How to Cancel a Task

Alexey Kolesnichenko, Sebastian Nanz, and Bertrand Meyer

ETH Zurich, Switzerland
firstname.lastname@inf.ethz.ch

Abstract. Task parallelism is ubiquitous in modern applications for event-based, distributed, or reactive systems. In this type of programming, the ability to cancel a running task arises as a critical feature. Although there are a variety of cancellation techniques, a comprehensive account of their characteristics is missing. This paper provides a classification of task cancellation patterns, as well as a detailed analysis of their advantages and disadvantages. One promising approach is cooperative cancellation, where threads must be continuously prepared for external cancellation requests. Based on this pattern, we propose an extension of SCOOP, an object-oriented concurrency model.

1 Introduction

Task parallelism has become part of the standard inventory of professional developers, and programming frameworks for this domain are sprouting to help them express their intentions in a safe and concise manner. At the same time, learning to proficiently use such frameworks is far from easy. They offer a confusing variety of abstractions and constructs, often to provide similar but subtly different functionality. Frequently, the only source of information are code examples where the relevance of the constructs cannot be sufficiently discussed. Too little research is spent on consolidating the various approaches by explaining commonalities and differences, which would help developers learn to use new frameworks more quickly and aid designers in developing their frameworks further.

This paper strives to address these deficiencies, focusing on one central problem in task parallelism: task cancellation techniques. Cancellable tasks are mainly used for interrupting long-running or outdated tasks, but the pattern can also be used as a building block for more high-level patterns, such as MapReduce. The paper provides an overview of existing cancellation approaches, extracting techniques from different programming languages and concurrency libraries, classifying them, and discussing their strong and their weak points. This knowledge is then applied to provide a novel task cancellation technique for SCOOP [8,10], an object-oriented concurrency model. The technique is based on the idea of cooperative cancellation where both the canceling and the canceled task must cooperate in order to succeed.

The remainder of the paper is structured as follows. Section 2 provides a taxonomy and discussion of task cancellation techniques. Section 3 describes a cooperative cancellation technique for SCOOP. Section 4 provides an overview of related work and Section 5 concludes with an outlook on future work.

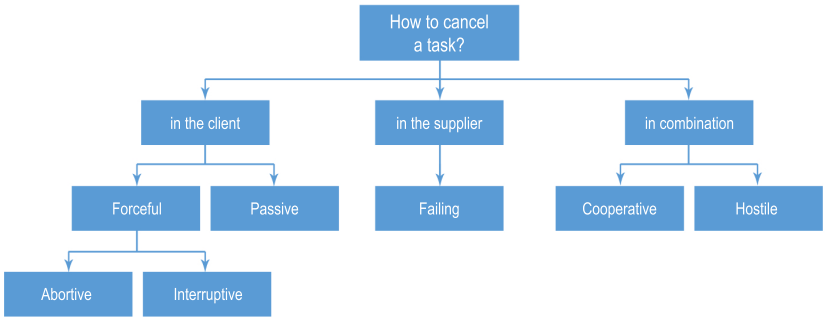

Fig. 1. A taxonomy of task cancellation techniques

2 Classification of task cancellation techniques

A *task* denotes the abstraction of an execution, such as a CPU thread, a thread pool, or a remote machine. Designing a programming model for task parallelism has to deal with the cancellation of tasks, a highly reusable pattern which can be applied to stand-alone applications, client-server systems, and distributed clusters alike. Without proper support for task cancellation, the developer has to write the synchronization code by hand, a task prone to subtle errors.

Various approaches to canceling a running task have been implemented in programming languages and libraries and described in theory. However, so far there is little evaluation and comparison of the proposed techniques. To provide a foundation for discussing them, we have examined a number of popular languages (Java, Python, C# TPL, Pthreads, etc.) and provide a taxonomy in Figure 1.

Client-based cancellation describes techniques where the control over the cancellation process lies entirely with the client (the canceling task):
- *Forceful cancellation* The client forces the supplier (the canceled task) to stop without the possibility to resist:
  - *Abortive cancellation* The supplier is terminated immediately.
  - *Interruptive cancellation* The supplier is allowed to reach a safe point before being terminated.
- *Passive cancellation* The client stops waiting for the result of a supplier, allowing it to continue on its own.

Supplier-based cancellation describes techniques where the control over the cancellation process lies entirely with the supplier:
- *Failing* The supplier encounters an unrecoverable error and needs to inform its clients.

Client/supplier combination describes techniques where client and supplier must act together in order to succeed with the cancellation:
- *Cooperative cancellation* The client asks the supplier to terminate, which decides itself how and when it should terminate.
- *Hostile cancellation* The supplier may resist a cancellation request of the client, and interrupt the client instead.

In the following, we discuss each of the approaches and provide examples of languages where they are employed.

2.1 Client-based cancellation

Abortive cancellation. Immediate termination does not give the canceled thread a chance to respond. As an example, consider the Java code in Listing 1, where `t.stop()` aborts the thread.

```
Thread t = new Thread(){ @override void run(){...} }
t.start();
...
t.stop(); // aborts the running thread
```

Listing 1. Aborting a thread-based task in Java

The advantage of the approach is clearly its simplicity. However, the approach is unsafe because aborting a running thread can leave a program in an inconsistent state. Consider the money transfer example in Listing 2.

```
void transfer(Account from, Account to, int amount) {
  synchronized{
    from.withdraw(amount); // if stopped here, money is lost
    to.deposit(amount);
  }
}
```

Listing 2. Unsafe cancellation using abortion

In this example, a synchronized block is used to guarantee that no thread interferes with the transfer. However, if a running thread is aborted during execution and is forced to unlock all of the monitors that it has locked, the transferred money may be lost and the remaining execution started in an inconsistent state. The Pthreads library [12] with `set_cancellalation mode` set to `PTHREAD_CANCEL_ASYNCHRONOUS` is a further example of abortive cancellation.

Interruptive cancellation. Using this technique, a running task is aware of potential interruption and usually cannot ignore it. However, a task cannot be canceled in every execution state, but only at so-called *safe points*: places where certain program invariants hold such that the execution may be interrupted safely. Usually a programmer must specify these places by hand, either by calling a library function, or handling a specific type of exception [5]. A special case of this technique allows interrupting a task at only one point in its lifetime: when the task has not been started yet. While it may seem not very useful, in some languages (Scala [13], Python [3]) this is the only built-in cancellation mechanism.

Consider the example in the Pthreads library[1] in Listing 3.

[1] Pthreads supports two cancellation modes: Deferred (as in the example) and Asynchronous. The latter one is an example of *aborting* tasks, with no safety guarantees.

```
// setting the cancellation mode to interruption
pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
...
void* CancellablePthread(void* argument){
  ...
  pthread_testcancel(); // the execution can be safely canceled here
}
```

Listing 3. Cancellation points in Pthreads

At a safe point for cancellation, the call `pthread_testcancel()` checks on potential cancellation requests. Additionally, some of the blocking system calls are also considered to be cancellation points in Pthreads [12]. Java's thread interruptions[2] and `thread.Abort()` in C# (unlike the method's name is suggesting) are another examples of interruptive cancellation [6,1].

Its potential safety guarantees are a benefit of this approach: *if* the approach is applied correctly, a program can be considered to be in a consistent state after a cancellation. Writing correct interruption-aware code is however difficult [11,1] as a programmer has to remember subtle rules (e.g. in Pthreads some I/O calls are interruptible, others are not) and maintain a program's invariants by hand.

Passive cancellation. This technique is different from the forceful methods in that a canceling task does not need to become active: it simply stops waiting for a task result, while the running task is still being executed.

As an example, consider downloading a file over a network, illustrated in Listing 4 with C#'s Task Parallel Library (TPL). The call to a `StartDownload()` is asynchronous and returns only a handle to a future (an object, representing a computation that is still being computed [2]), represented by the `Task` class. After some time the downloader's result might not be needed anymore, i.e. the execution is abandoned (in the **if** branch).

```
void PassiveCancellation(string url){
  Downloader downloader = new Downloader(url);
  Task<byte[]> bytesFuture = downloader.StartDownload();
  ...
  if(noNeedToDownload) {
    // the download is not needed anymore
    return; // data is still being downloaded ...
  }
  else {
    // the download is still needed
    var result = bytesFuture.Result; // fetching the result
  }
}
```

Listing 4. Passive cancellation in the Task Parallel Library (TPL) of C#

Obviously, this approach is not uniformly applicable; for example, we might still want to cancel a state-changing procedure. And it is important to know

[2] User-defined code may ignore interruption [11], but only between calls to library methods (which will not ignore it).

in advance that the task will eventually be completed, i.e. listening to a TCP-socket cannot be canceled in this way. Another disadvantage is that the running task continues to consume machine resources. However, in a distributed setting this approach can find its application: consider a framework for a distributed computing, such as MapReduce. Often for the last piece of work several tasks are spawned [4] but only a single result will be used. In this case, there is no need to write sophisticated cancellation code, and it is valid to "forget" about the remaining executing tasks.

2.2 Supplier-based cancellation

This class of techniques deals with the special case that cancellation is not requested by a client but that a failure happens in supplier, i.e. it cannot fulfill it's obligations to clients; the supplier therefore needs to terminate. To indicate a failure, exceptions are typically used in object-oriented programming environments. Hence, this case boils down to the problem of exception handling in concurrent environments [9], which is not the focus of this paper.

2.3 Client/supplier combination

Cooperative cancellation. A gentle way to stop a task is to cooperate and *ask* it to do so. The rationale for this approach is simple: a task is the abstraction of an execution, and hence should contain the information about how and when it should be stopped.

In other words, a task must be ready to be canceled at any time by external request. C#'s Task Parallel Library (TPL) follows this pattern, where a single point of cooperation is denoted by two classes: `CancellationTokenSource`, a generator of `CancellationToken`, which itself is a concrete request to cease the execution. An example is given in Listing 5.

```
void Client(){
    var cts = new CancellationTokenSource();  // create the token source
    // pass the token to the cancelable operation
    Task.Run(() => Supplier(cts.Token));
    ...
    cts.Cancel(); // request cancellation
}
void Supplier(CancellationToken token) {
    for (int i = 0; i < 100000; i++) {
        // some work
        if (token.IsCancellationRequested) {
            break; // potentially perform cleanup, terminate
        }
    }
}
```

Listing 5. Cooperative cancellation in TPL

This technique provides a solid general structure for writing a cancellable tasks (see Listing 6), with a guarantee that no invariants will be violated. Unlike

	retain	*yield*
demand	the client is interrupted	the supplier is interrupted
insist	the client waits	the supplier is interrupted

Table 1. Dueling rules

in interruptive cancellation, the programmer does not need to remember subtle rules of a concrete library or language. Cooperative cancellation also does not require any runtime support. Unfortunately, one cannot use the true power of this technique unless libraries support this pattern too (as far as we know, to date only limited support is introduced in C#). As another disadvantage, the latency between a cancellation request and actual cancellation is increased.

```
void function run(CancelRequest cancel)
  while(not is_done){
    if cancel is requested
      exit
    loop_once
  }
//need to be specified for concrete task.
void function loop_once;
boolean function is_done;
```

Listing 6. General structure of a cancellable task in cooperative cancellation

Hostile cancellation. While in the previous paragraph client and supplier are cooperating in order to succeed, in hostile techniques involve a struggle between the canceling and the canceled task. We describe these techniques on the example of *duels*, a mechanism was theoretically described in [8] but not yet implemented. We are using the terminology from [8] in the rest of this chapter.

The key insight in this approach is that a canceling task (a "challenger", in the original) might not be strong enough to request an actual cancellation. If the canceling task is worthy enough, its request is fulfilled (the task is "killed"); if not, it gets an exception itself (therefore the approach is named a duel). In other words, dueling is a two-way interruption, where the result depends on which of the tasks is stronger.

To specify its preferences, a supplier can be in one of the two modes: either *retain* or *yield*. The former means that the task refuses to be canceled, and the latter specifies that it is OK to be interrupted. On the side of the client, there are also two options available: *demand* and *insist*. The first is more impatient, the second more gentle. The complete set of rules is shown in Table 1.

The dueling mechanism is useful in environments where executions are prioritized. For example, one can imagine a robotics system that needs to handle simultaneously a variety of different tasks: route planning, controlling the motors, etc. These tasks can arise non-deterministically and compete for processing units, attempting to cancel other activities. However, it is completely unacceptable that low priority task succeeds in canceling a more important one (for

```
producer: separate PRODUCER
consumer: separate CONSUMER
buffer: separate BUFFER [INTEGER]

  consume (a_buffer: separate BUFFER [INTEGER])
  -- consume an item from the buffer
  require
    not (a_buffer.count = 0)
  local
    consumed_item: INTEGER
  do
    consumed_item := a_buffer.item
  end
```

Listing 7. Producer-consumer example in SCOOP

example, the data collection routine should not be able to cancel a task that adjusts the speed of the motors). In this case, a proper setup of dueling rules could both permit a cancellation request from high priority challengers and provide security from cancellation for important computational tasks.

3 Cooperative cancellation in SCOOP

This section provides an introduction to SCOOP, an object-oriented concurrency model for contract-equipped languages, and evaluates the patterns of Section 2 for use within this model. Furthermore, the section shows how the cooperative cancellation pattern can be applied to a SCOOP, implemented in Eiffel. For the rest of the paper, we are using Eiffel notation and terminology [8].

3.1 Overview of SCOOP

The goal of SCOOP [8,10] is to provide a simple and safe way to write concurrent code while retaining sequential object-oriented programming principles in as far as possible. Each object in SCOOP is associated with a *processor* (typically implemented as a thread), called its *handler*. Features of objects that reside on different processors can be executed in parallel. The keyword **separate** is used to mark objects residing on different processors, relative to the current object.

The producer-consumer problem serves as an illustration of these ideas. The main entities `producer`, `consumer`, and `buffer` are shown in Listing 7. The keyword **separate** specifies that the referenced objects may be handled by a processor different from the current one. A *creation instruction* on a separate entity such as `producer` will create an object on another processor; by default the instruction also creates that processor.

A consumer accesses an unbounded buffer in a feature call `a_buffer.item`. To ensure exclusive access, the consumer must lock the buffer before accessing it. Such locking requirements of a feature must be expressed in the formal argument list: any target of separate type within the feature must occur as a formal argument; the arguments' handlers are locked for the duration of the feature

execution, thus preventing data races. Such targets are called *controlled*. For instance, in `consume`, `a_buffer` is a formal argument; the consumer has exclusive access to the buffer while executing `consume`.

Condition synchronization relies on preconditions (after the **require** keyword) to express wait conditions. Any precondition makes the execution of the feature wait until the condition is true. For example, the precondition of `consume` delays the execution until the buffer is not empty.

3.2 Choosing a cancellation mechanism for SCOOP

The main goal of SCOOP is to provide an easy-to-use model for expressing concurrency, with a focus on the correctness of the resulting programs. Any cancellation mechanism proposed for SCOOP must be designed in this spirit.

Clearly, *abortive cancellation* is an error-prone pattern, and it does not go well with SCOOP's focus on design-by-contract mechanisms. *Interruptive cancellation* has no strict correctness guarantees and can be complicated to use, which does not correspond to SCOOP's simplicity principle. In other concurrency models, where stricter techniques are not favored, interrupting may be a viable option. As mentioned, *passive cancellation* does not need to be explicitly implemented. As it is highly depended on particular usage scenarios, and has no guarantees that it will succeed (the termination of a passively canceled thread is not ensured), it also partly contradicts SCOOP design principles.

A concurrent object-oriented language needs to have well-defined rules about exception handling. The SCOOP implementation is discussed in [9].

As a simple and safe approach, *cooperative cancellation* is a natural candidate to be implemented in SCOOP. It can be implemented as a library approach, thus even eliminating the need to modify the compiler. *Dueling* could be considered as an alternative to cooperative cancellation. However, while duels are only a good fit for specific scenarios, and less suitable in others, we prefer cooperative cancellation as a general-purpose approach.

3.3 SCOOP with cooperative cancellation

It is instructive to try to directly implement the approach introduced in Listing 6 in SCOOP. One can start with an abstract[3] class `CANCELLABLE_EXECUTOR` and introduce a `CANCEL_REQUEST` as a shared object that propagates a cancellation request; the descendants need to define the termination criteria in `is_done` and a single loop iteration `loop_once`.

An example of using this implementation of cooperative cancellation is shown in Listing 8. Unfortunately, this attempt does not work because in SCOOP a cross-processor call of `run` would force the executing processor to block on executing the loop for the entire execution. Thus all subsequent cancellation requests would be queued in the processor's request queue, effectively making `CANCELLABLE_EXECUTOR` useless. This happens because `CANCELLABLE_EXECUTOR` is both responsible for listening to cancellation requests and the execution itself.

[3] **deferred** in Eiffel notation.

```
executor: separate CONCRETE_CANCELLABLE_EXECUTOR
cancel: separate CANCEL_REQUEST --shared between two processors
...
-- launching an execution
executor.run (cancel) -- execution started on different processor

cancel.request -- canceling an execution
```

Listing 8. Usage of `CANCELLABLE_EXECUTOR`

This problem can be solved by decoupling the listening and the execution logic; the design is provided in Figure 2. The `CANCELLABLE_EXECUTOR` is now responsible only for listening for cancellation requests; the actual execution is handled by a different processor. To represent a concrete execution, a concretization of the deferred class `EXECUTION_UNIT` is needed. The `CANCEL_REQUEST` may not be actually **separate**, but keeping it this way provides additional flexibility for the case when cancellation request is coming from a client residing on processor separate from `CANCELLABLE_EXECUTOR`.

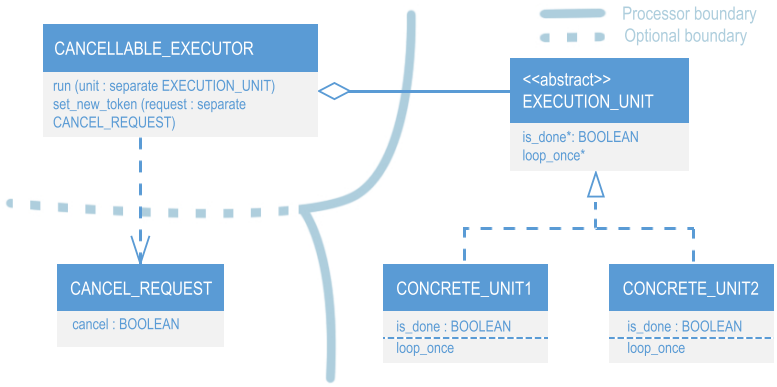

Fig. 2. SCOOP cancellation design

In this design, `EXECUTION_UNIT` is a deferred class, only responsible for performing a single-loop iteration and termination criteria[4]. A task's life cycle is expressed in `CANCELLABLE_EXECUTOR` (see Listing 9), with the following methods:

- `make` (omitted) creates an empty cancellation request. At this point execution cannot be canceled from the outside.
- `set_new_token(a_token: separate CANCELLATION_REQUEST)` sets a new cancellation request, enabling a cancellation.
- `run(a_unit: separate EXECUTION_UNIT)` accepts the execution unit (where execution details are encapsulated) and starts the cancellation-aware execution, according to Listing 6.

[4] This functionality could also be implemented with Eiffel *agents* (function objects), but we present an abstract class to avoid providing execution details.

```
class CANCELLABLE_EXECUTOR
feature
  set_new_token (a_token: separate CANCELLATION_REQUEST)
  do token := a_token end

  run (a_unit: separate EXECUTION_UNIT)
  do
    from until
      is_done (a_unit) or cancel_requested
    loop
      if check_cancel_requested (token) then
        cancel_requested := TRUE
      else
        loop_once (a_unit)
      end
    end
    cancel_requested := FALSE
  end
end

feature{NONE}
  cancel_requested: BOOLEAN
  token: separate CANCELLATION_REQUEST
```

Listing 9. Cancellable executor

As soon as a cancellation is requested, the loop body can be executed at most once more. After one cancellation request, the instance of `CANCEL_REQUEST` becomes useless, therefore we provide `set_new_token` to refresh a request as many times as needed.

3.4 Example of usage

As an example of using cooperative cancellation in SCOOP, we present a downloader application that requests a URL, starts a background download process and provides progress reports. While still in progress, downloading can be canceled by the user. The complete source code is available for download;[5] in the following description, we focus on key aspects of this application.

The `DOWNLOADER_UNIT`, responsible for downloading a single portion of bytes from specified URL, is shown in Listing 10 (some code is omitted for brevity). Note that a **separate STRING** is required in the constructor to obtain control over it, as `DOWNLOADER_UNIT` resides on a different processor than its clients. The implementation is straightforward otherwise. Launching an asynchronous download task is done in a pattern similar to Listing 8, applying the design in Section 3.3. One should create one `CANCEL_REQUEST` per launch: the cancel requests are designed to be used only once.

4 Related work

To the best of our knowledge, this work is the first to attempt a comprehensive classification and evaluation of task cancellation techniques. The work closest

[5] `http://se.inf.ethz.ch/people/kolesnichenko/src/downloader_sample.7z`

```
class DOWNLOADER_UNIT inherit EXECUTION_UNIT
feature
  make (a_url: separate STRING)
  do
    -- use URL 'a_url' and init http_downloader (omitted)
    create parts.make -- create a storage buffer
  end

  is_done: BOOLEAN -- done when all bytes are transferred
  do
    Result := http_downloder.bytes_transferred = http_downloder.count
  end

  action
  do
    http_downloder.read
    if attached http_downloder.last_packet as last_p then
      parts.put_front (last_p)
    end
  end

feature {NONE}
  http_downloder: HTTP_PROTOCOL
  parts: LINKED_LIST [STRING] -- buffer for content
end
```

Listing 10. Example: Downloader unit

to ours is [11], Chapter 7, where some cancellation techniques are discussed for Java. In particular, cooperative cancellation with a shared variable or future is presented, along with rules to write a correct interrupt-aware code.

Further related work is also found in descriptions of individual techniques as part of language and library designs. The degree of support of task cancellation varies in such approaches. C# natively supports interruptive cancellation, and since release of TPL also cooperative techniques were introduced [7].

Things get even more complicated when cancellation involves several tasks that need to agree on shutting down and terminate in a safe order. One approach taking this into account is applied to OpenMP [14]. The authors introduce an abortive cancellation of already launched OpenMP tasks (which boils down to the cancellation mechanisms of Pthreads), with unrestricted possibility to cancel unstarted tasks. Their techique works on task groups, involving a child-parent relationship allowing to cancel the whole group, starting from the root.

Python supports interruptive cancellation of non-started tasks via executors [3] and abortive cancellation of already started ones. Similiarly, Scala supports the `Cancellable` interface which allows canceling only non-started tasks [13].

Java supports interruptive cancellations natively [11]. Pthreads library supports both abortion and interruption, depending on the setup [12].

5 Conclusion

The role of parallel programming in modern applications is notable and continuously increasing; research into programming models for this setting is therefore

critical. Indeed, many new models and frameworks for concurrent and parallel programming have been proposed in the past decade. To provide guidance in this vast field, it is important to consolidate the knowledge about commonly used patterns, and to provide a coherent frame for discussion and evaluation. In this paper we selected one important task parallelism pattern, task cancellation, provided a taxonomy of techniques, scrutinized their usage, and proposed a novel cancellation technique for SCOOP on this basis.

In future work, cooperative cancellation in SCOOP could be further extended to support task chaining (canceling an intermediate task causes all other tasks to be canceled) and precondition-aware tasks (these could effectively be deferred in the executing process). Another extension of this work is to provide a formal model, describing the control flow in different cancellation techniques.

Acknowledgments. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 291389, the Hasler Foundation, and ETH (ETHIIRA).

References

1. Albahari, J., Albahari, B.: C# 3.0 in a Nutshell: A Desktop Quick Reference. O'Reilly Media, Incorporated (2007)
2. Baker, Jr., H.C., Hewitt, C.: The incremental garbage collection of processes. In: Artificial Intelligence and Programming Languages. pp. 55–59. ACM (1977)
3. Concurrent futures in Python: `http://docs.python.org/dev/library/concurrent.futures.html` (2013)
4. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Communications of the ACM 51(1), 107–113 (2008)
5. Destroying Threads in C#: `http://msdn.microsoft.com/en-us/library/cyayh29d.aspx` (2013)
6. Hyde, P.: Java thread programming. Sams Pub. (1999)
7. Leijen, D., Schulte, W., Burckhardt, S.: The design of a task parallel library. In: OOPSLA'09. pp. 227–242. ACM (2009)
8. Meyer, B.: Object-Oriented Software Construction. Prentice-Hall, 2nd edn. (1997)
9. Morandi, B., Nanz, S., Meyer, B.: Who is accountable for asynchronous exceptions? In: APSEC'12. pp. 462–471. IEEE Computer Society (2012)
10. Nienaltowski, P.: Practical framework for contract-based concurrent object-oriented programming. Ph.D. thesis, ETH Zurich (2007)
11. Peierls, T., Goetz, B., Bloch, J., Bowbeer, J., Lea, D., Holmes, D.: Java Concurrency in Practice. Addison-Wesley (2005)
12. POSIX threads specification: `http://man7.org/linux/man-pages/man7/pthreads.7.html` (2013)
13. Scala Scheduler: `http://doc.akka.io/docs/akka/snapshot/scala/scheduler.html` (2013)
14. Tahan, O., Brorsson, M., Shawky, M.: Introducing task cancellation to OpenMP. In: IWOMP'12. pp. 73–87. Springer-Verlag (2012)