified inside loops become imprecise immediately.

Before increasing this option, let us consider the other warnings. For line 19, we actually get two different warnings depending on which thread is analyzed.

The first warning which occurs during the analysis of thread `jobs[0]` is:

```
[mt] phil.c:19: Warning: Trying to lock a possibly uninitialized mutex.
[mt] phil.c:19: Warning:
    Unable to check that mutex #2 has not already been locked; {0; 1} should be 1
```

This is the same problem as before: mutex #2 is actually a set of mutexes and **Mthread** is not able to know which of the concrete mutexes is being locked.

The second warning occurs during the analysis of thread #3:

```
[mt] phil.c:19: Warning:
    During mutex lock: invalid mutex id. Non-integer value: {0; 1; 2} Ignoring.
```

This is similar to the previous warning in that **Mthread** cannot know which mutex is being locked, but this time this is due to the fact that the threads too have been merged into an abstract thread representing several concrete threads. We can see that by checking the logs for the line 51 that creates the threads:

```
[mt] phil.c:51: New thread: jobs[0], fun job, parent <main>, args {0}
[mt] phil.c:51: Start thread jobs[0]
[mt] phil.c:51: New thread: #3, fun job, parent <main>, args {0; 1}
[mt] phil.c:51: Start thread #3
[mt] phil.c:51: New context for #3, fun job, parent <main>, args {0; 1; 2}
[mt] phil.c:51:
    New context for #3, fun job, parent <main>, args {0; 1; 2; 3; 4}
[mt] phil.c:51: Thread #3, fun job, parent <main>, args {0; 1; 2; 3; 4}
```

Here we see that after creating a thread for `jobs[0]`, **Eva** starts merging the states for several threads and **Mthread** creates a second thread generically called #3. Instead of creating several new threads, the context of this abstract thread is updated to support more arguments until the set `{0; 1; 2; 3; 4}` is used. Since this argument is used to select the mutexes to lock and unlock in function `aux`, **Mthread** cannot know which mutex is being used.

4.1.2 Unrolling loops

Let us increase the `slevel` to separate the states of the loops in the `Eva` analysis. Here, it suffices to execute loops precisely 5 times. Thus, we add option `-eva-slevel 5` to the command-line used above. In this case, it is sufficient to make all warnings disappear.

Alternatively, it is also possible to *syntactically* unroll loops. Although it is seldom useful for the `Eva` analysis, it may be for `Mthread` itself. Let us consider the file `doc/eva/examples/mthread/init.c` of `Mthread`, which is reproduced below.

```

1  /* This example tests the various way a structure can be named:
2     with a pointer, with a string, without any indication */
3  #include "mthread_pthread.h"
4  #define NULL ((void*)0)
5  #define N 3
6
7  int locks[N];
8  char (*names[3*N]) =
9      { "mu1", "mu2", "mu3", "mu4", "mu5", "mu6", "mu7", "mu8", "mu9" };
10
11
12 int mutex_init(void* mname) {
13     return Framac_mutex_init(mname);
14 }
15
16 void main() {
17     int i ;
18
19     for(i=0;i<N;i++)
20         mutex_init(&locks[i]);
21
22     for(i=0;i<N;i++)
23         mutex_init(names[i]);
24
25     //@ loop unfold 2*N;
26     for(i=0;i<2*N;i++)
27         if (i >= N)
28             mutex_init(names[i]);
29
30     for(i=0;i<3*N;i++)
31         if (i >= 2*N)
32             mutex_init(names[i]);
33
34     // Warning: the same mutex is repeatedly created
35     for(i=0;i<N;i++)
36         mutex_init(NULL);
37 }

```

This example try to initialize 12 mutexes, using three different mechanisms. In the first loop, the mutexes are named using a location in an array. In the second loop, third and fourth loops, they are named using constant strings in increasingly large loops. The result of the analysis of the main thread by `Mthread` are given below (with a proper `-eva-slevel 3`):

```

[mt] *** Computing value analysis for main thread
[mt] New thread: <main>, fun main
[mt] init.c:20: Initializing mutex locks[0]

```

4.2. OTHER MTHREAD WARNINGS

```
[mt] init.c:20: Initializing mutex locks[1]
[mt] init.c:20: Initializing mutex locks[2]
[mt] init.c:23: Initializing mutex mu1
[mt] init.c:23: Initializing mutex mu2
[mt] init.c:23: Initializing mutex mu3
[mt] init.c:28: Initializing mutex mu4
[mt] init.c:28: Initializing mutex mu5
[mt] init.c:28: Initializing mutex mu6
[mt] init.c:32: Initializing mutex mu7
[mt] init.c:32: Initializing mutex #11
[mt] init.c:32: Warning:
  Unable to check that mutex #11 has not been already initialized;
  {0; 1} should be 0
[mt] *** First value analysis for main thread done.
```

As can be seen, the first 9 initializations succeed without problem. The mutexes created in the first loop are named after the array passed as hint (and the index of the mutex in the array), while the exact names contained in names are used for the mutexes 4 to 9 in the second and third loops.

Problems however arise in the last loop, although it is very similar to the third one. **Mthread** initializes the tenth mutex on the first iteration, but complains in the second that it cannot determine if the mutex has already been initialized. Indeed the number of iterations of the loop is greater than the value given to `-eva-slevel` and **Mthread** is not able to separate the different mutexes. A possible solution (other than increasing the `-eva-slevel` value) is to syntactically unroll the loop, as was done for the third one. The instruction `//@ loop unfold N;` instructs **Frama-C** to unroll a loop `N` times. Afterward, **Mthread** sees some mutex initializations at different statements, and accepts them. Internally however, those statements point to the same initial line number, hence the messages for the third loop in the log.

4.2 Other Mthread warnings

In this section, we give a short survey of some the warnings emitted by **Mthread**. In a log output, those warnings contain the string `[mt]`. Most of the time, the warnings are self-explanatory, and they sometimes contain their own solution. Roughly speaking, they can be partitioned in the categories below.

Erroneous or imprecise arguments. **Mthread** systematically sanitizes the arguments it receives from the `Frama_C_*` functions defined in `mthread.h`, and ignores the entire call (with a warning) when it cannot give a sense to them. We have already given an example in §4.1.1 with an imprecise name for a mutex initialization. Nearly identical warnings are emitted with imprecise names or ids, for threads, mutexes or queues.

Other similar errors can include passing a function without a body to the thread creation function `Frama_C_thread_create`, or too few arguments. The corresponding messages are given below.¹

```
phil.c:51:[mt] warning: During thread creation: invalid thread function.
               Missing definition for function 'job'. Ignoring.
```

```
phil.c:51:[mt] user error: When creating thread &jobs[0] from function
               job: too few arguments, 1 expected but 0 given. Ignoring.
```

Multiple creation of a unique thread. **Mthread** is quite tolerant when it encounters code that would initialize again a mutex or a queue potentially already initialized:

```
phil.c:46:[mt] warning: Mutex &locks might be already initialized
```

¹ The messages are obtained by simple modifications of our examples and stubs, not shown in this document.

4.2. OTHER MTHREAD WARNINGS

We cannot be as lenient for threads, however, to preserve the correctness of our analyzes. Thus, when we detect a thread that seems to be started twice, we immediately fail. Of course, it remains possible to launch two threads with exactly the same arguments, but the program must use two different names.

```
philo.c:51:[mt] Thread &jobs[0], fun job, parent _main_, args {0; }
philo.c:51:[mt] user error: Thread &jobs[0] has already been created
                        previously in the current thread.
```

Read or write of the entire memory. If the *Eva* analysis dereferences a very imprecise pointer, it can access the whole memory. This completely invalidates the assumptions made by *Mthread* when it searches for shared memory, and can make it very imprecise. We therefore entirely ignore the access. Since such an imprecise pointer almost always comes from an erroneous stubbing, or a very buggy original code, this is not a limitation in practice.

The directory `tests` contains an example designed to test this case, called `read_all.c`. The *Eva* analysis prints a warning when the faulty pointer `p` is being dereferenced (line 4). The *Mthread* warning is on line 7.

```
1 read_all.c:26:[mt] New thread: &jobs, fun f, parent <main>,
2   args [0..4294967295]
3 read_all.c:26:[mt] Start thread &jobs
4 read_all.c:28:[mt] user error: read of the whole memory. Ignoring to allow
5   Mthread to continue, but the analysis will not be correct.
6 read_all.c:28:[eva] warning: Completely invalid destination for assigns
7   clause *p. Ignoring.
```

Buffer overflow in message sending or receiving. *Mthread* send and receive functions take as input either a source buffer, or a destination one, as well as its size. Of course, the *Eva* analysis must be wary of buffer overflow. Let us successively change the declaration of the buffers `buf` and end of `philo.c` to `char[1]` (lines 31 and 42).

Small buffer during emission

```
../share/mthread_queue.c:10:[kernel] warning: out of bounds read.
      assert \valid(mess+(0..size-1));
philo.c:25:[mt] Sending message on &queue, content [0..1] ${in$ {}}
```

The indicated line is inside the function `msgsnd`:

```
int result = Frama_C_queue_send(msgqid, mess, size);
```

Here we have `mess=buf`, `size=2`, and the program is defined only if `buf[0..(2-1)]` is a valid array slice. This is indeed false in the modified program, as `buf` has size 1. In this case, the *Eva* analysis is sure that the range is always invalid, as there is no approximation on either `buf` or `size`, and the read fails. This can be verified with the empty message content in the second line of the log.

Small buffer during reception

```
../share/mthread_queue.c:16:[kernel] warning: out of bounds write.
      assert \valid(mess+(0..size-1));
../share/mthread_queue.c:16:[kernel] warning: all target addresses were
      invalid. This path is assumed to be dead.
philo.c:54:[mt] warning: Found message of length 2, which is too long for
```

4.2. OTHER MTHREAD WARNINGS

```
buffer 'mess'. Execution will continue without those messages.  
(Ignore "This path is assumed to be dead message if any".)  
philo.c:54:[mt] Receiving message on &queue, max size 2, stored in &end.  
No valid value to receive.
```

Again, the **Eva** analysis detects that we are accessing past the end of an array. The warning “This path is assumed to be dead” is actually not really relevant here, and should be ignored. Next, **Mthread** adds a more precise warning about which buffer is too small, and warns that messages of length 2 are too long. This means that any message of at least that size will be ignored by `Frama_C_queue_receive`. Since all messages are of size 2, there is nothing valid to receive (hence the last line of the log), and **Mthread** instructs the **Eva** analysis to stop when evaluating the call to `Frama_C_queue_receive`.

Too many objects. By default, **Mthread** allows the creation of 32 threads, mutexes or queues, with different counters for each kind of object. This value is hard-coded in `mthread.h`, in order to have valid C. If **Mthread** detects that a program wants to allocate more than this number of objects, it issues a warning.

```
philo.c:51:[mt] warning: During thread creation. Too many thread ids,  
unable to register another one. Try to increase MTHREAD_NUMBER_IDS  
above 32 in the preprocessing directive. Ignoring.
```

As hinted by the message, the number of possible distinct **Mthread** objects is defined by the C macro `MTHREAD_NUMBER_IDS`. Thus, it suffices to increase its value in the preprocessing directive `-cpp-command`, *eg.* by adding `-DMTHREAD_NUMBER_IDS=40` for **Gcc** or **cpp**.

Unrecognized id. The ids returned by **Mthread** for threads, mutexes and queues are C ints, unless they are cast to another type by the program itself. If the code does strange things with those ints, *eg.* incrementing them, it can build precise but incorrect ids. **Mthread** will then fail with a message similar to the one below.

```
philo.c:29:[mt] warning: During mutex lock. Id 13 for mutexes does not  
exists (incrementation inside program?). Ignoring.
```

Uninitialized concurrency structures. Primitives receiving an id as argument can be passed the value 0. This typically corresponds to non-initialized mutexes, queues *etc.*... Either this is a mistake in the code (the programmer forgot the initialization), or the initialization will be done later, by another thread, and the warning should disappear in later iterations of the analysis.

```
philo.c:38:[mt] warning: Trying to unlock uninitialized mutex. Ignoring
```

READING MTHREAD RESULTS

This chapter explains how to interpret the results output by `Mthread`, on the philosophers example.

5.1 Reading the results of the philosophers examples

Running `Mthread` on `philos.c` goes smoothly once a proper `slevel` (of at least 5) is used. No warning is emitted during the analysis. `Mthread` reports it stops after 4 iterations, having reached the fixpoint. However, not all threads are executed at each iteration. For example, `Mthread` detects it would learn nothing by analyzing the thread `main` during its second step, and thus skips this analysis. If we read more finely the output, for example by setting `-mt-verbose 2`, the iteration structure looks like this:

Initial run of the main thread This analysis detects the five secondary threads. Receiving a message on `&queue` fails. No potential shared zone is detected — quite logically, as only one thread was running.

First iteration The five secondary threads are executed. Messages sent on `&queue` are memorized for an eventual use in another thread.

```
[mt] philos.c:25:
    Sending message on queue, content [0] $\in$ {1}
                                     [1] $\in$ UNINITIALIZED
```

Second iteration The main thread is recomputed, because `Mthread` detects that some messages can be received on `&queue`.

```
[mt] *** Computing thread <main>, iteration 2 (new message received)
```

During this iteration, the call to `msgrcv` succeeds, and the value `end[0]` becomes possibly non-null. As a result, a new shared memory zone is detected, the variable `end2`.

```
[mt] Concurrent imprecise accesses have changed: before
      \nothing
      vs.
      end2
```

Third iteration All threads are recomputed because we want to monitor the accesses to the potential shared variable `end2`:

```
[mt] *** Computing thread jobs[0], iteration 3
      (potential shared vars changed, interferences changed)
```

At the end of the iteration, `end2` is detected as being a (really) shared zone, not just a potential one:

5.1. READING THE RESULTS OF THE PHILOSOPHERS EXAMPLES

```
[mt] Shared memory: end2
[mt] Concurrent precise var accesses have changed: before
    \nothing
vs.
    end2
```

Mthread also detects that `end2` is not protected in a coherent way, *ie.* that there might be a race condition on it.

```
[mt] Possible read/write data races:
end2:
  read by <main> at philo.c:53, unprotected
  write by jobs[0] at philo.c:24, protected by locks[1] locks[4]
  write by jobs[2] at philo.c:24, protected by locks[1] locks[3]
  write by jobs[3] at philo.c:24, protected by locks[2] locks[4]
  write by jobs[4] at philo.c:24, protected by locks[0] locks[3]
[mt] Mutexes for concurrent accesses:
end2 write protected by (?)locks[0] (?)locks[1] (?)locks[2] (?)locks[3]
      (?)locks[4], read unprotected
```

Mthread does not report any new potential shared variable however, which is coherent with the program.

Fourth iteration During this iteration, the thread `main` is recomputed. Indeed, new possible values for `end2` (coming from the other threads), have been found in iteration 3.

```
[mt] *** Computing thread <main>, iteration 4 (shared vars values changed)
```

During this iteration, the return statement of the `main` function becomes reachable.

As this state of the analysis, there is no reason to recompute any of the threads, and **Mthread** detects that a fixpoint is reached.

```
[mt] ***** Analysis performed, 4 iterations
```

Not all the logs given above are available with the default verbosity level of 1. Indeed, they are not important to understand the *results* of the analysis, only the way it proceeded.

Let us point out a few more information. For example, the information on the mutexes protecting the accesses to `end2` are two-fold. First, we have an exhaustive account, with all accesses by each thread; each access is listed together with the mutex contexts at those points of the analyzes. In this example, the information is as precise as possible. Second, we have a summary, that aggregates the exhaustive listing.

```
[mt] Mutexes for concurrent accesses:
end2 write protected by (?)locks[0] (?)locks[1] (?)locks[2] (?)locks[3]
      (?)locks[4], read unprotected
```

This shows that `end2` is not protected at all when it is read. Conversely, it is protected by various mutexes when it is written, but never in a consistent way: there is always a `(?)` in front of the mutex name, indicating that in at least one case, the mutex was not locked. This indicates possible race conditions both when reading and writing `end2`, which is indeed the case in the program.

Finally, let us discuss the values of the messages sent and received on `&queue`. We reprint some relevant messages below.

```
[mt] philo.c:25:
    Sending message on queue, content [0] $\in$ {1}
                                     [1] $\in$ UNINITIALIZED
```

```
[mt] philo.c:54:
  Receiving message on queue, max size 2, stored in &end. Possible values:
  From thread jobs[0]: [0] $\in$ {1}
                        [1] $\in$ UNINITIALIZED
  From thread jobs[2]: [0] $\in$ {3}
                        [1] $\in$ UNINITIALIZED
  From thread jobs[3]: [0] $\in$ {4}
                        [1] $\in$ UNINITIALIZED
  From thread jobs[4]: [0] $\in$ {5}
                        [1] $\in$ UNINITIALIZED
```

Mthread is quite accommodating about the content of the message, and tolerates the fact that a part of the source buffer is uninitialized. Inspecting the value of `end`¹ after line 54 of `philo.c` reveals that the possible values are

```
end[0] $\in$ {1; 3; 4; 5}
[1] $\in$ UNINITIALIZED
```

This is also the most precise approximation possible.

5.2 HTML

Mthread HTML output, triggered by option `-mt-extract html`, produces a summary of the concurrent program, as well as control-flow graphs of each thread as analyzed by **Mthread**. Let us start by the first representation extracted from **Mthread**'s internal control-flow-graph model: a set of HTML pages. This allows easy browsing through the various information computed by **Mthread**.

For our simple dining philosophers' example, these pages can be produced in the directory `html_summary` by typing :

```
% frama-c -mthread -eva-domains mthread -mt-extract html \
  -cpp-extra-args="-I$MTSHARE" \
  $MTSHARE/mthread_pthread.c $MTSHARE/mthread_queue.c \
  -eva-slevel 256 philo.c
```

An HTML summary of the code (Figure 5.1) is displayed at `html_summary/index.html`, providing information about thread creations, lock and unlock directives as well as message queue uses. There also are links to the various threads encountered in the program. Clicking on one of those links leads to a summary concerning the given thread. This thread-focused summary (Figure 5.2) shows the concurrent control-flow graph of the thread (§2.2).

The precision and details shown on this graph can be controlled by **Mthread**'s options detailed in 2.2 and Appendix 6. Links to all the other threads are provided but one important feature here is that the control-flow graph is clickable. A click on the `Call aux` node yields the source code behind this node (Figure 5.3). Most of the expressions in the source code page are themselves clickable, for example to navigate from function to function.

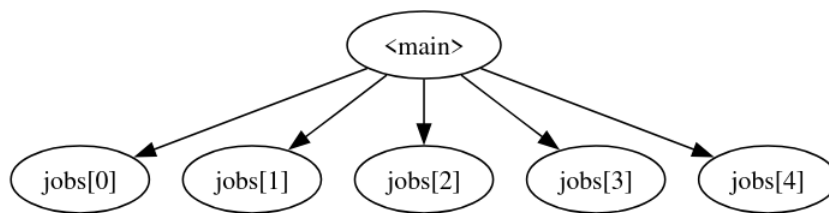
¹ For example using **Frama-C**'s GUI

Summary

This program has 6 thread(s)

- [<main>](#)
- [jobs\[0\]](#)
- [jobs\[1\]](#)
- [jobs\[2\]](#)
- [jobs\[3\]](#)
- [jobs\[4\]](#)

Thread creation graph



[Direct link](#)

Lock operations

uses lock ← ↓	<main>	jobs[0]	jobs[1]	jobs[2]	jobs[3]	jobs[4]
locks[0]			PV			PV
locks[1]		PV		PV		
locks[2]			PV		PV	
locks[3]				PV		PV
locks[4]		PV			PV	

P = lock taken, V = lock released

Queue operations

uses lock ← ↓	<main>	jobs[0]	jobs[1]	jobs[2]	jobs[3]	jobs[4]
queue	CR	S		S	S	S

R = queue read, S = queue written, C = queue created

Figure 5.1: HTML summary of dining philosophers

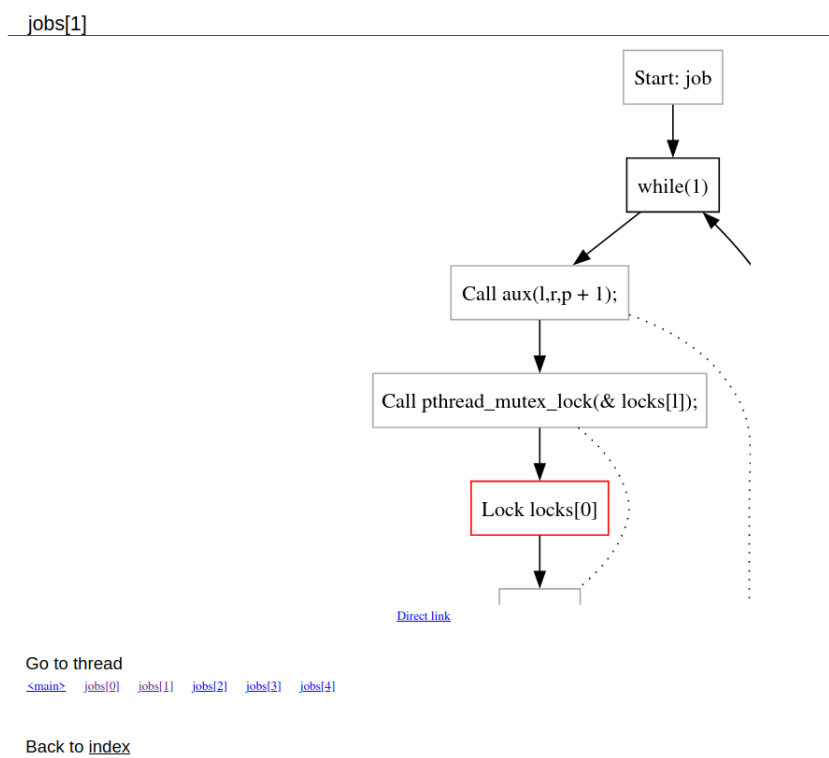


Figure 5.2: Excerpt from the first philosopher's control-flow graph

[jobs\[0\]](#)

```
void *__retres;
int tmp;
int tmp_0;
int p = (int)k;
if (p > 0) tmp = p - 1; else tmp = 5 - 1;
int l = tmp;
if (p < 5 - 1) tmp_0 = p + 1; else tmp_0 = 0;
int r = tmp_0;
while (1) aux(l,r,p + 1);
return __retres;
}

int main(void)
{
    int __retres;
    int i;
    char end[2];
    end[0] = (char)0;
    i = 0;
    while (1) {
        if (i < 5) ; else break;
        pthread_mutex_init(& locks[i],(pthread_mutexattr_t const *)0);
        i += 1;
    }
    queuecreate((__fc_mthread_name) (& queue),5);
    i = 0;
    while (1) {
        if (i < 5) ; else break;
        pthread_create(& jobs[i],(pthread_attr_t const *)0,& job,(void *)i);
        i += 1;
    }
    while (1) {
        if (end[0]) {
            Frama_C_mthread_sync();
            if (end[2]) break; else ;
        }
        else ;
        magrcv(queue,2,&(end[0]));
    }
}
```

[Direct link](#)

Go to thread

[<main>](#) [jobs\[0\]](#) [jobs\[1\]](#) [jobs\[2\]](#) [jobs\[3\]](#) [jobs\[4\]](#)

Back to [index](#)

Figure 5.3: Excerpt from the philosophers' source code

COMMAND-LINE OPTIONS

This section describes the list of options available to finely tune the behavior of **Mthread**. They can also be manually found using `frama-c -mt-help`. Some experimental options are intentionally left undocumented.

Basics. As a reminder, the generic options for **Mthread** are the following:

- `-mthread` This enables the **Mthread** plug-in. This option is mandatory for any use of **Mthread** and launches the **Mthread** analysis.
- `-eva-domains mthread` This enables the **Mthread** domain for **Eva**. This option is strongly recommended to compute more precise interferences.
- `-mt-verbose n` Change the verbosity of **Mthread**. Default is 1. Any value strictly above 1 will show the internal state of the analysis at the end of each iteration.
- `-mt-help` Display a short summary of all the **Mthread** options available.
(The options for **Frama-C** in general can be obtained through `frama-c -kernel-help`, while those for the **Eva** analysis are invoked by `frama-c -eva-help`.)

External outputs. The **Mthread** results printed as **HTML** for further study.

- `-mt-extract html` Extracts a partial version of the results found by **Mthread** as **HTML**. All results can be browsed¹ starting from the file `./html_summary/index.html`.

Control-flow graph options. The options below control how the concurrent control-flow graphs are displayed and simplified.

- `-mt-return-edges` Link nodes for function calls to their corresponding `return` nodes. This makes it easier to see nested calls of big functions. (Set by default)
- `-mt-non-shared-accesses` Do not remove nodes corresponding to accesses to false shared accesses (§2.3). Not set by default; if the option is set, the accesses are shown in white in the control-flow graph.
- `-mt-non-concurrent-accesses` Do not remove nodes corresponding to accesses to shared accesses that occur in a non-concurrent context (§2.3). Set by default, those accesses are shown in green in the control-flow graph.
- `-mt-full-cfg` Do not simplify the bodies of functions that contain multithreaded events. All the statements of those functions will be reflected in the control-flow graphs, which can result in very big graphs: use this option with caution. Calls to functions that do not contain multithreaded events are however never inlined in the control-flow graphs. Not set by default.

¹ A navigator with support for **SVG** files is required to display the control-flow graphs.

Debug options. Those debug options are not intended for general use, but can sometimes be useful to diagnose a strange behavior of **Mthread**. Other debug options are unintentionally not described.

`-mt-stop-after <i>` Instructs **Mthread** to only perform at most `i` iterations of the analysis. If the analysis has not converged by then, it is stopped, and the remaining steps to perform are shown on the log.

MTHREAD FUNCTIONS FOR STUBBING

A

This appendix details the concurrent functions `Mthread` is able to detect and handle. Their prototypes can be found in the file `$MTSHARE\mthread.h`.

Thread-related primitives

- Thread creation, through function `Frama_C_thread_create`
- Thread start, through function `Frama_C_thread_start`
- Thread immediate exit, through function `Frama_C_thread_exit`
- Current thread id, through function `Frama_C_thread_id`
- Thread suspension, through function `Frama_C_thread_suspend`
- Thread canceling, through function `Frama_C_thread_cancel`
(*This functions currently cancels the thread regardless of any potential cancelability state notion, such as the one available in `pthread`s.*)

Mutex-related primitives

- Mutex initializing, through function `Frama_C_mutex_init`
- Mutex locking, through function `Frama_C_mutex_lock`
- Mutex release, through function `Frama_C_mutex_unlock`

Queue-related primitives

- Queue initializing, through function `Frama_C_queue_init`
- Message sending, through function `Frama_C_queue_send`
- Message reception, through function `Frama_C_queue_receive`

Miscellaneous functions

- Logging, through function `Frama_C_mthread_show`
This function takes as first argument a constant string will be used as a message, and a number of C values that will be printed after the message. It can be used to show in the control-flow graph any information relevant to the analysis, and does not modify the memory state at all.
- Forcing synchronization of unprotected shared values, through the use of the function `Frama_C_mthread_sync`.

More involved concurrency primitives, such as spinlocks *etc.*... are not currently supported. They may be added to `Mthread` later.

BIBLIOGRAPHY

- [Fer09] Pietro Ferrara. *Static analysis via abstract interpretation of multithreaded programs*. PhD thesis, Ecole Polytechnique of Paris (France) and University "Ca' Foscari" of Venice (Italy), May 2009.
- [HFP06] Michael Hicks, Jeffrey S. Foster, and Polyvios Prattikakis. Lock inference for atomic sections. In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. ACM, June 2006.
- [Min12] A. Miné. Static analysis of run-time errors in embedded real-time parallel C programs. *Logical Methods in Computer Science (LMCS)*, 8(26):1–63, Mar. 2012. <http://www.di.ens.fr/~mine/publi/article-mine-LMCS12.pdf>.
- [VV07] Vesal Vojdani and Varmo Vene. Goblint: path-sensitive data race analysis. In *SPLST*, pages 171–187, 2007.