ISO/IEC DTR 13211–5:2007
Prolog multi-threading support

Editor: Paulo Moura
pmoura@di.ubi.pt

March 18, 2008

Introduction

This technical recommendation (TR) is an optional part of the International Standard for Prolog, ISO/IEC 13211. Prolog systems wishing to implement multi-threading supporting predicates should do so in compliance with this TR.

Multi-thread predicates are based on the semantics of POSIX threads. They have been implemented in some Prolog systems. As such, they are deemed a worthy extension to the ISO/IEC 13211 Prolog standard.

This TR was designed to follow the predicate naming conventions found on other Prolog standard documents. In addition, the specified set of built-in predicates allow for extension by Prolog vendors in a controlled way aimed to prevent future sources of incompatibilities.

This draft may contain in several places informative text, type-set in *italics*. Such informative text is used for editorial comments deemed useful during the development of this draft and may not be included in the final version.

Previous editors and draft documents

- Paulo Nunes (April 12, 2006 — July 31, 2006).

Draft document comments

- None.

Contributors

- Jan Wielemaker (Netherlands)
- Manuel Carro (Spain)
- Paulo Moura (Portugal)

- Paulo Nunes (Portugal)
- Peter Robinson (Australia)
- Rui Marques (Portugal)
- Terrance Swift (USA)
- Ulrich Neumerkel (Austria)
- Victor Santos Costa (Portugal)

1 Scope

This TR is designed to promote the portability of multi-threaded Prolog applications. As such, this TR specifies:

a) A set of built-in predicates for thread management, monitoring, and communication.

b) A set of built-in predicates for thread synchronization using mutexes.

c) A set of built-in predicates for message queue management, monitoring, and communication.

This TR specifies both mandatory and optional built-in predicates. All the optional built-in predicates are related to statistical information that, although sometimes useful, imply a runtime overhead that a conforming implementation may prefer to avoid.

2 Language concepts and semantics

2.1 Types

2.1.1 Kibibyte

A `kibibyte` is 1024 bytes. The unit `kibibyte` (`KiB`) is specified by the IEC 60027-2 standard.

2.1.2 Thread and message queue aliases

The predicates specified in this TR allow the definition of thread aliases and message queue aliases, which share the same namespace.

NOTE — Each thread is associated with its own message queue, accessed using the thread identifier. Thus, in order to avoid ambiguities, the alias of a new thread may not be in use as a queue alias. Likewise, the alias of a new queue may not be in use as a thread alias.

2.1.3 Thread status

The thread status shall include:

`exception(Exception)` — The thread goal has been terminated due to an uncaught exception, represented by the term `Exception`.

`exited(ExitTerm)` — The thread goal has been terminated using the built-in predicate `thread_exit/1` (3.2.7) with `ExitTerm` as its argument.

`false` — The thread goal has been completed and failed.

`running` — The thread is running. This is the initial status of a thread (note that threads suspended or waiting are also considered as running).

`true` — The thread goal has been completed and succeeded.

2.2 Thread-creation options

Thread-creation options may be used with the `thread_create/3` (3.2.1) built-in predicate in order to bypass default creation options.

2.2.1 Mandatory options

The supported thread-creation options shall include:

`alias(Alias)` — Specifies that the atom `Alias` is to be an alias for the thread.

`detached(Bool)` — When `Bool` is `false`, the thread can be waited for using the built-in predicate `thread_join/2` (3.2.5). When `Bool` is `true`, is expected that the system will reclaim all associated resources automatically after the thread finishes.

2.2.2 Optional data area options

The thread-creation data area options supported may include:

`local(KiB)` — Sets the size limit to the local stack of the thread. When omitted, a default size value is used.

`global(KiB)` — Sets the size limit to the global stack of the thread. When omitted, a default size value is used.

`trail(KiB)` — Sets the size limit to the trail stack of the thread. When omitted, a default size value is used.

`stack(KiB)` — Sets the size limit to the system stack of the thread. When omitted, a default size value is used.

NOTE — The default size values are implementation dependent (3.2.2).

2.2.3 Optional scheduling options

The thread-creation scheduling options supported may include:

`scheduling_scope(Value)` — The possible values of scheduling scope `Value` shall include:

`system` — Kernel threads.

`process` — User Threads.

`policy(Value)` — The possible values of policy `Value` shall include:

`round_robin` — Round Robin.

`fifo` — First in first out.

`other` — Other.

`priority(Value)` — `Value` is an integer value between `0` (minimum priority) and `31` (maximum priority). Priority is settable at thread creation time or dynamically.

NOTE — The interval `[0, 31]` is a virtual priority range that must be mapped to real priority values, which are implementation and operating-system dependent. This virtual range follows the POSIX 1b recommendation and the current practice on Microsoft Windows operating-systems.

`yield(Value)` — The possible values of yield `Value` shall include:

`round_robin` — Round Robin.

`fifo` — First in first out.

`other` — Other.

NOTE — `round_robin` and `fifo` only yields to a thread of the same priority.

`scheduling_inheritance(Value)` — The possible values of scheduling inheritance `Value` shall include:

`inherit` — Inherits scheduling attributes from the parent thread.

`explicit` — Uses thread specific scheduling attributes.

2.3 Thread-creation options list

A thread-creation options list is a list of thread-creation options which may be used with the built-in predicate `thread_create/3` (3.2.1) in order to bypass default creation options.

2.4 Thread properties

Thread properties include the thread creation options specified in section 2.2, plus the following one:

`status(Status)` — `Status` is the state of the thread (2.1.3).

2.5 Mutex-creation options

Mutex-creation options may be used with the built-in predicate `mutex_create/2` (3.4.1) in order to bypass default creation options. The supported mutex-creation options shall include:

`alias(Alias)` — Specifies that the atom `Alias` is to be an alias for the mutex.

2.6 Mutex-creation options list

A mutex-creation options list is a list of mutex-creation options (2.5) which may be used with the built-in predicate `mutex_create/2` (3.4.1) in order to bypass default creation options.

2.7 Mutex properties

Mutex properties include the mutex creation options specified in section 2.5, plus the following optional one:

`status(Status)` — the status of the mutex is `Status`.

The possible mutex status are:

`locked(ThreadOrAlias, Count)` — `ThreadOrAlias` is the identifier of, or an alias to, the holding thread, and `Count` is the recursive count of the mutex.

`unlocked` — the mutex is unlocked.

NOTE — Implementations shall return thread aliases, when defined, instead of thread identifiers.

2.8 Queue-creation options

Queue-creation options may be used with the `message_queue_create/2` built-in predicate (3.5.1) in order to bypass default creation options.

`alias(Alias)` — Specifies that the atom `Alias` is to be an alias for the queue.

`max_size(MaxSize)` — Sets the size limit to the queue. When omitted, the maximum value of items in a message queue is expected to be resource bound.

2.9 Queue-creation options list

A queue-creation options list is a list of queue-creation options (2.8) that may be used with the built-in predicate `message_queue_create/2` (section 3.5.1) in order to bypass default creation options.

2.10 Queue properties

Queue properties include the queue creation options specified in section 2.8, plus the following one:

`size(Size)` — `Size` is the number of elements on the message queue.

NOTE — Due to concurrent access, the value of this property may be outdated before it is returned (3.5.3). Nevertheless, this property may be used for debugging purposes.

2.11 Database

The Prolog database is shared with all threads.

2.11.1 Dynamic predicates

It is possible to use dynamic predicates for thread communication. Compliant implementations shall ensure that access to a dynamic predicate is properly synchronized in order to guarantee both the logical update semantics of single-threaded Prolog when two threads call the same dynamic predicate and the consistency of the dynamic predicate when two threads assert or retract clauses for the same dynamic predicate.

2.12 Executing a threaded Prolog goal

Executing a Prolog goal in a thread works by first making a copy of the goal term and then executing the copy in the thread Prolog engine. This implies that further instantiation of the goal term in one thread does not have consequences for the other thread; i.e. any variable bindings are lost when a goal is proved using another thread.

2.13 Flags

The following flags are added to those of section 7.11 of ISO/IEC 13211-1.

2.13.1 Flag: max_threads

Possible values: `undefined`, *positive integer value*
Default value: implementation defined
Changeable: No

Description: If the value of this flag is `undefined`, the number of threads that can coexist at any given time on a Prolog process is not defined.

If the value of this flag is a positive integer value, the number of Prolog threads that can coexist at any given time is limited to this value. The actual number of threads that can be created by the user may be lower as the Prolog runtime itself may create threads for its own internal bookkeeping.

2.13.2 Flag: hardware_threads

Possible values: `undefined`, *positive integer value*
Default value: implementation defined
Changeable: Yes

Description: If the default value of this flag is `undefined`, the maximum number of hardware supported simultaneous threads is not defined.

If the default value of this flag is a positive integer value, the maximum number of hardware supported simultaneous threads is given by this value. The flag value can be modified by the programmer in order to simulate running an application in hardware with a greater or a lower number of supported simultaneous threads.

2.14 Errors

The following errors are defined in addition to those defined in section 7.12 of ISO/IEC 13211-1.

2.14.1 Error classification

The following types are added to the classification of 7 12.2 of ISO/IEC 13211-1.

a) The list of valid domains is extended by the addition of `mutex_or_alias`, `mutex_option`, `thread_or_alias`, `thread_option`, `thread_property`, `queue_or_alias`, `queue_option`, and `queue_property` (see 7 12.2 c of ISO/IEC 13211-1).

b) The list of object types is extended by the addition of `mutex`, `queue`, and `thread` (see 7 12.2 d of ISO/IEC 13211-1).

c) The list of operations is extended by the addition of `detach`, `join`, `unlock`, `cancel`, and `destroy`; the list of permission types is extended by `mutex`, `queue`, `thread`, and `thread_option` (see 7 12.2 e of ISO/IEC 13211-1).

d) The list of valid flags is extended by the addition of `max_threads` (see 7 12.2 f of ISO/IEC 13211-1).

2.15 Main Prolog thread

When starting a multi-threading enabled Prolog process, there shall be a main thread, referenced by the alias `main`. This thread shall be automatically created and cannot be canceled during the lifetime of the Prolog process.

2.16 Signaling threads

This TR specifies a predicate, `thread_signal/1`, to make a thread execute some goal as an interrupt (3.2.8). Conforming implementations shall ensure that these interrupts are only checked at safe points in their virtual machines. These safe points shall include thread sleep, thread suspended waiting for a message (from a queue), and stream read operations. Signaling should always be handled with care as the receiving thread may hold a mutex.

Signaling may be used to start debugging a thread or to cancel no-longer-needed threads with `throw/1`, where the receiving thread should be designed carefully to handle exceptions at any point.

2.17 Thread message queues

Each thread owns a private message queue that can be accessed using the thread identifier or the thread alias. For a non-detached thread, its private message queue is only destroyed when thread is joined. For a detached thread, its private message queue is destroyed when the thread exits.

3 Built-in predicates

3.1 The format of built-in predicate definitions

These subclauses define the format of the definitions of built-in predicates.

3.1.1 Description

The description of the built-in predicate assumes that no error condition is satisfied, and is in two parts: (1) the logical condition for the built-in predicate to be true, and (2) a procedural description of what happens when a goal is executed and whether the goal succeeds or fails.

Most built-in predicates are not re-executable; the description mentions the exceptional cases explicitly.

3.1.2 Templates and modes

3.1.2.1 Type of an argument

This TR defines the following additional argument types, augmenting those found on the ISO/IEC 13211 Prolog standard:

`mutex` — a term identifying a mutex (shall be regarded as an opaque type)

`mutex_or_alias` — a mutex or an atom acting as a mutex alias

`mutex_property` — a mutex property (2.7)

`queue` — a term identifying a queue (shall be regarded as an opaque type)

`queue_option` — a queue option (2.8)

`queue_options` — a list of queue options (2.8)

`queue_or_alias` — a queue or an atom acting as a queue alias

`queue_property` — a message queue property (2.10)

`thread` — a term identifying a thread (shall be regarded as an opaque type)

`thread_option` — a thread option (2.2)

`thread_options` — a list of thread options (2.3)

`thread_or_alias` — a thread or an atom acting as a thread alias

`thread_property` — a thread property (2.4)

`thread_status` — a thread status (2.1.3)

NOTE — The types `queue` and `queue_or_alias` also include, respectively, the types `thread` and `thread_or_alias` given that each thread is associated with a queue identified by the thread identifier or the thread alias.

3.2 Thread management

These built-in predicates enable threads to be created, detached, joined, and destroyed. They include predicates for setting and querying default thread creation options.

The examples provided for these built-in predicates assume the environment created from the following Prolog text:

```
:- initialization(management).

thread_goal :-
    thread_get_message(M),
    write(M).

signal_goal :-
```

```
    write('Hello, from signal goal!').

hello :-
    write('Hello!').
dog.

management :-
    thread_create(thread_goal, _,
        [detached(true), alias(thread_detached)]),
    thread_create(hello, _,
        [alias(thread_hello), local(512)]),
    thread_create(thread_goal, _, [alias(thread)]).
```

3.2.1 thread_create/3, thread_create/2, thread_create/1

3.2.1.1 Description

`thread_create(Goal, Thread, Options)` creates a new thread and starts it by executing `Goal`. If the thread is created successfully, the thread identifier of the created thread is unified to `Thread`. `Options` is a list of thread options (2.3). The `Goal` argument is copied to the new Prolog engine. This implies further instantiation of this term in either thread does not have consequences for the other thread.

3.2.1.2 Template and modes

```
thread_create(@callable_term, -thread, @thread_options)
```

3.2.1.3 Errors

a) `Goal` is a variable
— `instantiation_error`

b) `Goal` is neither a variable nor a callable term
— `type_error(callable, Goal)`

c) `Thread` is not a variable
— `type_error(variable, Thread)`

d) `Options` is a partial list or a list with an element `E` which is a variable
— `instantiation_error`

e) `Options` is neither a partial list nor a list
— `type_error(list, Options)`

f) An element of `E` of the `Options` list is not a valid thread option
— `domain_error(thread_option, E)`

g) The maximum number of threads has been reached
— `resource_error(threads)`

h) There is no memory available for creating the thread data areas
— `resource_error(memory)`

i) An element `E` of the `Options` list is `alias(A)` and `A` is already associated with an existing thread or queue
— `permission_error(create, thread, alias(A))`

NOTE — When the maximum number of threads is known by the Prolog compiler, its value shall be available using the Prolog flag `max_threads` (2.13.1).

3.2.1.4 Bootstrapped built-in predicates

The built-in predicates `thread_create/2` and `thread_create/1` provide similar functionality to the built-in predicate `thread_create/3`.

Goal `thread_create(Goal, Thread)` creates a new thread using default options (3.2.2). Goal `thread_create(Goal)` creates a new detached thread using default options (3.2.2)

```
thread_create(Goal, Thread) :-
    thread_create(Goal, Thread, []).

thread_create(Goal) :-
    thread_create(Goal, _, [detached(true)]).
```

3.2.1.5 Examples

These examples assume the environment has been created from the Prolog text defined at the beginning of section 3.2.

```
thread_create(hello, Thread, []).
    Succeeds.
    [It creates a thread, unifies Thread
    with a thread-term for the thread.]

thread_create(hello, Thread).
    Succeeds.
    [It creates a thread using de default
    thread-creation options, unifies Thread
    with a thread-term for the thread.]

thread_create(thread_goal, Thread, [alias(thread11)]).
    Succeeds.
    [It creates a thread, unifies Thread
```

```
    with a thread-term for the thread, and
    associates the alias 'thread11' with the thread.]

thread_create(Goal, Thread, []).
    instantiation_error.

thread_create(4, Thread, []).
    type_error(callable, 4).

thread_create(hello, thread, []).
    type_error(variable, thread).

thread_create(hello, Thread, Options).
    instantiation_error.

thread_create(hello, Thread, [alias(thread_x), X]).
    instantiation_error.

thread_create(thread_goal, Thread, options).
    type_error(list, options).

thread_create(thread_goal, Thread, [name(paul)]).
    domain_error(thread_option, name(paul)).

thread_create(thread_goal, Thread, [alias(thread)]).
    permission_error(create, thread, alias(thread)).
    [Assuming that there already exists a thread
     named 'thread'.]
```

3.2.2 thread_default/1

3.2.2.1 Description

`thread_default(Default)` gets the default options used when a new thread is created. The allowed values for `Default` are listed in section 2.2. Depending on the implementation, some default options may be read-only and thus not modifiable. The order in which thread options are found by `thread_default/1` is implementation dependent.

NOTE — There is no default value for the option `alias/1`.

Procedurally, `thread_default(Default)` is executed as follows:

a) Create a Set_D of all terms `(D)` such that `D` is a default option for thread creation,

b) If Set_D is empty, the goal fails,

c) Else, chooses a member (`DD`) of Set_D and removes it from the set,

d) Unifies `DD` with `Default`,

e) If unification succeeds, the goal succeeds,

f) Else proceeds to 3.2.2.1 b.

`thread_default(Default)` is re-executable. On backtracking, continue at 3.2.2.1 b.

3.2.2.2 Template and modes

```
thread_default(?thread_option)
```

3.2.2.3 Errors

a) `Default` is neither a variable nor a thread option
— `domain_error(thread_option, Default)`

3.2.2.4 Examples

```
thread_default(detached(Boolean)).
    Succeeds, unifying Boolean with the
    current detached default value.

thread_default(cpu(X)).
    domain_error(thread_option, cpu(X)).
```

3.2.3 thread_set_default/1

3.2.3.1 Description

`thread_set_default(Default)` sets a default option for thread creation. The items allowed in `Default` are listed in section 2.2. Some implementations may not support changing the value of some thread options.

NOTE — It is not possible to set a default value for the option `alias/1`.

3.2.3.2 Template and modes

```
thread_set_default(+thread_option)
```

3.2.3.3 Errors

a) `Default` is a variable
— `instantiation_error`

b) `Default` is neither a variable nor a thread option
— `domain_error(thread_option, Default)`

c) The thread option is read-only or does not support a default value
— `permission_error(modify, thread_option, Default)`

3.2.3.4 Examples

```
thread_set_default(detached(true)).
    Succeeds.

thread_set_default(Default).
    instantiation_error.

thread_set_default(cpu(100)).
    domain_error(thread_option, cpu(100)).

thread_set_default(detached(yes)).
    domain_error(thread_option, detached(yes)).
```

3.2.4 thread_self/1

3.2.4.1 Description

`thread_self(ThreadOrAlias)` is true when `ThreadOrAlias` is the identifier or an alias of the calling thread.

3.2.4.2 Template and modes

```
thread_self(+thread_or_alias)
thread_self(-thread)
```

3.2.4.3 Errors

a) `ThreadOrAlias` is neither a variable nor a thread-term or alias
— `domain_error(thread_or_alias, ThreadOrAlias)`

3.2.4.4 Examples

The goal `thread_self(Thread)` succeeds, unifying `Thread` with miguel.

```
:- initialization(main(paulo, miguel)).

child(Parent) :-
    thread_self(Thread),
    write('I\'am '),
    write(Thread),
    write(' child of '),
    write(Parent),
    write('.').

main(Parent, Child):-
    thread_create(child(Parent), T,
        [alias(Child), detached(false)]).

%
% Second example
%
child(Parent) :-
    thread_self(thread_s),
    ...
    domain_error(thread_or_alias, thread_s).
```

3.2.5 thread_join/2

3.2.5.1 Description

`thread_join(ThreadOrAlias, Status)` waits for the termination of thread with given `ThreadOrAlias`, unifying the result-status of the thread with `Status`. After this call, the `ThreadOrAlias` identifier or alias becomes invalid.

NOTES

1 After this call it is expected that all resources associated with the thread be reclaimed.
2 Threads with the property `detached(true)` cannot be joined.

3.2.5.2 Template and modes

```
thread_join(+thread_or_alias, -thread_status)
```

3.2.5.3 Errors

a) `ThreadOrAlias` is a variable
— `instantiation_error`

b) `ThreadOrAlias` is neither a variable nor a thread-term or alias
— `domain_error(thread_or_alias, ThreadOrAlias)`

c) `ThreadOrAlias` is not associated with a current thread
— `existence_error(thread, ThreadOrAlias)`

d) `ThreadOrAlias` does not correspond to a joinable thread
— `permission_error(join, thread, ThreadOrAlias)`

e) `Status` is not a variable
— `type_error(variable, Status)`

3.2.5.4 Examples

These examples assume the environment has been created from the Prolog text defined at the beginning of section 3.2.

```
thread_join(thread, Status).
    Succeeds, unifying Status with true.
    [After the thread receives a message.]

thread_join(Thread, Status).
    instantiation_error.

thread_join(dog, Status).
    domain_error(thread_or_alias, dog).

thread_join(thread5, Status).
    existence_error(thread, thread5).

thread_join(thread_detached, true).
    permission_error(join, thread, thread_detached).

thread_join(thread, status).
    type_error(variable, status).
```

```
:- initialization(main_join).

thread_goal :-
    thread_self(T),
    write('I\'am '),
    write(T), nl.

main_join :-
    write('I\'m main.'), nl,
    thread_create(thread_goal, T, [alias(t1)]),
    thread_join(t1, Status),
    write(Status).
```

The predicate `thread_join(t1, Status)` in the previous example succeeds, unifying `Status` with `true`.

3.2.6 thread_detach/1

3.2.6.1 Description

`thread_detach(ThreadOrAlias)` switches a thread into detached-state at runtime. `ThreadOrAlias` is the identifier of, or an alias to, the thread to be placed in a detached state.

Threads that are created as detached are expected to leave no traces if they terminate. Threads nobody is waiting for may be created normally and detach themselves just before completion. This way they leave no traces on normal completion and their reason for failure can be inspected. This guarantees that the resources consumed by the thread will be freed immediately when the thread terminates. However, this prevents other threads from synchronizing on the termination of `ThreadOrAlias` using `thread_join/2` (3.2.5). If, when `thread_detach/1` is called, the thread has already terminated, all remaining threads resources will be freed.

3.2.6.2 Template and modes

```
thread_detach(+thread_or_alias)
```

3.2.6.3 Errors

a) `ThreadOrAlias` is a variable
— `instantiation_error`

b) `ThreadOrAlias` is neither a variable nor a thread-term nor an alias
— `domain_error(thread_or_alias, ThreadOrAlias)`

c) `ThreadOrAlias` is not associated with a current thread
— `existence_error(thread, ThreadOrAlias)`

d) `ThreadOrAlias` is not joinable
— `permission_error(detach, thread, ThreadOrAlias)`

3.2.6.4 Examples

```
thread_detach(thread).
    Succeeds.

thread_detach(Thread).
    instantiation_error.

thread_detach(dog).
    domain_error(thread_or_alias, dog).

thread_detach(thread5).
```

```
    existence_error(thread, thread5).

thread_detach(thread_detached).
    permission_error(detach, thread, thread_detached).
    [Assumes that the thread has the thread-creation
    option detached(true).]
```

3.2.7 thread_exit/1

3.2.7.1 Description

`thread_exit(Term)` terminates the thread immediately, leaving `exited(Term)` as result-state for `thread_join/2` (3.2.5). If the thread has the thread-creation option `detached(true)` it terminates, but its exit status cannot be retrieved using `thread_join/2` making the value of `Term` irrelevant.

3.2.7.2 Template and modes

```
thread_exit(@nonvar)
```

3.2.7.3 Errors

a) `Term` is a variable
   — `instantiation_error`.

b) the calling thread is the main Prolog process thread
   — `permission_error(cancel, thread, main)`

3.2.7.4 Examples

```
:- initialization(main).

goal :-
    repeat,
    thread_get_message(M),
    (    M == 'exit' ->
         thread_exit(write('Exit'))
    ;    write(M), nl, fail
    ).

main :-
    thread_create(goal, _, [alias(t)]).

thread_join(t, Term).
    Succeeds, unifying Term with exited(write('Exit')).
    [Immediately after the execution:
    thread_send_message(t, exit).]
```

3.2.8 thread_signal/2

3.2.8.1 Description

`thread_signal(ThreadOrAlias, Goal)` makes thread `ThreadOrAlias` execute `Goal` at the first opportunity. The predicate `thread_signal/2` places `Goal` into the signaled thread's signal queue and returns immediately. `Goal` can be any valid Prolog goal, including `throw/1` to make the receiving thread generate an exception.

3.2.8.2 Template and modes

```
thread_signal(+thread_or_alias, @callable_term)
```

3.2.8.3 Errors

a) `ThreadOrAlias` is a variable
— `instantiation_error`

b) `ThreadOrAlias` is neither a variable nor a thread-term or alias
— `domain_error(thread_or_alias, ThreadOrAlias)`

c) `ThreadOrAlias` is not associated with a current thread
— `existence_error(thread, ThreadOrAlias)`

d) `Goal` is a variable
— `instantiation_error`

e) `Goal` is neither a variable nor a callable term
— `type_error(callable, Goal)`

3.2.8.4 Examples

```
thread_signal(thread_detached, signal_goal).
    Succeeds.

thread_signal(Thread, signal_goal).
    instantiation_error.

thread_signal(dog, signal_goal).
    domain_error(thread_or_alias, dog).

thread_signal(thread5, signal_goal).
    existence_error(thread, thread2).

thread_signal(thread, Goal).
    instantiation_error.
```

```
thread_signal(thread, 4).
   type_error(callable, 4).
```

In the following example, the predicate `main` creates two threads with alias `thread_1` and `thread_2`. Then `thread_2` sends a signal to `thread_1` with goal `write('from thread2')`.

```
:- initialization(main).

thread_1 :-
    thread_self(T),
    write('Thread '),
    write(T), nl,
    write('Send work to thread_2.'), nl
    thread_signal(t2, write('from thread_1')).

thread_2 :-
    thread_self(T),
    write('Thread '),
    write(T), nl,
    write('Receives from thread_1: '),
    write(M), nl.

main :-
    thread_create(thread_1, _, [alias(t1)]),
    thread_create(thread_2, _, [alias(t2)]).
```

3.2.9 thread_cancel/1

3.2.9.1 Description

`thread_cancel(ThreadOrAlias)` cancels a thread. Any mutexes held by the thread shall be automatically released. The main Prolog thread (2.15) cannot be cancelled. Other than this, any thread can cancel any other thread.

NOTE — It is expected that all the resources consumed by a thread will be released upon thread cancellation.

3.2.9.2 Template and modes

`thread_cancel(+thread_or_alias)`

3.2.9.3 Errors

a) `ThreadOrAlias` is a variable
   — `instantiation_error`

b) `ThreadOrAlias` is neither a variable nor a thread-term or alias
— `domain_error(thread_or_alias, ThreadOrAlias)`

c) `ThreadOrAlias` is not associated with a current thread
— `existence_error(thread, ThreadOrAlias)`

d) `ThreadOrAlias` is the main Prolog process thread
— `permission_error(cancel, thread, main)`

3.2.9.4 Examples

```
thread_cancel(thread).
    Succeeds.

thread_cancel(main).
    permission_error(cancel, thread, main).
```

3.2.10 thread_yield/0

3.2.10.1 Description

`thread_yield` forces the calling thread to relinquish use of its processor and to wait in the run queue before it is scheduled again. If the run queue is empty when the `thread_yield/0` predicate is called, the calling thread is immediately rescheduled.

3.2.10.2 Template and modes

```
thread_yield
```

3.2.10.3 Errors

None.

3.2.10.4 Examples

```
thread_yield.
    Succeeds.
```

3.2.11 thread_sleep/1

3.2.11.1 Description

`thread_sleep(Seconds)` suspends the current threads execution until `Seconds` seconds elapsed. When the number of seconds is zero or a negative value, the call succeeds and returns immediately.

NOTE — The thread shall not be awakened before this time but, depending on the implementation of the thread scheduler, it might be awakened later.

3.2.11.2 Template and modes

`thread_sleep(+number)`

3.2.11.3 Errors

a) `Seconds` is a variable
— `instantiation_error`

b) `Seconds` is neither a variable nor a number
— `type_error(number, Seconds)`

3.2.11.4 Examples

```
thread_sleep(Seconds).
    instantiation_error.

thread_sleep(dog).
    type_error(number, dog).

thread_sleep(1.5).
    Succeeds.

thread_sleep(-10).
    Succeeds.

%
% The thread simply count from 1 to N,
% incremented one time per second.
%
:- initialization(counter(10)).

counter(N) :-
    thread_create(do_counter(N), _, [detached(true)]).

do_counter(0) :- !.
do_counter(N) :-
    M is N - 1,
    thread_sleep(1),
    write(M), nl,
    do_counter(M).
```

3.3 Monitoring threads

The predicates described in this section are provided for diagnosing and monitoring threads. Most multi-threaded applications should not need the predicates from this section because almost any usage of these predicates is unsafe. For

example checking the existence of a thread before signaling it is of no use as it may vanish between the two calls. Catching exceptions using `catch/3` is the only safe way to deal with thread-existence errors.

3.3.1 thread_property/2, thread_property/1

3.3.1.1 Description

`thread_property(ThreadOrAlias, Property)` enumerates thread properties.

Procedurally, `thread_property(ThreadOrAlias, Property)` is executed as follows:

a) Create a Set_{TP} of all terms `(T, P)` such that `T` is a current thread which has property `P`,

b) If Set_{TP} is empty, the goal fails,

c) Else, chooses a member `(TT, PP)` of Set_{TP} and removes it from the set,

d) Unifies `TT` with `ThreadOrAlias`, and `PP` with Property,

e) If unification succeeds, the goal succeeds,

f) Else proceeds to 3.3.1.1 b.

`thread_property(ThreadOrAlias, Property)` is re-executable. On backtracking, continue at 3.3.1.1 b.

3.3.1.2 Template and modes

```
thread_property(?thread_or_alias, ?thread_property)
```

3.3.1.3 Errors

a) `ThreadOrAlias` is neither a variable nor a thread-term or alias
— `domain_error(thread_or_alias, ThreadOrAlias)`

b) `ThreadOrAlias` is not associated with a current thread
— `existence_error(thread, ThreadOrAlias)`

c) `Property` is neither a variable nor a thread property
— `domain_error(thread_property, Property)`

3.3.1.4 Bootstrapped built-in predicate

The built-in predicate `thread_property/1` provides similar functionality to the built-in predicate `thread_property/2`.

Goal `thread_property(Property)` allows access to the properties of the calling thread.

```
thread_property(Property) :-
    thread_self(Thread),
    thread_property(Thread, Property).
```

3.3.1.5 Examples

These examples assume the environment has been created from the Prolog text defined at the beginning of section 3.2.

```
thread_property(thread, alias(thread)).
    Succeeds.

thread_property(thread, alias(N)).
    Succeeds, unifying N with thread.

thread_property(thread, status(Status)).
    Succeeds, unifying Status with running.

thread_property(thread, P).
    Succeeds, unifying P with a property of the thread.
    On re-execution, succeeds, in turn with
    each property of the thread.

thread_property(T, P).
    Succeeds, unifying T with the thread and P with a
    property of the thread.
    On re-execution, succeeds, in turn with
    each property of the thread and in turn with thread.

thread_property(thread_detached, detached(false)).
    Fails.

thread_property(dog, P).
    domain_error(thread_or_alias, dog).

thread_property(thread_detacehd, man(paul)).
    domain_error(thread_property, man(paul)).
```

3.3.2 thread_statistics/3, thread_statistics/2

3.3.2.1 Description

`thread_statistics(ThreadOrAlias, Key, Value)` obtains statistical information on thread `ThreadOrAlias`. The possible values of `Key` are implementation defined. The order in which the keys are found is implementation dependent. The implementation of this predicate is optional for Prolog compilers compliant with this TR due to the implied computation overhead in keeping statistical information.

Procedurally, `thread_statistics(ThreadOrAlias, Key, Value)` is executed as follows:

a) Create a Set_{TKV} of all terms `(T, K, V)` such that `T` is a current thread with statistics `K` which has the value `V`,

b) If Set_{TKP} is empty, the goal fails,

c) Else, chooses a member `(TT, KK, VV)` of Set_{TKV} and removes it from the set,

d) Unifies `TT` with `ThreadOrAlias`, `KK` with `Key`, and `VV` with Value,

e) If unification succeeds, the goal succeeds,

f) Else proceeds to 3.3.2.1 b.

`thread_statistics(ThreadOrAlias, Key, Value)` is re-executable. On backtracking, continue at 3.3.2.1 b.

3.3.2.2 Template and modes

```
thread_statistics(?thread_or_alias, ?key, ?value)
```

3.3.2.3 Errors

a) `MutexOrAlias` is neither a variable nor a mutex-term or alias
— `domain_error(mutex_or_alias, MutexOrAlias)`

b) `Key` is neither a variable nor an atom
— `type_error(atom, Key)`

c) `Value` is neither a variable nor an atom
— `type_error(variable, Value)`

3.3.2.4 Bootstrapped built-in predicate

The built-in predicate `thread_statistics/2` provides similar functionality to the built-in predicate `thread_statistics/3`.

Goal `thread_statistics(Key, Value)` allows access to the statistics information of the calling thread.

```
thread_statistics(Key, Value) :-
    thread_self(Thread),
    thread_statistics(Thread, Key, Value).
```

3.3.2.5 Examples

These examples assume the environment has been created from the Prolog text defined at the beginning of section 3.2.

```
thread_statistics(thread, cputime, Time).
    Succeeds, unifying Time with the CPU time.
    [Assumes that the cputime key are defined.]

thread_statistics(thread, disktime, Time).
    Fails.
    [Assumes that the disktime key are not defined.]

thread_statistics(dog, cputime, Time).
    domain_error(thread_or_alias, dog).

thread_statistics(thread, 1, Time).
    type_error(atom, Key).
```

3.4 Thread synchronization

These built-in predicates enable thread synchronization using a *mutex*, which is an abbreviation for *mutual exclusion*. Mutex built-in predicates provide for creating, destroying, locking, and unlocking mutexes. In addition, predicates for getting runtime mutex usage statistics are also specified.

All internal Prolog operations are thread-safe. This implies two Prolog threads can operate on the same dynamic predicate without corrupting the consistency of the predicate. A mutex is a MUTual EXclusive device, which implies at most one thread can hold a mutex.

Mutexes are used to realize related updates to the Prolog database. With *related*, we refer to the situation where a *transaction* implies two or more changes to the Prolog database. For example, we have a predicate `address/2`, representing the address of a person and we want to change the address by retracting the

old and asserting the new address. Between these two operations the database is invalid: this person has either no address or two addresses, depending on the `assert/retract` order.

The examples provided for these built-in predicate assume the environment created from the following Prolog text:

```
:- initialization(init).

init :-
    mutex_create(_,[alias(mutex1)]),
    mutex_create(_,[alias(mutex2)]),
    mutex_create(_,[alias(mutex3)]),
    mutex_create(_,[alias(mutex21)]),
    mutex_create(_,[alias(mutex22)]),
    mutex_create(_,[alias(mutex33)]),
    mutex_lock(mutex22),
    mutex_create(_, [alias(bank)]),
    mutex_create(_, [alias(addressbook)]).

cat.
dog.

:- dynamic(balance/2).

balance(paul, 1000).

bank_transation(Id, Deposit) :-
    balance(Id, V),
    T is V + Deposit,
    retractall(balance(Id, _)),
    asserta(balance(Id, T)).

:- dynamic(address/2).

address(paul, guarda).

change_address(Id, Address) :-
    mutex_lock(addressbook),
    retractall(address(Id, _)),
    asserta(address(Id, Address)),
    mutex_unlock(addressbook).
```

3.4.1 mutex_create/2, mutex_create/1

3.4.1.1 Description

`mutex_create(Mutex, Options)` creates a new mutex using the given list of options. The variable `Mutex` is unified with the mutex identifier.

3.4.1.2 Template and modes

```
mutex_create(-mutex, @mutex_options)
```

3.4.1.3 Errors

a) `Mutex` is not a variable
— `type_error(variable, Mutex)`

b) `Options` is a partial list or a list with an element `Option` which is a variable
— `instantiation_error`

c) `Options` is neither a partial list nor a list
— `type_error(list, Options)`

d) An element of `Option` of the `Options` list is not a valid mutex option
— `domain_error(mutex_option, Option)`

e) An element `Option` of the `Options` list is `alias(Alias)` and `Alias` is not an atom
— `type_error(atom, Alias)`

f) An element `Option` of the `Options` list is `alias(Alias)` and `Alias` is already associated with an existing mutex
— `permission_error(create, mutex, Alias)`

3.4.1.4 Bootstrapped built-in predicate

The built-in predicate `mutex_create/1` provides similar functionality to the built-in predicate `mutex_create/2`.

Goal `mutex_create(Mutex)` creates a new mutex with alias `Mutex` when the argument is instantiated to an atom; otherwise `Mutex` must be a variable which will be instantiated to a new mutex identifier. I.e the predicate behaves as defined by the following clause:

```
mutex_create(Mutex) :-
    (   atom(Mutex) ->
        mutex_create(_, [alias(Mutex)])
    ;   mutex_create(Mutex, [])
    ).
```

3.4.1.5 Examples

```
mutex_create(Mutex, []).
    Succeeds.
    [It creates a mutex using the default mutex-creation
     options, unifying Mutex with the new mutex identifier.]

mutex_create(Mutex).
    Succeeds.
    [It creates a mutex using the default mutex-creation
     options, unifying Mutex with the new mutex identifier.]

mutex_create(Mutex, [alias(mutex_1)]).
    Succeeds.
    [It creates a mutex, unifying Mutex with a mutex-term
     for the mutex, and associating the alias ‘mutex_1’
     with the mutex.]

mutex_create(mutex, []).
    type_error(variable, mutex).

mutex_create(mutex, [alias(mutex_x), X]).
    instantiation_error.

mutex_create(Mutex, options).
   type_error(list, options).

mutex_create(Mutex, [alias(mutex_1), size(512)]).
    domain_error(mutex_option, size(512))

mutex_create(Mutex, [alias(mutex1)]).
    permission_error(create, mutex, mutex1).
    [The mutex named mutex1, has been created previously]
```

3.4.2 mutex_destroy/1

3.4.2.1 Description

`mutex_destroy(MutexOrAlias)` destroys a mutex. After this call, the mutex identifier or alias, `MutexOrAlias`, becomes invalid.

3.4.2.2 Template and modes

```
mutex_destroy(+mutex_or_alias)
```

3.4.2.3 Errors

a) `MutexOrAlias` is a variable
— `instantiation_error`

b) `MutexOrAlias` is neither a variable nor a mutex-term or alias
— `domain_error(mutex_or_alias, MutexOrAlias)`

c) `MutexOrAlias` is not associated with a current mutex
— `existence_error(mutex, MutexOrAlias)`

d) `MutexOrAlias` the mutex is locked
— `permission_error(destroy, mutex, MutexOrAlias)`

3.4.2.4 Examples

These examples assume the environment has been created from the Prolog text defined at the beginning of section 3.4.

```
mutex_destroy(mutex1).
    Succeeds.

mutex_destroy(Mutex).
   instantiation_error.

mutex_destroy(cat).
    domain_error(mutex_or_alias, cat).

mutex_destroy(mutex112).
    existence_error(mutex, mutex112).

mutex_lock(mutex1), mutex_destroy(mutex1).
    permission_error(destroy, mutex, mutex1).
```

3.4.3 with_mutex/2

3.4.3.1 Description

`with_mutex(MutexOrAlias, Goal)` executes `Goal` while holding `MutexOrAlias`. The call blocks if the mutex is not available until it is released. If `Goal` leaves choice-points, these are destroyed. The mutex is unlocked regardless of whether `Goal` succeeds, fails, or raises an exception. An exception thrown by `Goal` is re-thrown after the mutex has been successfully unlocked.

NOTE — The predicate `with_mutex/2` allows the execution of a Goal as an atomic transaction, without the need for explicit synchronization by the programmer.

3.4.3.2 Template and modes

```
with_mutex(+mutex_or_alias, +callable_term)
```

3.4.3.3 Errors

a) `MutexOrAlias` is a variable
— `instantiation_error`

b) `MutexOrAlias` is neither a variable nor a mutex-term or alias
— `domain_error(mutex_or_alias, MutexOrAlias)`

c) `MutexOrAlias` is not associated with a current mutex
— `existence_error(mutex, MutexOrAlias)`

d) `Goal` is a variable
— `instantiation_error`

e) `Goal` is neither a variable nor a callable term
— `type_error(callable, Goal)`

3.4.3.4 Examples

These examples assume the environment has been created from the Prolog text defined at the beginning of section 3.4.

```
with_mutex(bank, bank_transation(paul, 200)).
    Succeeds, noting the list of clauses to be retracted
        = [ balance(paul, 1000) ], and the list of clauses
        to be asserted = [ balance(paul, 1200).]

with_mutex(M, bank_transation(paul, 200)).
    instantiation_error.

with_mutex(dog, bank_transation(paul, 200)).
    domain_error(mutex_or_alias, dog).

with_mutex(mutex_bank, bank_transation(paul, 200)).
    existence_error(mutex, mutex_bank).

with_mutex(bank, Goal).
    instantiation_error.

with_mutex(bank, (bear :- 4)).
    type_error(callable, 4).
```

3.4.4 mutex_lock/1

3.4.4.1 Description

`mutex_lock(MutexOrAlias)` locks a mutex. Prolog mutexes shall act as recursive mutexes, enabling them to be locked multiple times by the same thread. Only after unlocking it as many times as it is locked, the mutex shall become available for locking by other threads. If another thread has locked the mutex the calling thread is suspended until to mutex is unlocked.

NOTE — Locking and unlocking mutexes should be paired carefully in order to avoid deadlocks. In particular, the programmer shall ensure that mutexes are properly unlocked even if the protected code fails or raises an exception. For most common cases, the built-in predicate `with_mutex/2` (3.4.3) provides a safer way for using mutexes.

3.4.4.2 Template and modes

```
mutex_lock(+mutex_or_alias)
```

3.4.4.3 Errors

a) `MutexOrAlias` is a variable
   — `instantiation_error`

b) `MutexOrAlias` is neither a variable nor a mutex-term or alias
   — `domain_error(mutex_or_alias, MutexOrAlias)`

c) `MutexOrAlias` is not associated with a current mutex
   — `existence_error(mutex, MutexOrAlias)`

3.4.4.4 Examples

These examples assume the environment has been created from the Prolog text defined at the beginning of section 3.4.

```
mutex_lock(mutex1).
    Succeeds.

mutex_lock(mutex1), mutex_lock(mutex1).
    Succeeds.
    [A mutex may be locked several times by the same thread.]

mutex_lock(M).
    instantiation_error.

mutex_lock(dog).
```

```
    domain_error(mutex_or_alias, dog).

mutex_lock(mutex4).
    existence_error(mutex, mutex4).

change_address(paul, alfarazes).
    Succeeds, noting the list of clauses to be retracted
        = [ address(paul, guarda) ], and the list of clauses
        to be asserted = [ address(paul, alfarazes).]
```

3.4.5 mutex_trylock/1

3.4.5.1 Description

`mutex_trylock(MutexOrAlias)` as `mutex_lock/1` (3.4.4), but if the mutex is held by another thread, this predicates fails immediately.

3.4.5.2 Template and modes

```
mutex_trylock(+mutex_or_alias)
```

3.4.5.3 Errors

a) `MutexOrAlias` is a variable
   — `instantiation_error`

b) `MutexOrAlias` is neither a variable nor a mutex-term or alias
   — `domain_error(mutex_or_alias, MutexOrAlias)`

c) `MutexOrAlias` is not associated with a current mutex
   — `existence_error(mutex, MutexOrAlias)`

3.4.5.4 Examples

These examples assume the environment has been created from the Prolog text defined at the beginning of section 3.4.

```
mutex_trylock(mutex1).
    Succeeds.

mutex_trylock(mutex22).
    Fails.
    [The mutex named mutex22 has been locked.]

mutex_trylock(M).
    instantiation_error.
```

```
mutex_trylock(dog).
    domain_error(mutex_or_alias, dog).

mutex_trylock(mutex4).
    existence_error(mutex, mutex4).
```

3.4.6 mutex_unlock/1

3.4.6.1 Description

`mutex_unlock(MutexOrAlias)` unlocks a mutex. This predicate must be called by the thread holding the mutex.

3.4.6.2 Template and modes

`mutex_unlock(+mutex_or_alias)`

3.4.6.3 Errors

a) `MutexOrAlias` is a variable
— `instantiation_error`

b) `MutexOrAlias` is neither a variable nor a mutex-term or alias
— `domain_error(mutex_or_alias, MutexOrAlias)`

c) `MutexOrAlias` is not associated with a current mutex
— `existence_error(mutex_or_alias, MutexOrAlias)`

d) the calling thread do not hold the mutex `MutexOrAlias`
— `permission_error(unlock, mutex, MutexOrAlias)`

3.4.6.4 Examples

These examples assume the environment has been created from the Prolog text defined at the beginning of section 3.4.

```
mutex_lock(mutex22).
    Succeeds.

mutex_unlock(dog).
    domain_error(mutex_or_alias, dog).

mutex_unlock(mutex4).
    existence_error(mutex, mutex4).

mutex_unlock(mutex3).
    permission_error(unlock, mutex, mutex3).
    [Mutex was not locked previously.]
```

3.4.7 mutex_unlock_all/0

3.4.7.1 Description

`mutex_unlock_all` unlocks all mutexes held by the current thread.

3.4.7.2 Template and modes

```
mutex_unlock_all
```

3.4.7.3 Errors

None.

3.4.7.4 Examples

These examples assume the environment has been created from the Prolog text defined at the beginning of section 3.4.

```
mutex_unlock_all.
    Succeeds.
```

3.4.8 mutex_property/2

3.4.8.1 Description

`mutex_property(MutexOrAlias, Property)` enumerates the properties of all current mutexes.

Procedurally, `mutex_property(MutexOrAlias, Property)` is executed as follows:

a) Create a Set_{MP} of all terms `(M, P)` such that `M` is a current mutex which has property `P`,

b) If Set_{MP} is empty, the goal fails,

c) Else, chooses a member `(MM, PP)` of Set_{MP} and removes it from the set,

d) Unifies `MM` with `MutexOrAlias`, and `PP` with Property,

e) If unification succeeds, the goal succeeds,

f) Else proceeds to 3.4.8.1 b.

`mutex_property(MutexOrAlias, Property)` is re-executable. On backtracking, continue at 3.4.8.1 b.

3.4.8.2 Template and modes

```
mutex_property(?mutex_or_alias, ?mutex_property)
```

3.4.8.3 Errors

a) `MutexOrAlias` is neither a variable nor a mutex-term or alias
— `domain_error(mutex_or_alias, MutexOrAlias)`

b) `MutexOrAlias` is not associated with a current mutex
— `existence_error(mutex, MutexOrAlias)`

c) `Property` is neither a variable nor a mutex property
— `domain_error(mutex_property, Property)`

3.4.8.4 Examples

These examples assume the environment has been created from the Prolog text defined at the beginning of section 3.4.

```
mutex_property(mutex21, alias(mutex21)).
    Succeeds.

mutex_property(mutex21, alias(A)).
    Succeeds, unifying A with mutex21.

mutex_property(mutex21, status(locked(main, C))).
    Succeeds, unifying C with 1.

mutex_property(mutex22, status(locked(main, C))).
    Succeeds, unifying C with 2.

mutex_property(Mutex, Property).
    Succeeds, unifying Mutex with a mutex-term of an existing mutex
    and Property with a property of the mutex.
    On re-execution, succeeds, in turn with
    each property of the mutex.

mutex_property(Mutex, status(locked(Thread, 1))).
    Succeeds, unifying Mutex  with a mutex-term of an existing
    mutex and Thread with the thread that holding the mutex, and has
    locked one time.
    On re-execution, succeeds, in turn with each mutex.

mutex_property(mutex22, status(locked(main, 1))).
    Fails.
```

```
mutex_property(dog, status(locked(main, 1))).
    domain_error(mutex_or_alias, dog).

mutex_property(mutex4, status(locked(main, 1))).
    existence_error(mutex, mutex4).

mutex_property(mutex4, status(locked(dog, 1))).
    domain_error(thread_or_alias, dog)}

mutex_property(mutex22, status(locked(main, -2))).
    domain_error(not_less_than_zero, -2).
```

3.4.9 mutex_statistics/3

3.4.9.1 Description

`mutex_statistics(MutexOrAlias, Key, Value)` obtains statistical information of the `MutexOrAlias` for the calling thread. The possible values of `Key` are implementation defined. The implementation of this predicate is optional for Prolog compilers compliant with this TR due to the implied computation overhead in keeping statistical information.

NOTE — Two suggested key values are `acquisitions` (the number of times the thread acquired the mutex) and `collisions` (the number of times a thread failed to acquire the mutex).

Procedurally, `mutex_statistics(MutexOrAlias, Key, Value)` is executed as follows:

a) Create a Set_{MKV} of all terms `(M, K, V)` such that `M` is a current mutex with statistics `K` which has the value `V`,

b) If Set_{MKP} is empty, the goal fails,

c) Else, chooses a member `(MM, KK, VV)` of Set_{MKV} and removes it from the set,

d) Unifies `MM` with `MutexOrAlias`, `KK` with `Key`, and `VV` with Value,

e) If unification succeeds, the goal succeeds,

f) Else proceeds to 3.4.9.1 b.

`mutex_statistics(MutexOrAlias, Key, Value)` is re-executable. On backtracking, continue at 3.4.9.1 b.

3.4.9.2 Template and modes

```
mutex_statistics(?mutex_or_alias, ?key, ?value)
```

3.4.9.3 Errors

a) `MutexOrAlias` is neither a variable nor a mutex-term or alias
— `domain_error(mutex_or_alias, MutexOrAlias)`

b) `Key` is neither a variable nor an atom
— `type_error(atom, Key)`

c) `Value` is neither a variable nor an atom
— `type_error(variable, Value)`

3.4.9.4 Examples

3.5 Message queues

Prolog threads may exchange data using dynamic predicates. However, these provide no suitable means to wait for data or a condition as they can only be checked in an expensive polling loop. Message queues provide a means for threads to wait for data or conditions without consuming computation time. Explicit message queues are commonly used when implementing worker-pool models, where multiple threads wait on a single queue and pick up the first goal to execute.

The built-in predicates presented in this section enables message queue management, monitoring, and communication.

NOTE — Each thread has a message queue attached to it that is identified by the thread.

The examples provided for these built-in predicate assume the pool created from the following Prolog text:

```
:- initialization(main).

main :-
    thread_create(thread_goal, _, [alias(t1)]),
    message_queue_create(_, [alias(q1)]),
    message_queue_create(_, [alias(q2), max_size(20)]).

noqueue.

thread_goal :-
    repeat,
    thread_get_message(q2, M),
    write(M), nl,
    fail.
```

3.5.1 message_queue_create/2, message_queue_create/1

3.5.1.1 Description

`message_queue_create(Queue, Options)` creates a new queue using the given list of options. The variable `Queue` is unified with the queue identifier.

3.5.1.2 Template and modes

```
message_queue_create(-queue, @queue_options)
```

3.5.1.3 Errors

a) `Queue` is not a variable
— `type_error(variable, Queue)`

b) `Options` is a partial list or a list with an element `E` which is a variable
— `instantiation_error`

c) `Options` is neither a partial list nor a list
— `type_error(list, Options)`

d) An element of `E` of the `Options` list is not a valid queue option
— `domain_error(queue_option, E)`

e) An element `E` of the `Options` list is `alias(A)` and `A` is already associated with an existing thread or queue
— `permission_error(create, queue, alias(A))`

3.5.1.4 Bootstrapped built-in predicate

The built-in predicate `message_queue_create/1` provides similar functionality to the built-in predicate `message_queue_create/2`.

Goal `message_queue_create(Queue)` creates a new queue using default options.

```
message_queue_create(Queue) :-
    message_queue_create(Queue, []).
```

3.5.1.5 Examples

```
message_queue_create(Queue, []).
    Succeeds.
    [It creates a queue, unifying Queue
    with a queue-term for the queue.]

message_queue_create(Queue).
```

```
    Succeeds.
    [It creates a queue using de default
    queue-creation options, unifying Queue
    with a queue-term for the queue.]

message_queue_create(Queue, [alias(q3)]).
    Succeeds.
    [It creates a queue, unifies Queue with a
    queue-term for the queue, and  associates
    the alias 'q3' with the queue.]

message_queue_create(Queue, [alias(q3), max_size(20)]).
    Succeeds.
    [It creates a queue with maximum size 20, unifies Queue
    with a queue-term for the new queue, and
    associates the alias 'q3' with the queue.]

message_queue_create(queue).
    type_error(variable, queue).

message_queue_create(Queue, [alias(q3), X]).
    instantiation_error.

message_queue_create(Queue, options).
    type_error(list, options).

message_queue_create(Queue, [alias(q2), name(q2)]).
    domain_error(queue_option, name(q2)).

message_queue_create(Q, [alias(q1)]).
    permission_error(create, queue, alias(q1)).
    [Assume that already exists one
     queue named 'q1'.]
```

3.5.2 message_queue_destroy/1

3.5.2.1 Description

`message_queue_destroy(QueueOrAlias)` destroys a message queue created with `message_queue_create/1-2` built-in predicates (3.5.1). It is not allowed to destroy the queue of a thread. Neither is it allowed to destroy a queue other threads are waiting for or may try to wait for later.

3.5.2.2 Template and modes

```
message_queue_destroy(+queue_or_alias)
```

3.5.2.3 Errors

a) `QueueOrAlias` is a variable
— `instantiation_error`

b) `QueueOrAlias` is neither a variable nor a queue-term or alias
— `domain_error(queue_or_alias, QueueOrAlias)`

c) `QueueOrAlias` is the identifier of an existing thread
— `permission_error(destroy, thread_queue, QueueOrAlias)`

d) `QueueOrAlias` there are threads waiting for, or for anonymous message queues
— `permission_error(destroy, queue, QueueOrAlias)`

e) `QueueOrAlias` is not associated with a current queue
— `existence_error(queue, QueueOrAlias)`

3.5.2.4 Examples

These examples assume the environment has been created from the Prolog text defined at the beginning of 3.5.

```
message_queue_destroy(queue1).
    Succeeds.

message_queue_destroy(Queue).
    instantiation_error.

message_queue_destroy(noqueue).
    domain_error(queue_or_alias, noqueue).

message_queue_destroy(t1).
    permission_error(destroy, thread_queue, t1).
    [It is not allowed to destroy the queue of a thread]

message_queue_destroy(q2).
    permission_error(destroy, queue, q2).
    [The thread named t1 is waiting for a message]

message_queue_destroy(q3).
    existence_error(queue, q3).
```

3.5.3 message_queue_property/2

3.5.3.1 Description

`message_queue_property(QueueOrAlias, Property)` enumerates message queue properties.

Procedurally, `message_queue_property(QueueOrAlias, Property)` is executed as follows:

a) Creates a Set_{QP} of all terms `(Q, P)` such that `T` is a current message queue which has property `P`,

b) If Set_{QP} is empty, the goal fails,

c) Else, chooses a member `(QQ, PP)` of Set_{QP} and removes it from the set,

d) Unifies `QQ` with `QueueOrAlias`, and `PP` with Property,

e) If unification succeeds, the goal succeeds,

f) Else proceeds to 3.5.3.1 b.

`message_queue_property(QueueOrAlias, Property)` is re-executable. On backtracking, continue at 3.5.3.1 b.

3.5.3.2 Template and modes

```
message_queue_property(?queue_or_alias, ?queue_property)
```

3.5.3.3 Errors

a) `QueueOrAlias` is neither a variable nor a queue-term or alias
— `domain_error(queue_or_alias, QueueOrAlias)`

b) `QueueOrAlias` is not associated with a current message queue
— `existence_error(queue, QueueOrAlias)`

c) `Property` is neither a variable nor a message queue property
— `domain_error(queue_property, Property)`

3.5.3.4 Examples

```
message_queue_property(q1, size(Size)).
    Succeeds, unifying Size with 0.

message_queue_property(q2, size(Size)).
    Succeeds, unifying Size with 0.

message_queue_property(q2, size(4)).
    Fails.

message_queue_property(noqueue, Size).
```

```
    domain_error(queue_or_alias, noqueue).

message_queue_property(q3, Property).
    existence_error(queue, q3).

message_queue_property(q2, size(size)).
    type_error(integer, size).
```

3.6 Thread and message queue communication

These built-in predicates enable thread and message queue communication, and queue monitoring.

Prolog threads can exchange data using dynamic predicates, database records, and other globally shared data. These provide no suitable means to wait for data or a condition as they can only be checked in an expensive polling loop. Message queues provide a means for threads to wait for data or conditions without using the CPU. Each thread has a message queue attached to it that can be referenced using the thread identifier or alias (if defined).

The examples provided for these built-in predicate assume the environment created from the following Prolog text:

```
:- initialization(main).

thread_goal :-
    repeat,
    thread_get_message(M),
    write(M), nl,
    fail.

main :-
    message_queue_create(_, [alias(queue1)]),
    message_queue_create(_, [alias(queue2), max_size(2)]),
    thread_create(thread_goal, _, [alias(thread1)]),
    thread_send_message(queue2, father(antonio)),
    thread_send_message(queue2, mother(maria)).
```

3.6.1 thread_send_message/2, thread_send_message/1

3.6.1.1 Description

`thread_send_message(QueueOrAlias, Term)` places `Term` in a message queue. Any term may be placed in a message queue. The term is copied to the receiving queue; thus, any variable-bindings are lost. This call returns immediately.

3.6.1.2 Template and modes

```
thread_send_message(+queue_or_alias, @term)
```

3.6.1.3 Errors

a) `QueueOrAlias` is a variable
— `instantiation_error`

b) `QueueOrAlias` is neither a variable nor a queue-term or alias
— `domain_error(queue_or_alias, QueueOrAlias)`

c) `QueueOrAlias` is not associated with a current queue
— `existence_error(queue, QueueOrAlias)`

3.6.1.4 Bootstrapped built-in predicate

The built-in predicate `thread_send_message/1` provides similar functionality to the built-in predicate `thread_send_message/2`.

Goal `thread_send_message(Term)` sends a message to the message queue of the calling thread.

```
thread_send_message(Term) :-
    thread_self(Thread),
    thread_send_message(Thread, Term).
```

3.6.1.5 Examples

These examples assume the environment has been created from the Prolog text defined at the beginning of 3.6.

```
thread_send_message(queue1, father(antonio)).
    Succeeds.

thread_send_message(thread1, father(antonio)).
    Succeeds.

thread_send_message(queue2, father(paul)).
    Succeeds.

thread_send_message(Queue, mother(maria)).
    instantiation_error.

thread_send_message(noqueue, mother(maria)).
    domain_error(queue_or_alias, noqueue).
```

```
thread_send_message(queue3, mother(maria)).
    existence_error(queue, queue3).
```

3.6.2 thread_get_message/2, thread_get_message/1

3.6.2.1 Description

`thread_get_message(QueueOrAlias, Term)` gets a `Term` from a message queue. The predicate walks the message queue and if necessary blocks execution until a term that unifies to `Term` arrives in the queue. After a term from the queue has been unified to `Term`, the term is deleted from the queue and this predicate returns. Note that non-unifying messages remain in the queue.

3.6.2.2 Template and modes

```
thread_get_message(+queue_or_alias, ?term)
```

3.6.2.3 Errors

a) `QueueOrAlias` is a variable
   — `instantiation_error`

b) `QueueOrAlias` is neither a variable nor a queue-term or alias
   — `domain_error(queue_or_alias, QueueOrAlias)`

c) `QueueOrAlias` is not associated with a current queue
   — `existence_error(queue, QueueOrAlias)`

3.6.2.4 Bootstrapped built-in predicate

The built-in predicate `thread_get_message/1` provides similar functionality to the built-in predicate `thread_get_message/2`.

Goal `thread_get_message(Term)` gets a message from the message queue of the calling thread.

```
thread_get_message(Term) :-
    thread_self(Thread),
    thread_get_message(Thread, Term).
```

3.6.2.5 Examples

These examples assume the environment has been created from the Prolog text defined at the beginning of 3.6.

```
thread_get_message(queue2, father(antonio)).
    Succeeds.
```

```
thread_get_message(queue2, father(Father)).
    Succeeds, unifying Father with antonio.

thread_get_message(queue2, Message).
    Succeeds, unifying Message with father(antonio).

thread_get_message(Queue, mother(maria)).
    instantiation_error.

thread_get_message(noqueue, mother(maria)).
    domain_error(queue_or_alias, noqueue).

thread_get_message(queue3, mother(maria)).
    existence_error(queue, queue3).
```

After the following has been executed, thread_1 has the term b(gnu) in its queue and continues execution using `A` is gnat.

```
<thread_1>
    thread_get_message(a(A)),

<thread_2>
    thread_send_message(Thread_1, b(gnu)),
    thread_send_message(Thread_1, a(gnat)),
```

3.6.3 thread_peek_message/2, thread_peek_message/1

3.6.3.1 Description

`thread_peek_message(QueueOrAlias, Term)` walks a message queue and compares the queued terms with `Term` until one unifies or the end of the queue has been reached. In the first case the call succeeds (possibly instantiating `Term`). If no term from the queue unifies this call fails.

3.6.3.2 Template and modes

`thread_peek_message(+queue_or_alias, ?term)`

3.6.3.3 Errors

a) `QueueOrAlias` is a variable
   — `instantiation_error`

b) `QueueOrAlias` is neither a variable nor a queue-term or alias
   — `domain_error(queue_or_alias, QueueOrAlias)`

c) `QueueOrAlias` is not associated with a current queue
— `existence_error(queue, QueueOrAlias)`

3.6.3.4 Bootstrapped built-in predicate

The built-in predicate `thread_peek_message/1` provides similar functionality to the built-in predicate `thread_peek_message/2`.

Goal `thread_peek_message(Term)` searches the calling thread queue for a message match.

```
thread_peek_message(Term) :-
    thread_self(Thread),
    thread_peek_message(Thread, Term).
```

3.6.3.5 Examples

These examples assume the environment has been created from the Prolog text defined at the beginning of 3.6.

```
thread_peek_message(queue2, father(antonio)).
    Succeeds.

thread_peek_message(queue2, mother(maria)).
    Succeeds.

thread_peek_message(queue2, father(Father)).
    Succeeds, unifying Father with antonio.

thread_peek_message(queue2, Message).
    Succeeds, unifying Message with father(antonio).

thread_peek_message(queue2, father(anne)).
    Fails.

thread_peek_message(Queue, mother(maria)).
    instantiation_error.

thread_peek_message(noqueue, mother(maria)).
    domain_error(queue_or_alias, noqueue).

thread_peek_message(queue3, mother(maria)).
    existence_error(queue, queue3).
```