CRAM

Design & Implementation of a Reactive Plan Language

Tobias Christian Rittweiler

May 5, 2010

Abstract

CRAM, the Cognitive Robotic Abstract Machine, is a toolbox aimed at the design, implementation, and deployment of cognition-enabled mobile robots. Its core consists of the CRAM Plan Language (CPL) which is a new incarnation of the Reactive Plan Language (RPL) by McDermott et al. CPL is a modernized, truly Common Lisp based, compiled domain specific language to express robots' execution plans as concurrent reactive control programs with rich control structures for perception-guided control and failure detection, classification and handling. In particular, it features a multithreaded architecture to benefit from the concurrency available in today's multiprocessor systems. This bachelor thesis will cover the rationale, details of implementation, and outlook after a major revision of the underlying multithreaded architecture. Furthermore, it will contain an attempt at a coherent description of the language, including usage examples and additional tools that may be useful to its user.

1 Introduction

CRAM (Cognitive Robot Abstract Machine) is a set of distinct, yet highly interconnected tools that are specially designed to enable robots to perform mobile manipulation and cognition-enabled control, for example to empower mobile agents to perform everyday manipulation tasks in human environments like a kitchen. See fig.1 for one of the current research objects making use of CRAM in the laboratories of the Intelligent Autonomous Systems Group at the Technical University of Munich.

Figure 1: "Rosie"

CRAM sits on top of the ROS middle-ware which provides a distributed framework of nodes that are interconnected to share services with each other. Fig.2 depicts the architecture employed in CRAM. All its building blocks are implemented inside ROS

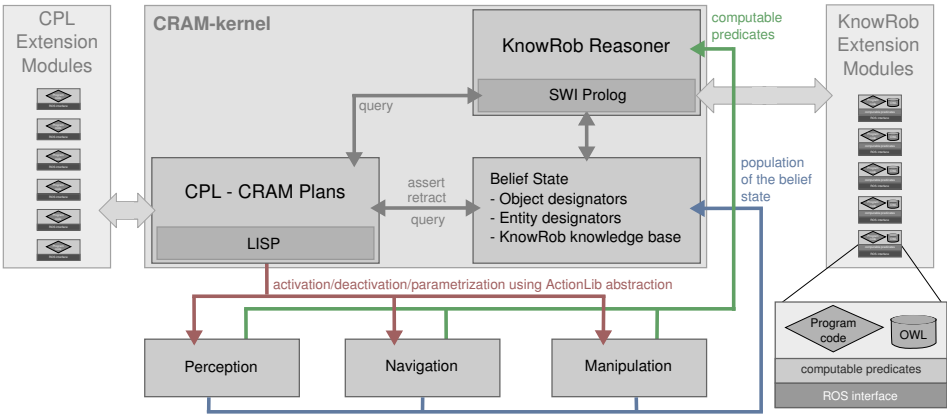

Figure 2: Architecture of CRAM

nodes, and it hence benefits from the modular and extensible architecture of ROS. As can be seen in fig.2, the core of CRAM is very lightweight; it consists of the CRAM Plan Language and interfaces to provide plans access to the belief state, to perception, to navigation, and to manipulation facilities like the arms of "Rosie" (fig.1). In detail:

CRAM Plan Language (CPL)

CPL is very expressive behaviour specification language that unifies control programs to be executed and plans to be reasoned about into the same language – essentially by allowing the programmer to annotate his programs with reasoning information. This unification allows to infer semantics of the code being executed and to use this information, coupled with other resources, to verify that a robot achieved all its goals, to detect potential flaws in the execution plan by means of projection, and possibly to repair it based on the projected outcome.

The Cram Plan language can be seen as a successor, or a modern reimplementation, of the Reactive Plan Language (RPL).[2] Like RPL, it is based on the idea of semantically-annotated plans to support high-level reasoning and plan transformation. Furthermore, its implementation is based on RPL concepts like tasks, and fluents. However, RPL was implemented as an interpreted, multitasking-emulating language on top of McDermott's dialect NISP.[3] Contrarily, CPL is implemented as a compiled, natively-multithreaded language on top of Common Lisp.

CPL itself consists of three parts:

1. Tasks, which represent threads of execution in a plan to, ultimatively, achieve goals like "navigate to position (x, y)"; tasks are organized in trees as it's common that a task requires additional sub tasks for its accomplishment.
2. Fluents (and fluent-nets), which act as a synchronization abstraction for convenient inter-task notification on value changes – an essential ingredient of autonomous agents, considering their need to react on exogenous events.
3. Control structures of the language, which are implemented on top of tasks and fluents.

CRAM Designators

Designators are symbolic descriptions of entities such as objects (mugs, plates, ...), locations or parameterizations of actions. Designators unify symbolic and grounded concepts in a plan with the parameterization of the lower level components, which is necessary to efficiently reason about the execution of plans.[4]

CRAM Process Modules

Process Modules provide interfaces to lower-level control processes that can be activated, deactivated and parameterized by the high-level control program. They resolve symbolic properties of designators and generate the parameterization of the low-level control routines, by taking the current belief state into account.[4]

CRAM Execution Trace

The execution trace records internal state of the control program and the belief state at any point in time. The information is useful not only for debuggability but also provides the mechanisms for complex monitoring and failure handling that is not based on local failures and exception handling but on the expressive formulation of error patterns in first order logics.[4]

CRAM Reasoning

The reasoning component includes a bridge between CPL and KnowRob[10], an extended knowledge base, by incorporating the foreign language interface of SWI-Prolog into Common Lisp. It further includes a reasoning component based on the RETE algorithm and a library of predicates that allow for reasoning about plan execution, based on the execution trace.[4]

This bachelor thesis will discuss implementation issues, and document design decisions of the CRAM Plan Language. It purports to serve the following two purposes: a) it shall assist the author's superviser in wrapping his head around the new implementation, and, more importantly, it shall act as guide for further development; b) it shall be passed on to fellow students and collegues to help them get started on CRAM.

2 CPL – The Implementation

As mentioned in the introduction, the CRAM Plan Language is implemented on top of two concepts called tasks and fluents. In the next section we will first provide an overview of these two concepts, and how their combination leads to a domain specific language that expresses execution as concurrent reactive plans. In the sections following, we will then discuss a major revision of the multithreaded architecture underlying tasks. In particular, we will argue why a revision was needed and we will illustrate problems and pitfalls encountered during the reimplementation. Finally, we will provide insights on the remaining work to be done.

2.1 Concepts

We will start with a brief description of what fluents are. Although they play only a marginal role with respect to this thesis, they still represent an essential part of the system.

A fluent contains a value. Tasks can wait on a fluent, and are notified[1] in case the value changes (see fig.3). More interestingly, multiple fluents can be combined in what we call a fluent network: essentially, a fluent bases its value on the value of another fluent and subscribes to that other fluent. If the other fluent changes its value, this change will propagate to the subscribed fluent whose value will then be recalculated (again, see fig.3).
As robots have to react on exogenous events (ideally in an intelligent way), an important use of fluents is to map such events into the system. However, fluents are actually used throughout the system – everywhere where values change over time and where a change of value may lead to change in behaviour.

Plans are control programs that are annotated with enough semantic information that they can be reasoned about. As this thesis does not discuss the high-level and reasoning aspects of CRAM, we can regard plans simply as descriptions of behaviour. Internally, they try to achieve some goal by splitting the achievement into tasks. Conversely, a task represents a thread of execution in a plan. In fact, tasks are precisely that: threads plus appropriate metadata. As tasks are often partitioned again into subordinated tasks, tasks are naturally

[1] Both edge-triggered and level-triggered notifications are supported.

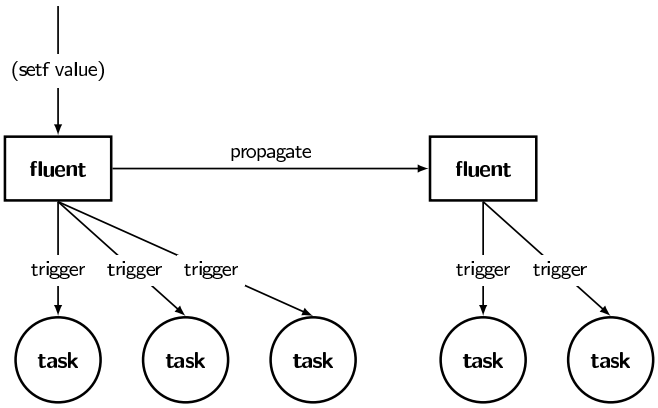

Figure 3: Fluents, schema.

organized in trees, or rather hierarchies[2]: a task is in control of its sub tasks; it determines which of its sub tasks are crucial (and must be accomplished for the accomplishment of the task at a whole), it determines how to react on failures of sub tasks, or what to do with sub tasks which became superfluous.

For purpose of this thesis (again we're neglecting semantic annotation), the CRAM Plan Language can now be seen as a language which allows to specify task hierarchies conveniently and concisely. In particular, it specifies the relations between a task, its sub tasks, and among the sub tasks. Each CPL control structure (which will be described in detail in section 3.1.3) represents a particular variant of such a relation. And by combining the control structures, a user can thus achieve complex task hierarchies with complex inter-dependent behaviour. CPL includes variants such as:

- All sub tasks should be accomplished.
- It's enough if one of the sub tasks is accomplished.
- Sub tasks must be accomplished according to a particular partial order.
- Suspend some of the sub tasks, do something else, resume tasks.

And many more. To enforce such a relation, a task subscribes to its sub tasks' status fluent which contains the state a task is in. Thus if one of the sub tasks changes its status (e.g. to indicate it's accomplished), the subscribed parent will be notified and can proceed as desired – for instance, to wait again on the fluent until all other sub tasks are accomplished, too, or to shut down remaining sub tasks and change its own status to accomplished.

[2]To describe the relation between a task and its subordinated task we sometimes use the terms "task" and "sub task", and other times "parent task" and "child task". The former expresses their hierarchical role, and the latter expresses their organization as data structure.

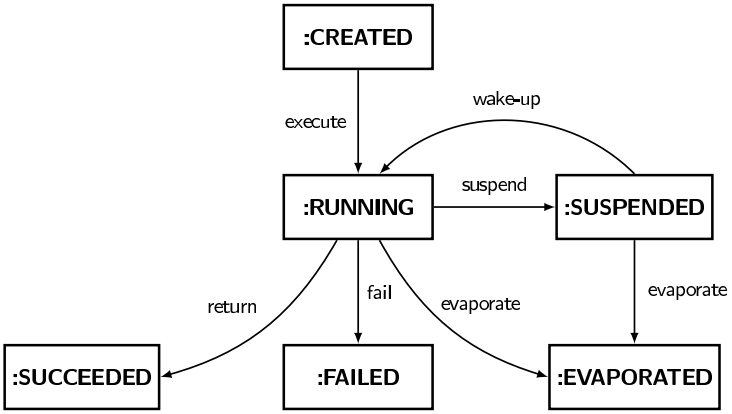

Figure 4: A task's state diagram

As can be seen, getting tasks right is all about getting inter-task communication right. And since tasks are executed in threads, it's actually about getting inter-thread communication right – with the caveat that this communication should mostly be performed implicitly without user's direct notice.

Inter-task communication is tied to state transitions in tasks. This makes sense as we said that a task is supposed to control its sub tasks, and inter-task communication is the means for a task to induce state transitions in its sub tasks. Fig.4 illustrates the interplay of states in a task. Notice that there are three exit states: `:succeeded` means the task was accomplished, `:failed` means the task could not be accomplished, and `:evaporated` means the task became superfluous and was killed by its parent.

2.2 Old Task Implementation

As described previously, the key part in implementing the CRAM Plan Language is the intercommunication between tasks belonging to the same hierarchy. The old implementation used the function `interrupt-thread` for that purpose, a common extension provided by multithreaded Common Lisp implementations. For example, a parent task would have executed the following to evaporate its children:

```
(dolist (child (child-tasks *current-task*))
  (interrupt-thread child #'(lambda ()
                              (signal 'termination))))
```

`interrupt-thread` interrupts the passed thread and makes it execute the passed function instead of what it was executing before – in this case, to signal a condition which indicates that the child is supposed to be torn down due to evap-

oration. A suspension was requested analogously but by a condition named `suspension`. Each task was executed in a context where appropriate handlers for `termination` and `suspension` were active. This implementation technique is neat in so far as it only involves minimal code, and reuses the ingredients of the Common Lisp maximally.

Albeit neat, this implementation technique is seriously flawed.

The problem is that `interrupt-thread` works by interrupting the passed thread totally asynchronously, at *whatever* point it is currently executing. Consequently, a task could have been evaporated, or suspended, at any time. And evaporation leads to unwinding, suspension leads to waiting – at any time. There are times, however, where an unexpected unwinding, or waiting, would result in rather dire consequences:

1. What if a a task is forced to unwind from amid a critical section?
   Likely, it will leave inconsistent state behind, for instance, in the object being modified in the critical section.

2. What if a task is forced to unwind while executing a cleanup clause of `unwind-protect`?
   At best, we'll leak some ressource. Or, again, inconsistent state.

3. What if a task is suspended within a critical section?
   The suspended task will continue to hold some lock. If another task wants to acclaim that lock, it won't be able to do so until the suspended task is resumed. But if the other task is the one responsible for waking up the suspended task, a dealock will occur.

As implementations usually provide some means to disable the receiving of interrupts for the duration of some forms, all of these issues can in principle be catered for. And there may even be ways to proactively help the user address these issues: For example, software transactional memory may be used to solve the first issue and prevent inconsistent state by rolling back state changes before unwind; `unwind-protect` could be implemented to disable interrupts while executing the cleanup forms to obliterate the second issue[3]; and for addressing the third issue, an abstraction could be provided that a) allows the user to specify safe places to unwind to on suspension, and b) how to rewind again on wake up – akin to Scheme's `dynamic-wind`.

However, all these mechanisms do not really change the crucial properties that seem to be inherent in asynchronous interrupts:

[3] There may be situations where a cleanup form is so complex and may involve so much time that it should in fact be interruptible. For that purpose, implementations should provide a means to locally reenable interrupts in an `unwind-protect` cleanup form. We argue, however, that it strikes us as the better policy to disable interrupts by default, and have the user enable them explicitly, rather than the other way around – if only to cope with users' mental model of `unwind-protect`.

- To reiterate as it's so important: asynchronous interrupts can happen *any* time, between and in-between *any* line of code.
- Interrupt-safe code does not compose. Even if a piece of code considered on its own deals with interrupts sensibly, does not necessarily mean that "sensible" transfers to all usages.[9]

Consequently, interrupt-safety cannot be localized into a small, heavily audited code area, but would have to be catered for across the whole source base; not only in user code, but also in the sources of the implementation, in third-party code, literally everywhere. Programming would require whole-world knowledge – those who have not ascended yet would have to program in angst and sorrow or, alternatively, in ignorance.[4]

Angst and sorrow is not mere rhetorics. As a case in point, we want to describe a puzzling phenomenon of the old implementation: when running the test suite, every now and then a frightning "compilation aborted" message was printed to the screen without any further information. It turned out that timing issues in one of the test cases spured an evaporation right during the computation of the discriminating function which typically involves compilation in a CLOS implementation. The evaporation caused unwinding out of the compilation which thus caused the message to be printed.

Another case in point. Let's revisit our previous code snippet:

```
(interrupt-thread thread #'(lambda ()
                             (signal 'termination)))
```

Notice that the above is tantamount[5] to the following Java code

```
thread.stop(new Termination());
```

`Thread.stop` (and `Thread.suspend` for that matter) were deprecated 10 years ago. In [1], the rationale behind that decision is explained; it can be summarized with one quote: "In sum, it just isn't practical."

2.3 New Task Implementation

We concluded the last section by comparing Common Lisp's `interupt-thread` to Java's `Thread.stop`. As Java also provides a `Thread.interrupt`, it begets the question how `Thread.interrupt` differs from `interrupt-thread` that it's not plagued by the same issues. In Java, an `InterruptException` is not thrown at arbitrary times, but only in predefined places. The user is supposed to extend

[4] The only real way out, it seems, would be to disable interrupts globally, or for vast areas of code. Of course, that would render the implementation technique being discussed pretty useless.

[5] We're neglecting the fundamental difference that signaling a condition in Common Lisp does not necessarily lead to stack unwinding.

this set of places appropriately. Thus interrupts in Java are not asynchronous.

In general, programmatic use of interrupts should be dealt with by whitelisting rather than blacklisting: define a set of places where interrupts are supposed to be triggered – enough to be adaquately responsive, not too many not to incur too much overhead. The obvious advantage is that the problem of having to deal with interrupts all over the place will be gone immediately. Interrupts will now be localized.

SBCL includes a feature called deadlines[6] which could be summarized as global (or thread-local) timeouts based on the aforementioned whitelisting approach. A deadline is an absolute time pointing into the future, and a couple of internal functions check whether current time exceeds the current deadline. If that's the case, they will signal a condition of type `deadline-timeout`. These functions include `get-mutex`, `condition-wait`, `wait-on-semaphore`, and `serve-event` (the recursive event loop used for I/O). An restart is associated with the condition `deadline-timeout` to defer the just expired deadline again into the future. On behalf of that restart, one can use deadlines to periodically execute a function and continue the original computation after that function has run. This is what the new task implementation is based on.

Indeed the new implementation works as follows:

1. Each task is executed with the deadline mechanism being activated.
2. Each task establishes a handler for `deadline-timeout` that invokes the task's event loop.
3. The event loop looks for new messages in a mailbox:

[6] Caveat: deadlines aren't actually exported from a public package, and are not available on all platforms (as of SBCL 1.0.38.)

- If no new message has arrived, the restart is invoked to continue the computation where the deadline was triggered. The restart also reinstalls the deadline so that we will enter the event loop again after a while.
- If a new message has arrived, it's processed. The message encodes another task's request for evaporation, suspension, or wakeup.
  - An evaporation request leads to tear down which includes: a) sending evaporation requests to the current task's children, b) changing status to `:evaporated`, and c) exiting.
  - A suspension request is first propagated to the current task's children, then the task simply stays in the event loop and waits on the mailbox for new messages.
  - Notice how the "current" continuation resides on the stack and is invokable through the restart during the time a task is suspended.
  - Hence a wake up means nothing more than invoking said restart (after the wake up was propagated to the current task's children, of course.)

The implementation can easily be illustrated with the following 60 lines of code:[7]

```
(defstruct (task (:constructor make-task (&key name function children)))
  name
  function
  children
  (status :created)
  (mailbox (sb-concurrency:make-mailbox)))

(defvar *current-task* nil)

(defconstant +period+ 0.01)

(defun change-status (task new-status)
  (setf (task-status task) new-status))

(defun execute (task)
  (sb-thread:make-thread
   #'(lambda ()
       (let ((*current-task* task))
         (change-status task :running)
         (call-with-event-loop task (task-function task))))))

(defun call-with-event-loop (task thunk)
  (handler-bind ((sb-sys:deadline-timeout
                  #'(lambda (c)
```

[7] The code requires an SBCL version of 1.0.38 or higher.

```
                        (event-loop task
                                    #'(lambda ()
                                        (sb-sys:defer-deadline +period+ c))
                                    #'(lambda ()
                                        (return-from call-with-event-loop))))))
    (sb-sys:with-deadline (:seconds +period+)
      (unwind-protect (funcall thunk)
        (propagate-message task :evaporate)))))

(defun event-loop (task continue finish)
  (let ((mailbox (task-mailbox task))
        (waitp nil))
    (loop
      (multiple-value-bind (message received?)
          (if waitp
              (values (sb-concurrency:receive-message mailbox) t)
              (sb-concurrency:receive-message-no-hang mailbox))
        (when received?
          (process-message task message))
        (ecase (task-status task)
          (:running    (funcall continue))
          (:suspended  (setq waitp t))
          (:evaporated (funcall finish)))))))

(defun process-message (task message)
  (propagate-message task message)
  (ecase message
    (:suspend   (change-status task :suspended))
    (:wakeup    (change-status task :running))
    (:evaporate (change-status task :evaporated))))

(defun propagate-message (task message)
  (dolist (child (task-children task))
    (sb-concurrency:send-message (task-mailbox child) message)))
```

Notice that we haven't addressed failure handling neither in the description nor in the exemplary implementation. The reason is that failures aren't actually implemented by messages, but implicitly through `join-task` (see section 3.1.1).

Further notice that the exemplary implementation does not address the problem of suspending in mid of a critical section. For example, a deadline signaled in `condition-wait` may invoke a task's event loop while holding the lock associated with the relevant condition variable. And, as fluents are implemented on top of condition variables, getting this issue right was a requirement right from the beginning. It was one of the aspects in the real implementation which was

a tad bit tricky to implement correctly. Let's sketch how we did it:

1. We provide a macro `retry-after-suspension` which is supposed to unwind out of a critical section right before suspending and to re-execute the critical section on wake up. Essentially it's an adaption of the `dynamic-wind` idea from page 7.

2. `retry-after-suspension` registers a callback which is invoked when the event loop processes a suspension request.

3. This callback performs three things in the following order:

   (a) It unwinds out of the body of `retry-after-suspension`.

   (b) It reestablishes necessary context which was lost due to the unwind; e.g. it reestablishes the deadline and makes up the deadline's associated restart.

   (c) It continues the suspension process so we will eventually end up back in the event loop waiting on the mailbox.

Excursion

While trying to implement retry-after-suspension for the first time, we were majorly screwed by an ideosyncracy of Common Lisp's condition system. Kent Pitman, author of the chapter about the condition system in the Common Lisp ANSI standard, writes in [6]:

> Probably the most controversial semantic component of the Common Lisp condition system is what has come to be called the "condition firewall". The idea behind the condition firewall is that a given handler should be executed in an environment that does not "see" intervening handlers that might have been established since its establishment.

And, indeed, the ANSI standard is very explicit on the issue:[7]

> Handlers are invoked in a dynamic environment equivalent to that of the signaler, except that the set of active handlers is bound in such a way as to include only those that were active at the time the handler being invoked was established.

Let's try to illustrate the exact meaning and its consequences on behalf of an example:

1.
```
(handler-case
    (handler-bind ((warning
                    #'(lambda (c)
                        (declare (ignore c))
                        (error "Got a warning! Turned it into an error."))))
      (warn "A warning:"))
  (error (e)
    (declare (ignore e))
    (format t "Caught an error.~%")))
```

2.
```
(handler-bind ((warning
                #'(lambda (c)
                    (declare (ignore c))
                    (error "Got a warning! Turned it into an error."))))
  (handler-case
      (warn "A warning.")
    (error (e)
      (declare (ignore e))
      (format t "Caught an error.~%" c c))))
```

In the first form, the warning is first turned into an error, the error is caught by the `handler-case` and thus only a message is printed.
In the second form, however, the handler established by `handler-case` is actually out of scope when executing the handler established by `handler-bind` – so the error will propagate through and cause an invocation of the debugger.

The specified behaviour makes sense as it ensures that the scope of handlers with respect to each other is lexical: a handler X that is established textually before Y, will will only see those handlers that appeared textually before X. This is an important property as the handler X may in fact not know about the exact state of affairs down its line. (As a special case, consider how easily one could be tricked into endless recursion between a `handler-bind` and a nested `handler-case` if the "condition firewall" were not in place.)

So what's the matter?

Let's recall that we enter the event loop through a deadline handler. And this deadline handler is the outermost handler within a task's execution as can be witnessed from the exemplary implementation above. So a consequence of the "condition firewall" is that we cannot reuse Common Lisp's condition system to communicate between the event loop and implementation details (e.g. set up by macros like `retry-after-suspension`) down the stack. Any handler established down the stack will be out of scope in the event loop.

This property of Common Lisp's condition system puts a pretty harsh restriction on using deadlines as a way to invoke an event loop periodically. Although we see the new task implementation as described in this section as a major

improvement over the previous implementation, we will propose a slightly different implementation strategy in the next section which will also overcome the "condition firewall."

2.4 Future Work

To recapitulate: In section 2.2, we argued how problematic the previous implementation was due to its use of asynchronous interrupts. We argued that trying to tame asynchronous interrupts resembles the duty of Sysiphus because to reason about them would require knowledge about the whole software stack. We compared the old implementation technique to Java's `Thread.stop` which has long been deprecated for these reasons. In section 2.3, we argued that programmatic use of interrupts (like we intend to do) should be based on whitelisting rather than blacklisting. We provided Java's interrupt mechanism as an example for how it's supposed to be done. Further, we proposed SBCL's deadline mechanism to be such a mechanism. This begets the question now in how far exactly deadlines compare to Java interruptions.

Both, deadline and Java interrupts, are raised in a handful of predefined places – in fact, almost the same places. However, Java's `InterruptException` is a checked exception, so it's an explicit and permanent part of the places' API (even enforced by the compiler). The crux is that handling `InterruptException` is not only enforced for direct callers but for all callers. It follows that a Java programmer knows precisely where in a system interrupts can occur. The same cannot be said on the Lisp side: a deadline can, for example, expire within `get-mutex` no matter what mutex was passed to it. Consequently, a `deadline-timeout` can actually be signaled from within *any* function that makes use of synchronization internally (or uses one which does).

And these internal details really do leak. Case in point: a semaphore is implemented[8] on top of mutexes and condition variables. So a `deadline-timeout` may be signaled within `wait-on-semaphore` while holding the mutex that actually is completely internal to the semaphore. If we enter the event loop under such circumstances, and process a suspension request, we just rendered the semaphore unusable – and danger awaits. Notice that our hands as user are tied on the issue; we cannot just modify the implementation of semaphores to use our `retry-after-suspension`. It follows that any direct, or indirect use of semaphores is liable to lead to problems.[9] Any dream of reasonability is gone.

Notice that in the old task implementation, the processing of requests could

[8] As of SBCL 1.0.38.

[9] We were bitten by it: spawning a thread, thus executing a task, uses a semaphore internally. As the semaphore is purely internal, no deadlock was warranted. However, problems arose nontheless because a parent task could enter its event loop before its child was properly set up. We solved this issue by locally disabling deadlines using `without-scheduling` (see section 3.1.1 for a description.)

happen anywhere. In the new task implementation, it can only happen where synchronization or I/O is directly or indirectly used. So it's definitively a step forward. However, for the reasons of unpredictability (and for the problems due to the "condition firewall" discussed in the previous section), we actually dimiss deadlines as a viable implementation strategy for our purpose.[10]

We will now conclude our discussion of the implementation with outlining an approach which differs only slightly but to great effect:

- We still want to periodically enter a task's event loop where requests are received and acted upon.
- However, instead of using deadlines provided by SBCL for that purpose, we propose to essentially implement our own deadline mechanism (which will then be used in our code only):
  - We subclass, or write wrappers as appropriately, for locks, condition variables, semaphores. These wrappers check whether our deadline expired.
  - We check for expiration in appropriate places like waiting on a fluent.
  - If necessary, we add loop abstractions for the user which internally check the deadline. (So the user does not have to be bothered with checking for expiration himself.) As plans usually involve fluents en masse, we suppose it's a rather rare circumstance.
  - If our deadline expires, we won't signal a condition, but either directly invoke the event loop, or go over the indirection of a special variable.
- As we implement deadlines ourselves, the now current architecture can mostly be kept just as is.

The advantages of the outlined approach include:

1. We can make use of `retry-after-suspension` in the wrappers appropriately.
2. Since it's all wrapped up and the deadline mechanism is provided by ourselves, our code can become portable across implementations again.
3. The event loop will only be triggered by our own code, so reasoning won't require whole-world knowledge.
4. We circumvent the problems induced by the "condition firewall".

[10] To be precise: We dismiss the implementation of deadlines as of Sbcl 1.0.38.

3 CPL - A User's Guide

The following section is probably the section most interesting to future users of CRAM and CPL. A note of caveat to these users: the following descriptions should provide a good head start, but CRAM was still in quite an early stage of existence as of the writing of this thesis. We're pretty certain, though, that the essential bulk of behaviour will stay as described here, details may differ slightly, however.

3.1 The Language[11]

3.1.1 Tasks

[Variable] ***current-task***

Inside a task, contains the task object being executed. Otherwise, `nil`.

[Classes] **abstract-task**, **toplevel-task**, **task**

A `toplevel-task` represents the root of a hierarchy of `task`s.

[Function] **make-task** &key name function

Create an object of class `task` named *name*. If executed, it'll execute *function*, a thunk.

[Function] **execute** task

Spawn a thread, and have it execute the function associated with *task*. This induces a status transition from `:created` to `:running`.

[Accessor] **child-tasks** task

Return all the children of *task*.

[Accessor] **status** task

Returns a fluent whose value is the current status of *task*.

Example Being a fluent, one can easily wait for status transitions of a task. The following will wait until a task, which was just created, will actually excute, or until a task, which was suspended, will execute again:

```
(wait-for (fl-eq (status task) :running))
```

[11] This section is mostly a reference. For a description on the concepts involved, please refer to section 2.1.

[Type] **status-indicator**

One of

- `:created`,
- `:running`,
- `:suspended`,
- `:succeeded`,
- `:evaporated`, or
- `:failed`.

[Constants] +**dead**+, +**done**+, +**alive**+

A task is...

dead	if it either succeeded, failed, or was evaporated.
done	if it either succeeded, or was evaporated.
alive	if it either has not run yet, is currently running, or is suspended.

[Accessor] **result** task

Return the result of the *task*. For multiple values, this function also returns multiple values. Setfable.

[Function] **join-task** task

Wait until *task* has finished.
When *task* fails, this function resignals the failure condition.
When *task* succeeds, this function returns its result values as multiple values.

[Function] **suspend** task &key reason sync

Sends a suspension request to *task*. This request will be propagated to all its children (and then to their children, and so on.)

For debugging purposes, *reason* is a string which will displayed as part of the logging output described in section 3.2.2.

By default, the current task (that is the caller of `suspend`) won't wait until *task* really becomes suspended. If *sync* is true, the current task *will* wait until *task* and all its children are suspended.

Example Notice the difference between

```
(progn (suspend task)
       (wait-for (fl-eq (status task) :suspended)))
```

and

```
(suspend task :sync t)
```

is that the former will only wait until *task* becomes suspended, while the latter will wait until *task* and all its children become suspended.

[Function] **wake-up** task &key reason sync

Sends a request to *task* to wake up from an earlier suspension request. The request will be propagated appropriately. See `suspend` for detailed information.

[Function] **evaporate** task &key reason sync

Sends an evaporation request to *task*. The request will be propagated appropriately. See `suspend` for a discussion of the parameters *reason*, and *sync*.

[Macro] **without-scheduling** &body body

Execute *body* without entering the current task's event loop periodically. This inhibits any acting on inter-task events like suspension, or even evaporation. `without-scheduling` should hence be used with care, and only for small periods of time.

[Macro] **on-suspension** when-suspended-form &body body

During the execution of *body*, execute *when-suspended-form* whenever the current task is going into suspension.

The following restrictions apply:

1. The dynamic environment in which *when-suspended-form* will be executed, is undefined. Except:
2. *when-suspended-form* is executed with scheduling disabled.
3. Non local exits out of *when-suspended-form* are prohibited.

Example The common scenario for needing `on-suspension` is when an action is performed that is going to take a while, like navigating to some coordinates. Before the task falls into sleep, it has to make sure to stop the robot's gears:

```
(on-suspension (stop)
  (move-to-point x y))
```

[Macro] **retry-after-suspension** &body body

Execute *body*, and return its values.

In case the current task is suspended during the execution, *body* will be unwound completely, and the suspension will take place outside of *body*. After wake up, *body* will be executed again.

Example The intended use case is to unwind out of critical sections to ensure that a task won't suspend while holding on some lock:

```
(retry-after-suspension
  (with-lock *lock*
    ...
    (condition-wait *condvar* *lock*)
    ...))
```

Notice how the design of condition variables make them particularly prone to this kind of scenario. Further notice that using `without-scheduling` is not an option as it would make the task completely unresponsive during the wait.

[Macro] **define-task-variable** name &optional global-value docstring
&key type init-function

Define a binding for *name* as a special variable, and in particular as a task variable. *global-value* is the variable's global value; if not given, the variable will be unbound. *docstring* is the variable's associated documentation string. *type* is, optionally, its globally proclaimed type.

A task variable is a binding that is established and initialized right before execution of a task. The binding is task-local, meaning that assignment to it won't affect the variable's global binding, or bindings within other tasks.

The initialization is performed by *init-function* which must be a function of three parameters: the task object going to be executed, the task object of the parent task, and the previous value of the variable *name* in the parent. In case the first argument is a `toplevel-task`, the second argument will be `nil`, and the third one will be the global value.

Task variables can hence be used for different purposes:

1. to initialize a variable based on the task object to be executed.
2. to initialize a variable based on the task's parent.
3. to inherit a task variable's value from its parent.

In case *init-function* is not given, the default behaviour is to inherit the value from the parent task.

Example In section 3.2.2, a logging facility will be described which prints timestamps relative to the start time of the containing `toplevel-task`. Obviously, for that to work out, we have to store that start time somewhere where all sub tasks (and only they) will see it. Notice how Lisp special variables cannot be used to solve this problem.

```
(define-task-variable *zero-real-time* nil
  "Base timestamp to compute relative real-time timestamps from.
   This variable gets its value from the top-level task, all
   sub tasks will just inherit it."
  :type (or null timestamp)
  :init-function
  #'(lambda (task parent-task previous-value)
      (declare (ignore task parent-task))
      (or previous-value (current-timestamp :real))))
```

3.1.2 Fluents

[Classes] **fluent**, **value-fluent**, **pulse-fluent**, **fluent-net**

Fluents come in three flavors, `value-fluent`, `pulse-fluent` and `fluent-net`.

`value-fluent`
: represents a fluent with an associated value which can a) be changed, and b) be waited for. A `value-fluent` is what is commonly stored into an object's slots for values whose changes are supposed to be tracked – just like the `status` slot in task objects.

`pulse-fluent`
: represents a special kind of fluent without an explicitly associated value (it only ever is `nil`, or `t`) that is used in special wait scenarios. In particular, it overcomes the ABA problems inherent in waiting on a `value-fluent`. See `fl-pulsed` for details.

`fluent-net`
: represents an interconnected network of fluents where a value change of one fluent propagates through the network by possibly causing neighbors to change their values, too. A `fluent-net` is commonly used to track specific aspects of fluent changes, and, as the examples below will show, they make it very convenient to work with fluents in general.

[Function] **make-fluent** &key name value

Create an object of class `value-fluent` named *name* with an initial value of *value*. No explicit constructors are provided for `pulse-fluent`, and `fluent-net`.

[Accessor] **value** fluent

Return the value of *fluent*.

[Accessor] **(setf value)** new-value fluent

Sets the value of *fluent* to *new-value*. Implicitly pulses the fluent.

[Function] **pulse** fluent

Trigger *fluent*.

- Tasks waiting on *fluent* are notified that the *fluent*'s value has possibly changed; these waiters will consequently wake up if the fluent contains a non-`nil` value now.
- fluent-nets built on top of *fluent* are notified to recalculate their values and propagate the pulse through the net. This, of course, can then lead to even more tasks to wake up.

[Function] **wait-for** fluent &key timeout

Block the current task until the value of *fluent* becomes non-`nil`. If *timeout* is specified, waits for at most *timeout* seconds. In case fluent becomes non-`nil`, `t` is returned. In case *timeout* expired, `nil` is returned.

[Macro] **whenever** (fluent) &body body

Whenever *fluent* becomes non-`nil`, execute *body* in an implicit block named `nil`. If *fluent* is of type `value-fluent`, and is changed multiple times during the execution of *body*, only the last change will count in the next iteration. See `fl-pulsed` for details about this kind of situation.

Example

```
(whenever ((fl-eq (status *task*) :suspended))
  (format t "Task ~S was suspended.~%" *task*))
```

[Function] **fl-funcall** function &rest arguments

Return a fluent of class `fluent-net`. The value of the returned fluent is the result of invoking *function* on *arguments* in the following way:

- if an argument in *arguments* is a non-fluent value, it is used as is.
- if an argument is a fluent object, that fluent's value will be used.

In particular, each time a fluent in *arguments* changes its value, the value of the returned fluent is recalculated (again by invoking *function* as above).

If no argument in *arguments* is a fluent object, `fl-funcall` will behave like `funcall`.

Example Another way to view `fl-funcall` is that it lifts functions operating on normal Lisp values into the world of fluents. So

```
(defparameter *fluent* (make-fluent :value 0))
(fl-funcall #'+ *fluent* 42)
```

will return a fluent whose current value is 42, and which is incremented by 42 each time `*fluent*` is updated.

[Function] **fl-apply** function &rest args

Return a fluent of class `fluent-net`. The value of the returned fluent is the result of applying *function* on *arguments*. See also `fl-funcall`.

Example Here we construct a fluent which will only become true if all the current task's children succeed.

```
(fl-apply #'every (curry #'eq :succeeded)
          (mapcar #'status (child-tasks *current-task*)))
```

[Function] **fl-pulsed** fluent &key (handle-missed-pulses :once)

Return a fluent of class `pulse-fluent` whose value becomes true "every" time *fluent* is pulsed. What "every" actually means is tricky in lieu of missed pulses, and thus the behaviour is customizable via *handle-missed-pulses* which can be either `:never`, `:once`, or `:always`.

Missed pulses mean pulses that occur while the current task is processing an older pulse. For instance:

```
(loop (wait-for *fluent*) (frob))
```

If `*fluent*` is changed more than twice during the excution of `(frob)`, only the last change will be what matters to the `wait-for` in the next iteration. The in-between changes will be lost. For changes, it usually does not matter as the current value is what counts. However, there are situations where the number of pulses (or changes) may be given meaning – and it would be rather bad if in-between pulses were lost.

One can think of the following three way to handle in-between pulses: either ignore them, do not ignore them but handle a bunch of in-between pulses as one big pulse, or handle each of the in-between pulses.

And, in fact, fl-pulsed can be used for each of these scenarios. Its behaviour depends on the value of *handle-missed-pulses* as follows:

`:never`
: Initial value of the resulting fluent is `nil`. Its value becomes `t` if *fluent* is pulsed, no matter how many times. A call to `value` on the resulting fluent will make it `nil` again.

`:once`
: Initial value is `nil` if *fluent* has not been pulsed since the last invocation of value, and `t` if it has been pulsed. Like `:never`, its value becomes `t` if *fluent* is pulsed, no matter how many times, and it becomes `nil` again on call to `value`.

`:always`
: All pulses will be handled. Invoking `value` on the resulting fluent will return true as many times as there has been pulses on *fluent*.

Caveat Note that to implement the stated behaviour, a `pulse-fluent` must track the number of pulses that *fluent* received, and how many times `value` was invoked on the `pulse-fluent`. Consequently, a `pulse-fluent` is inherently not reentrant, and it should hence be used within the extent of *one* task only.

[Function] **fl-value-changed** fluent &key test key

Return a fluent of class `fluent-net` that is pulsed whenever *fluent* changes its value. The change is determined by invoking *test* (defaulting to `eql`) to the current and the old value.

Example It is important to recall the semantics of `wait-for` which is specified to wait until a fluent's value becomes non-`nil` – not necessary until it's changed:

```
(loop (wait-for *fluent*) (frob))
```

will execute `(frob)` as long as the value of `*fluent*` is not `nil` even if the value stays the same. To the contrary,

```
(loop (wait-for (fl-value-changed *fluent*))
      (frob))
```

will invoke `(frob)` only each time `*fluent*` changes its value as per `eql`.

[Functions] **fl-and**, **fl-or**, **fl-not**

Like `and`, `or` and `not` but working on fluents. Notice that every one of these is defined as a function and can hence be applied to a list of fluents.

[Functions] **fl**$<$, **fl**$>$, **fl**$=$, **fl**$+$, **fl-**, **fl***, **fl**$/$, **fl-eq**, **fl-eql**, **fl-member**

Wrapper functions. These can be thought of as being defined using `fl-funcall` and their respective Common Lisp counterpart.

Displayed are the names in the `cram-language-implementation` (`cpl-impl`) package. In the `cram-language` (`cpl`) package, their Common Lisp counterparts are actually shadowed, and these functions are used instead.

3.1.3 CPL Control Structures

[CPL Operator] **toplevel** &body body

Create a new `toplevel-task`, execute *body* in it and waits until it has finished as per `join-task`.
All CPL operators can only be used within the dynamic scope of `top-level`.

[CPL Operator] **seq** &body forms

Execute each form in *forms* sequentially.
Succeed if all succeed.
Fail as soon as one fails.

[CPL Operator] **par** &body forms

Execute each form in *forms* in parallel.
Succeed if all succeed.
Fail as soon as one fails.

[CPL Operator] **pursue** &body forms

Execute each form in *forms* in parallel.
Succeed as soon as one succeeds.
Fail as soon as one fails.

[CPL Operator] **try-all** &body forms

Execute each form in *forms* in parallel.
Succeed as soon as one succeeds.
Fail only if all fail.

[CPL Operator] **try-in-order** &body forms

Execute each form in *forms* sequentially.
Succeed as soon as one succeeds.
Fail only if all fail.

[CPL Operator] **with-tags** &body body

While executing *body*, all lexically occuring `(:tag` name `.` forms`)` will be processed as follows:

- each (:tag ...) form will be replaced by a task named *name* which executes *forms.*
- each such task object will be bound to a lexical variable named *name* within *body.*

`with-tags` can hence be used to give names to arbitrary forms in *body*, associate tasks to these forms, and provide access to these tasks.

[CPL Operator] **partial-order** steps &body orderings

Execute *steps* in an implicit `par` form. *orderings* constrain the order these implicitly created tasks will be run under:

```
(with-tags
  (partial-order ((:tag A ...)
                  (:tag B ...)
                  (:tag C ...))
    (:order C B)
    (:order B A)))
```

In this case, we specify that B is constrained by C (i.e. has to wait until C finished), and A is constrainted by B. It follows that the tasks will actually be executed in the order C, B, A.

[CPL Operator] **with-task-suspended** (task &key reason) &body body

Suspend *task*, execute *body*, and wake *task* up again.

Usage Example

We will now give a contrived example that purports to help on conceptualizing the semantics of the control structures. It is contrived for two reasons: a) it is incomplete, almost pseudo-code, and the devil is always in the details; b) it consists of the CPL control structures, only, and does not make good use of higher-level concepts such as goals, designators, or process-modules.

```
(with-tags
  (pursue
    (:tag nav (navigate x y))
    (whenever ((fl> (sensor-fluent :front) *threshold*))
      (with-task-suspended (nav :reason "Object located")
        (circumvent-obstacle (analyze-obstacle))))))
```

Two tasks are started concurrently: the first one directs the robot to go to some coordinates. And while the robot is walking towards that location, the second one will watch out for obstacles. It determines that something is in the robot's way by querying one of it's sensor. If the sensor values peak, we must have detected something. In that case we suspend the navigation task so not to run into the obstacle. It then tries to find out what kind of object it is that's in the way, and instructs the robot to circumvent it. After we successfully circumvented the object, the nagivation task will be woken up to continue to bring us to the robot's target. (We assume the robot has a world map so the (x, y) coordinates were absolute.) Once we arrived at the target location, the navigation task will succeed, and hence the `pursue` will succeed, too.

The sub plan `analyze-obstacle` could be defined to look something like the following:

```
(try-all
  (find-object-using :laser)
  (find-object-using :camera))
```

Which will, again, start two tasks in parallel: one which tries to determine the object via the laser, and the other via the camera. Notice that we're only interested in one result (namely the first) and do not give priority to any method.

And the sub plan `circumvent-obstacle` could include something like:

```
(try-in-order
  (bypass designator :right)
  (bypass designator :left))
```

It's similiar to above, but we first try to bypass the obstacle from the right side, then from the left side – parallel execution would not make sense here.

3.2 Usage & Debugging Aids

3.2.1 Test Infrastructure

Overview

An essential part of the Lisp experience is the interactive writing, testing and incremental refining of one's programs. Naturally, the same applies to CPL. However, especially during the revision of the task implementation, a proper teardown of all the tasks involved in a CPL expression could not be guaranteed. Rather quickly, it became tedious to kill remaining "ghost" tasks manually. That's why the existing test infrastructure was significantly improved early on. So, for example, the infrastructure was modified to check, after a test has executed, whether all tasks spawned during the execution of the test are dead or still alive; in the latter case, they will be killed and what would have been a success previously, is turned into a test failure. Test definitions are supposed to ensure that they shutdown all the tasks they spawn.
Still now – where proper teardown should be assured – running one's program wrapped into a test case has a couple of advantages:[12]

- hangs, or deadlocks, are prevented by a timeout associated with a test.
- unhandled errors are caught, their backtraces are saved, and then the errors are turned into plan failures. The debugger is *not* automatically entered by default: multiple threads may easily lead to multiple entrances

[12]Some of these features should probably be decoupled from the test infrastructure and be provided to the user so he can make use of them without having to actually write tests (which may be regarded as unwiedly due to requirements on naming, and documentation.) For instance, in a macro called `interactive-top-level`.

into the debugger, often separated by a small delay – annoying if the development environment hijacks focus on entering the debugger. Instead, backtraces can be displayed as part of the test failure reports.

- the `cram-language-tests` (`cpl-tests`) package contains a number of utility functions and macros to significantly ease up the establishing of complex task-related state, and to check properties of such state.
- tests can automatically be run multiple times in a row, coupled with a QuickCheck like facility to generate random data.

Implementation notes

The test infrastructure is currently built on top of 5am[13], although that choice may be reconsidered in future.

Synopsis[14]

[13] `http://common-lisp.net/project/bese/FiveAM.html`

[14] All symbols mentioned are accessible in the `cram-language-tests` (`cpl-tests`) package.

[Macro] **define-cram-test** name docstring (option*) &body body

Define a new test named by *name*. For the time being, a test can be executed via `(run! 'name)`.

During the execution of *body*,

- a timeout is associated with the execution such that hangs and deadlocks will be caught and turned into test failures;
- all tasks spawned will be hold onto such that on test failure or on timeouts, the threads executing these tasks will be killed;
- unhandled errors and unhandled plan failures in any of the tasks executed will be handled as follows:
  - A log entry will be emitted, tagged `:debugger`. (See section 3.2.2)
  - The backtrace will be remembered. However, backtraces will only be displayed if `*include-backtrace*` is true. As backtraces tend to be long, they would make test failure reporting unnecessarily wordy in the common case of a simple test failure due to a test form evaluating to `nil`.
  - If it's a condition of type `plan-failure`, the failure will be propagated upwards, and child tasks will be evaporated.
  - If it's a condition of type `error`, it will be turned into a plan failure. Usually, like above, that failure will propagate upwards until it reaches the caller of `top-level`, that is the thread executing the test as such, and hence cause a test failure.
  - One can force entrance into the debugger by using `*break-on-signals*` appropriately. Notice that the timeout will still be valid – so to prevent sudden exit out of the debugger, one should probably specify a timeout of `nil` using the `:timeout` clause described below.

In case of test failure, the test report will contain: a message explaining the failure condition, possibly existing backtraces if `*include-backtrace*` is true, and some statistics regarding how many tasks were spawned in total, how many terminated by themselves, how many aborted (either due to error or plan failure), and how many had to be killed after test execution.

Each *option* has the following syntax:

```
option  ::= (:N-RUNS n)
          | (:TIMEOUT timeout)
          | (:GENERATORS binding+)
          | (:SKIP reason)
binding ::= (var value [guard])
```

`(:n-runs` ***n***`)`
: specifies that the test should be executed n times in a row.

`(:timeout` ***timeout***`)`
: specifies a timeout in seconds per single run. If *body* does not terminate until *timeout* has expired, a test failure will be generated. (*timeout* may be `nil` which denotes no timeout.)

`(:generators` ***binding****`)`
: Each *value* in the *bindings* is evaluated, and must return a generator (e.g. `gen-integer`, `gen-float`, `gen-list`) which is used to generate random data in each test run. A binding is established between each *var* and the random value generated. *guard* is a form (executed while bindings of each *var* is active) which must evaluate to true otherwise new random values are tried.

`(:skip` ***reason***`)`
: specifies that the test should be skipped, printing *reason* in the test report.

Notice that a documentation string must be provided. This choice was made deliberately as it can be difficult to quickly infer from a test case what precisely it is supposed to test from a high-level point of view.

[Macro] **with-task-hierarchy** hierarchy definitions &body body

with-task-hierarchy helps to conveniently and concisely generate a whole tree of interdependent tasks. *hierarchy* describes the structure of the tree to be generated, and *definitions* specify what forms each task is supposed to execute.

In principle, tests could also use CPL directly, however `with-task-hierarchy` has two advantages: on one hand, it makes the hierarchy very explicit and hence easily visualizable, and on the other hand, it was added to decouple the testing of the task implementation from the plan language.

hierarchy and *definitions* have the following syntax:

```
hierarchy   ::= (clause+)
clause      ::= (name -> sub-task*)

definitions ::= (definition+)
definition  ::= (:task  name   . task-body)
              | (:tasks name n . task-body)
```

During execution of *body*, each *name* is bound to a task object (or a list of task objects in case of `:tasks`) which execute their respective *task-body* and whose children are the specified *sub-tasks*. The specified hierarchy must not be cyclic, but may contain multiple roots. The parameter n is interpreted multiplicatively to the number of parent tasks. For example, in

```
(with-task-hierarchy ((As -> Bs)
                      (Bs ->   ))
     ((:tasks As 10 ..body-A..)
      (:tasks Bs 5  ..body-B..))
  ...)
```

a binding for *Bs* will be established which contains a list of 50 tasks executing *body-B*.

[Function] **wait-until** predicate &rest tasks

predicate is called on each one out of *tasks* and should return a fluent; `wait-until` waits until all the fluents have true values.
Each task in *tasks* can actually be a list of task objects which are regarded as if they were passed directly.

[Function] **become** &rest states

Predicate suitable for wait-until: `(wait-until (become` $s_1 \ldots s_i$`)` $t_1 \ldots t_j$`)` waits until the status of all tasks $t_1 \ldots t_j$ is one of $s_1 \ldots s_i$.

[Function] **no-longer** &rest states

Opposite of `become`: `(wait-until (no-longer` $s_1 \ldots s_i$`)` $t_1 \ldots t_j$`)` waits until the status of all tasks $t_1 \ldots t_j$ is *not* one of $s_1 \ldots s_i$.

[Function] **has-status** task &rest states

Check whether *task* currently has a status among *states*. If not, a test failure is provoked.

[Function] **have-status** tasks &rest states

Like above but *tasks* is a list of task objects.

[Functions] **suspend** / **wake-up** / **evaporate** &rest tasks

Suspends, wakes up, evaporates each task in *tasks*. Each task in *tasks* can actually be a list of task objects which are then regarded as if they were directly passed.

Example

```
(define-cram-test basics.1
    "Grant-childs awake before childs awake before parent."
    ((:timeout 5.0)
     (:n-runs  10)
     (:generators (i (gen-integer :min 1 :max 20))
                  (j (gen-integer :min 1 :max 5))))
  (with-task-hierarchy ((A  -> Bs)
                        (Bs -> Cs)
                        (Cs ->   ))
     ((:task  A    (sleep 0.5))
      (:tasks Bs i (sleep 0.1))
      (:tasks Cs j (sleep 0.01)))
    (wait-until (become +dead+) A Bs Cs)
    (has-status  A  :succeeded)
    (have-status Bs :succeeded)
    (have-status Cs :succeeded)))
```

The generators will in each run bind i to an integer in the range [1..20], and j to an integer in range [1..5]. These values are then used to initialize the number of children and grand children in the created task hierarchy, see fig.5. The tasks are defined such that, first, the grand children will terminate (shortest sleep interval), then the children, then the parent. This is how it's supposed to be – sub tasks must be completed before their parent task can be accomplished – and hence we check that all tasks will actually have succeeded.

Considering the conciseness of the example, we are of the opinion that in an

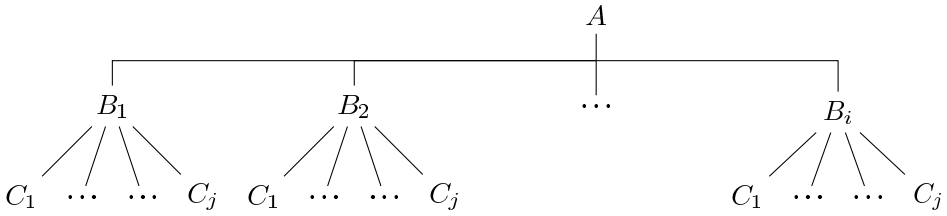

Figure 5: Visualization of the task hierarchy.

expressive language like Common Lisp, writing tests can actually be fun rather than tiring and tedious. The example illustrates how it allows to concentrate solely on *what* one wants to test rather than on what one would have to do to reach a point where one could possibly test what one had in mind to test. In Common Lisp, one can achieve such domain specific languages in reasonably short amount of code and time with the tools the language provides – a big plus as routiniers of the language are familiar with the process, and tend to extend the language in ways that reflects culture, so fellow programmers are able to take up extensions quickly and with less cognitive overhead than, say, some kind of external framework might incur.

Future work

The utilities `wait-until`, `become`, `no-longer` are convenient and address a need which comes up regularly, especially within the implementation of the CPL macros themselves. It's just sugar on top of fluent nets and `wait-for` – which are used in the implementation of the macros at the moment – still, it may make sense to incorporate those utilitiy functions into CPL proper.

3.2.2 Logging Facility

Overview

CRAM includes an advanced logging facility which is used in all crucial points of its implementation. It was written along early attempts at the new task implementation, and quickly proved itself to be an invaluable investment of time for it made internal behaviour traceable and by that comprehensible.
The logging facility has the noteable property of not being based on log levels, but on tags. When writing a log expression, one can associate that expression to one or more log tags. Only those log expressions will emit output at run time whose specified tags are included in the set of tags active at run time. Naturally, it's the user who can decide what tags belong to this "active tag set". Thus, by allowing activation and deactivation of tags at will, the logging facility allows the user to customize the desired degree of logging output in a very fine-grained way, adapted to the problem the user is trying to debug.
While tags allow a high degree of customization, it would soon become tedious having to activate tags anew each (non-consecutive) time one intends to use the logging facility. Hence log levels are provided, too, but they are merely predefined sets of tags. In practise, it's actually the ability to *disable* tags to arbitrarily inhibit superfluous output on high log levels that turned out to be most useful.

Implementation Notes

The active tag set is actually implemented as a bitmask, and each symbolic tag is resolved at compile-time to its corresponding bit in the mask. The run time test, whether a particular log expression is supposed to emit output, can thus be compiled down to a few machine instructions, and is hence very fast.

Synopsis[15]

[Macro] **log-msg** clause+

Each *clause* has the following syntax:

```
clause ::= (:CONTEXT c-fmt-ctrl . c-fmt-args)
         | (:DISPLAY d-fmt-ctrl . d-fmt-args)
         | (:TAGS tag+)
```

`log-msg` emits a log entry to `*log-output*` if the active tag set contains one of the *tags* specified. A log entry will consist of five columns: $\text{timestamp}_{\text{real}}$, $\text{timestamp}_{\text{run}}$, current task, context, and display.

$\text{timestamp}_{\text{real}}$
: denotes a wall clock timestamp in seconds relative to the start of the root of the task hierarchy the currently executing task was spawned within.

[15] All symbols listed are exported from the `cram-language-implementation` (`cpl-impl`) package.

timestamp$_{\mathbf{run}}$
: denotes a timestamp measuring cpu time.

current task
: denotes the (possibly abbreviated) name of the currently executing task.

context
: denotes the result of applying the format control-string *c-fmt-ctrl* to the format arguments *c-fmt-args*.

display
: ditto but applying *d-fmt-ctrl* to *d-fmt-args*.

[Function] **log-use** tags

Sets the active tag set exactly to *tags*.

[Function] **log-enable** tags

Adds each tag in *tags* to the active tag set.

[Function] **log-disable** tags

Removes each tag in *tags* from the active tag set.

[Function] **list-active-log-tags**

Returns the active tag set.

[Variables] +**log-default**+, +**log-verbose**+, +**log-very-verbose**+

Predefined sets of tags to represent different log levels.

Example

We will demonstrate the logging facility on the following simple example, and explain how its output can be used to reinforce our understanding of the interplay of the language constructs described in section 3.1.

```
(top-level
  (pursue (seq (sleep 0.1) :foo)
          (wait-for (end-of-time))
          (par (wait-for (end-of-time))
               (wait-for (end-of-time)))))
```

The function `end-of-time` is part of the `cram-language-tests` package, and simply returns a fluent which will never become true. Knowing that, coupled with the understanding derived from reading section 3.1, we expect the following to happen:

1. The task executing the form `(seq (sleep 0.1) :foo)` will finish first – it will succeed, and return the value `:foo`.

2. `pursue` will receive that value, return it again, thus propagating it further upwards until it's finally returned from the expression as a whole.

3. Before returning, `pursue` will evaporate the remaining tasks executing the `(wait-for (end-of-time))` expressions.

After setting the active tag set to `+log-default+`, running the example above results in the following output shown[16]:

real	run	task name	context	display
0.173	0.144	[PURSUE-C..-#1/3]-15	FINISH	:SUCCEEDED, :FOO
0.178	0.144	[PURSUE]-18	SENT ⟶ [PURSUE-C..-#3/3]-17	#<EVENT :EVAPORATE>
0.178	0.144	[PURSUE]-18	SENT ⟶ [PURSUE-C..-#2/3]-16	#<EVENT :EVAPORATE>
0.178	0.144	[PURSUE]-18	SENT ⟶ [PURSUE-C..-#1/3]-15	#<EVENT :EVAPORATE>
0.178	0.144	[PURSUE]-18	FINISH	:SUCCEEDED, :FOO
0.179	0.144	[PURSUE-C..-#2/3]-16	RECV ⟵ [PURSUE]-18	#<EVENT :EVAPORATE>
0.179	0.144	[PURSUE-C..-#2/3]-16	FINISH	:EVAPORATED, ...
0.180	0.144	[TOP-LEVEL]-1616	SENT ⟶ [PURSUE]-18	#<EVENT :EVAPORATE>
0.180	0.144	[TOP-LEVEL]-1616	FINISH	SUCCEEDED, :FOO
0.190	0.148	[PURSUE-C..-#3/3]-17	RECV ⟵ [PURSUE]-18	#<EVENT :EVAPORATE>
0.190	0.148	[PURSUE-C..-#3/3]-17	SENT ⟶ [PAR]-27	#<EVENT :EVAPORATE>
0.190	0.148	[PURSUE-C..-#3/3]-17	FINISH	:EVAPORATED, ...
0.197	0.144	[PAR]-27	RECV ⟵ [PURSUE-C..-#3/3]-17	#<EVENT :EVAPORATE>
0.197	0.144	[PAR]-27	SENT ⟶ [PAR-CHILD-#2/2]-26	#<EVENT :EVAPORATE>
0.197	0.144	[PAR]-27	SENT ⟶ [PAR-CHILD-#1/2]-25	#<EVENT :EVAPORATE>
0.197	0.144	[PAR]-27	FINISH	:EVAPORATED, ...
0.198	0.152	[PAR-CHILD-#2/2]-26	RECV ⟵ [PAR]-27	#<EVENT :EVAPORATE>
0.198	0.152	[PAR-CHILD-#2/2]-26	FINISH	:EVAPORATED, ...
0.198	0.152	[PAR-CHILD-#1/2]-25	RECV ⟵ [PAR]-27	#<EVENT :EVAPORATE>
0.198	0.152	[PAR-CHILD-#1/2]-25	FINISH	:EVAPORATED, ...

As mentioned in the description of `log-msg`, the first two columns represent cumulative timestamps relative to the start time of `top-level`. Names of tasks are abbreviated in their middle since doing it there is unlikely to accidently elide chunks necessary for identification. Truncating at the end would be especially bad as task names of sub forms of CRAM constructs contain a suffix `-#i/n` denoting the ith of nth total sub forms. So for example, [PURSUE-C..-#1/3]-15 denotes the task executing the first sub form of a `pursue`, i.e. `(seq (sleep 0.1) :foo)` in our case. The exact meaning of the "context", and "display" fields depend on the context in which the `log-msg` form was used;[17] as a rule

[16] Some verbosity was elided to make it fit on a page. Also, the actual output is in ASCII.

[17] For a newcomer to the source base, it's probably a good idea to test the logging facility on simple examples like the one in this section with a very high log level enabled, and then proceed grepping through the source base for the context fields. Juxtaposition of source code and log output should help in gaining understanding of the internals involved.

of thumb, the "context" field often contains the name of an internal function (possibly enriched with additional values), and the "display" field contains parameters passed to that function.

So what does the log show us?

0.173 The task [PURSUE-C..-#1/3]-15, executing the 1st sub form of `pursue`, succeeds and returns the value `:foo`.

0.178 Its parent, [PURSUE]-18, has noticed a child's success, evaporates all its children, and then succeeds with the return value yielded from the succeeded child. The propagation of return values is performed by `join-task`, whose operation is not part of the log level `+log-default+`. Notice that [PURSUE]-18 actually evaporates *all* its children, including the one who already finished – however, as that task is dead already, it won't be able to react on events anymore, so the superfluous event will just be ignored.

0.179 The task [PURSUE-C..-#2/3]-16, executing the 2nd sub form of `pursue`, evaporates. As the task does not have any children, it does not propagate the evaporation event any further.

0.180 The task [TOP-LEVEL]-1616, executing the top-level form, succeeds, and return the value `:foo` stemming from the `pursue` sub form, to its caller, the REPL. (And again, as fixed part of a task teardown's process, the task tries to evaporates its child, which, again, is dead at that point already.)

0.190..8 The task executing the 3rd sub form of the `pursue` form evaporates, propagates the evaporation event to the task executing the `par` form which again propagates the event to the tasks executing the `par` sub forms until all tasks spawned will finally have terminated.

A few things remain to be mentioned. First, the increase and then decrease of run time in 0.190 to 0.197 is a puzzling phenemenon that currently defeats our ability of explanation.[18] Second, from looking at the log, it's obvious how gratuitous the CRAM constructs spawn new tasks, and by that threads. Third, the log shows that parent threads do not wait until their children have performed desired state transitions induced by the parent.[19] Last, running the example with the log level `+log-very-verbose+` would have resulted in a table with over 100 entries.

[18] The second column is based on `get-internal-run-time` which, on SBCL, is implemented by calling `getrusage(2)` with `RUSAGE_SELF` as argument, denoting per-process timings. Furthermore, the example was actually executed on a uniprocessor machine.

[19] Actually, the implementation supports a mode in which parents *do* wait until their children have transitioned properly. If future proves that the waiting won't impair performance, it'll probably be a good idea to wait by default so invariants are ensured.

Future work

The second field of the log output, indicating CPU time, was added with the intention to fathom involuntary context switches caused by the operating system. The field's value is computed using `get-internal-run-time` which returns the sum of user and system time used by the current process. However, it does not include the time the process *voluntarily* waited (on I/O, synchronization), so comparing the wall clock time to the cpu time timestamp does not yield much information. We experimented using the `ru_nvcsw`, and `ru_nivcsw` slot values of the result returned by `getrusage` not to much avail (but then we may just have not been able to develop a good model of understanding of the details involved in the short time.) There may be venues of improvement.

At the moment, the logging output is emitted to `*log-output*` which defaults to `*standard-output*`. ROS, the robot middleware being used in combination with CRAM, has a special interface for logging and debugging output to centralize the collection of such data. It may make sense to plug the logging output of `log-msg` into that interface.

4 Conclusion

In the first part of this bachelor thesis, we discussed design considerations and implementation issues of the CRAM Plan Language: we glanced at fluents and shortly depicted how they're an ingredient part of our plan language as they conveniently allow tracking of changes inside and outside of the system. We continued to discuss tasks, how they are organized in hierarchies, and how their hierarchical organization leads to particular kinds of inter-task communication. We then argued that the robustness of inter-task communication plays the crucial role in the robustness of the task implementation as a whole. Following, we argued that asynchronous interrupts for intrinsic reasons can't result in robust code because their handling would require knowledge over all the software involved. We proposed to base the task implementation on a synchroneous mechanism instead, and suggested SBCL's deadline to provide the necessary machinery. There upon, we described the new task implementation based on an event loop which is periodically entered and which acts on messages stemming from other tasks. However, we eventually came to the conclusion that using SBCL's deadlines would still require us to have knowledge about all places where synchronization may be used internallly, and are hence impractical as well. We outlined an approach where we essentially implement our own deadlines so only our code will be affected, and code localization is retained.
In the second part we provided extensive documentation of CRAM's current API, and described other tools which may come handy to a user, like the test suite and the task-related logging facility.

References

[1] "Why Are `Thread.stop`, `Thread.suspend`, `Thread.resume` and `Runtime.runFinalizersOnExit` Deprecated?," `http://java.sun.com/j2se/1.4.2/docs/guide/misc/threadPrimitiveDeprecation.html`, retrieved 2010-05-04.

[2] McDermott, D. "A Reactive Plan Language," Yale University, Research Report YALEU/DCS/RR-864, 1991.

[3] McDermott, D. "Revised NISP Manual," Yale Computer Science Department Report 642, 1988.

[4] Mösenlechner, L. and Demmel, N. and Beetz, M. "Becoming Action-aware through Reasoning about Logged Plan Execution Traces", *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2010, submitted.

[5] Newman, W. H. et al. "SBCL User Manual" `http://www.sbcl.org/manual/`, retrieved 2010-05-04.

[6] Pitman, K. M. "Condition handling in the Lisp language family", *Advances in exception handling techniques*, Springer Verlag, 2001, pp. 39-59.

[7] Pitman, K.M. "9.1 Condition System Concepts", *Common Lisp Hyperspec*. (`http://www.lispworks.com/documentation/HyperSpec/Body/09_a.htm`), retrieved 2010-05-04.

[8] Quigley, M. and Conley, K. and Gerkey, B. and Faust, J. and Foote, T. and Leibs, J. and Wheeler, R. and Ng, A. "ROS: an open-source Robot Operating System," *IEEE International Conference on Robotics and Automation,* ICRA 2009

[9] Siivola, N., "Threads, Interrupts, Transactions", `http://random-state.net/log/3386927147.html`, retrieved 2010-05-04.

[10] Tenorth, M. and Beetz, M. "KnowRob – Knowledge Processing for Autonomous Personal Robots," *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2009.