Chair of Software Engineering

Concurrency Patterns in SCOOP

Master Thesis

Roman Schmocker
ETH Zürich
romasch@student.ethz.ch

10 March 2014 – 10 September 2014

Supervised by:
Alexey Kolesnichenko
Prof. Dr. Bertrand Meyer

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Concurrency Patterns in SCOOP

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Schmocker	Roman

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Winterberg, 8 September 2014	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.

Abstract

The wide distribution of multi-core processors increasingly forces programmers to deal with concurrency. Parallel programming is not easy, but there are many well-known patterns at hand to help developers.

SCOOP, an extension to the Eiffel programming language, provides an alternative approach to concurrent programming compared to the threading model used in many other languages. There is little experience in implementing and using concurrency patterns in SCOOP however.

We have investigated which patterns are used in practice and compiled a detailed list of pattern descriptions. From this list we selected several popular concurrency patterns and implemented them as a reusable Eiffel library. A small performance comparison shows that the new library is faster and more robust for large data sets than a raw SCOOP solution. We also describe some of the challenges when programming in SCOOP for the first time and provide solutions.

Acknowledgements

I would like to thank Alexey Kolesnichenko very much for his great support and helpful advice throughout my thesis. Many thanks go to Prof. Dr. Bertrand Meyer for his insightful comments and for giving me the opportunity to work on this project. I would also like to thank Julian Tschannen, Mischael Schill, Scott West, Sebastian Nanz and others at the Chair of Software Engineering for their useful input.

Roman Schmocker

Contents

1 Introduction

Concurrent programming has become an important part in software engineering due to the advent of multi-core processors. Dealing with parallelism isn't easy however. There are many pitfalls, such as race conditions and deadlocks.

In practice programmers have learned to avoid tricky concurrency problems with the use of some well-known patterns. These patterns are often shipped as part of the standard library of the language, such that users rarely have to implement them.

The Eiffel programming language [2][15] has an extension called SCOOP [17][7], which stands for Simple Concurrent Object-Oriented Programming. SCOOP simplifies concurrent programming a lot and eliminates one source of errors completely, namely race conditions [17]. However, there is little experience on how to implement popular concurrency patterns, like a worker pool, in SCOOP.

This thesis tries to fill this gap by providing a library of reusable concurrency patterns as well as methodical advice on programming in SCOOP. The main contributions are:

- A broad survey of known concurrency patterns.
- The identification of common SCOOP challenges and advice on how to solve them.
- A new library which provides implementations for some selected concurrency patterns. The selection was mainly based on input from the Software Engineering group at ETH Zürich and the study of Java [5] and C# [4] concurrency libraries.

1.1 Overview

Section 2 introduces a list of concurrency patterns which we found and categorized by studying literature and the standard libraries. A brief introduction of the SCOOP model is given in Section 3. Section 4 describes some challenges when programming in SCOOP and how to solve them. The latter two sections may be interesting for programmers having experience in thread programming and who wish to learn SCOOP.

The focus of Section 5 is on the goals and concepts of the concurrency patterns library. It also provides an overview over the available modules and describes which patterns are implemented by which modules.

A detailed explanation over the individual modules is given by Section 6. Finally, Section 7 provides a small performance evaluation of the library.

2 Pattern overview

The following section provides an overview over all patterns found during the initial survey. We introduce five categories of concurrency patterns: data-centric, task-centric, I/O-centric, SCOOP patterns and synchronization primitives.

The data-centric patterns all solve the general problem on how to move data from one thread to another.

Task-centric patterns share the abstract notion of a task. The individual patterns define when and on which thread a task shall be executed.

The I/O patterns are mostly used in server programming. They solve the problem how to distribute asynchronous requests from an I/O socket to a set of worker threads.

The SCOOP patterns form a category just because they are only applicable in SCOOP. Most of them were first described in Piotr Nienaltowski's PhD thesis [17].

The patterns categorized as synchronization primitives are some very basic building blocks which can be used to build concurrent data structures or higher-level patterns.

Finally, the section Miscellaneous Patterns contains several high-level patterns which don't really fit into one of the above categories.

The structure loosely follows the one used in the book by Gamma et al [10], although the description for each pattern is a lot shorter. We also don't show any implementation details here, because the implementation may differ significantly between languages and a detailed description may take a lot of space, which is not suited for an overview chapter. We provide references to books and articles instead which can be used to look up an implementation.

2.1 Data-centric patterns

Producer / Consumer

Identifier:	P/C
Intent:	Provide a synchronized shared buffer. Producer threads put items into the buffer, and consumers remove items.
Applicability:	When participants should not know each other. Also applicable if there's no one-to-one relation between producers and consumers or when buffering is desired.
Status:	Implemented library component
Example:	A logger service where many producers submit log messages to a buffer and a single consumer writes them to a file.
Known applications:	Very widely used. E.g. logging, input processing, buffering web server requests.
Relation to other patterns:	The Worker Pool [WP] uses this pattern for its task queue. Pipeline [PL] and Dataflow Networks [DFN] are chained Producer / Consumer instances.
References:	[8, p. 87], [21, p. 53]

Pipeline

Identifier:	PL
Intent:	Process data in several independent stages.
Applicability:	When the input consist of a stream of data where several processing steps need to be performed.
Status:	Possible library component
Example:	An emailing system that applies a spam filter, database logging, and a virus scan to each incoming email.
Known applications:	Messaging systems, multimedia streaming (receive - decode - display)
Relation to other patterns:	The Producer / Consumer pattern [P/C] is used between two stages. Pipeline is a special form of Dataflow Network [DFN].
References:	[12, p. 305], [14, p. 253], [21, p. 53]

Dataflow Network

Identifier:	DFN
Intent:	Process data in independent stages, with the option to branch and merge data streams.
Applicability:	When the input consists of a stream of data which allows for parallel processing.
Status:	Possible library component
Example:	A video player application that internally has a file decoder stage, which splits the input in an audio and video part for further processing.
Known applications:	The borealis engine [1].
Relation to other patterns:	The pattern is related to Pipeline [PL]. In Dataflow Network however data can be split and forwarded to two different stages and maybe merged again later.
References:	[12, p. 305], [14, p. 261]

Exchanger

Identifier:	EXC
Intent:	Exchange two objects between two threads atomically.
Applicability:	When synchronization and atomicity is required.
Status:	Possible library component.
Example:	A logger with two buffers: One is used by clients to submit messages, the other is used by the logger to write messages. When the latter is empty and the former is full the exchange happens.
Known applications:	-
Relation to other patterns:	Similar to Synchronous Message Passing, except that data passes in both directions.
References:	[8, p. 101], [12, p. 231]

2.2 Task-centric patterns

Worker Pool

Identifier:	WP
Intent:	Avoid expensive thread creation by providing a set of threads that can execute arbitrary operations.
Applicability:	When there are a lot of small tasks that may be executed in parallel.
Status:	Implemented library component
Example:	A set of HTTP request handlers in a web server.
Known applications:	Often used in server applications, e.g. databases, HTTP servers, web services.
Relation to other patterns:	Producer / Consumer [P/C] is used to pass along task objects. Worker Pool is usually an implementation of the Executor Framework [EF].
References:	[8, p. 117], [8, p. 167], [12, p. 290], [21, p. 61], [13]

Future

Identifier:	FUT
Intent:	Run a task asynchronously and fetch the result later.
Applicability:	When a computation may be run in parallel, but creating an extra thread is too expensive.
Status:	Implemented library component
Example:	A web browser which starts download tasks for image files in parallel to rendering an HTML file.
Known applications:	In UI programming for long-running background tasks, or parallelization of numerical computations.
Relation to other patterns:	Futures may be backed by a Worker Pool [WP] that execute them.
References:	[8, p. 125], [12, p. 332], [21, p. 36], [13]
Comment:	The *wait by necessity* semantics of SCOOP [7] also correspond to the Future pattern.

Executor Framework

Identifier:	EF
Intent:	Split task submission from task execution.
Applicability:	When the task execution strategy should be flexible, e.g. using a worker pool or creating a new thread per task.
Status:	Implemented library component
Example:	The Java Executor interface, where descendants can decide wether a submitted Runnable object is executed in the current thread, in a new thread, or by a worker pool.
Known applications:	Java Executor interface, Microsoft Task Parallel Library
Relation to other patterns:	The Worker Pool [WP] is an implementation of the Executor Framework.
References:	[8, p. 117], [12, p. 289]

Timer: Periodic

Identifier:	TP
Intent:	Apply an operation repeatedly in regular intervals.
Applicability:	When an operation, which can be executed in parallel to the application's main thread, needs to be applied repeatedly.
Status:	Implemented library component
Example:	An email client that checks for new messages every five seconds.
Known applications:	Message polling, buffer flushes, background log writes, heartbeat messages, cron jobs.
Relation to other patterns:	Similar to Active Object [AO], but it schedules just one operation repeatedly.
References:	[8, p. 123], [12, p. 298]

Timer: Invoke Later

Identifier:	TIL
Intent:	Invoke a certain operation at a later point in time.
Applicability:	When it is necessary to wait a bit before executing an operation.
Status:	Implemented library component
Example:	Send an email after a delay of one minute, during which the user can still press a cancel button.
Known applications:	"Grace periods" to cancel actions, robotics control, alarm clocks.
Relation to other patterns:	-
References:	[8, p. 123], [12, p. 297]

2.3 I/O-centric patterns

Half-Sync / Half-Async

Identifier:	HS/HA
Intent:	Simplify asynchronous event handling. A thread or an interrupt handler listens for incoming messages and puts them in a synchronized queue. Worker threads then retrieve and handle the messages.
Applicability:	When the application must react to several event sources at the same time.
Status:	not covered
Example:	The network stack in most UNIX system is implemented like this. A network socket is the "queue" which gets filled by interrupt handlers. Application threads take care of handling the data.
Known applications:	Network sockets, web servers.
Relation to other patterns:	-
References:	[9, p. 423]

Leader / Followers

Identifier:	L/F
Intent:	Reduce synchronization overhead when using a thread pool to handle requests on I/O sockets. A leader thread receives a request, promotes the next leader, and then handles the request.
Applicability:	When there are hundreds of I/O sockets.
Status:	not covered
Example:	A web server for a high volume website serving thousands of connections at the same time.
Known applications:	Online Transaction Processing (OLTP) applications.
Relation to other patterns:	Compared to Half-Sync / Half-Async [HS/HA] it avoids the synchronization overhead of a shared queue.
References:	[9, p. 447], [19]

Disruptor

Identifier:	DIS
Intent:	Provide a high-performance ring buffer with a single producer and multiple readers, each assigned to a thread. Readers can have dependencies to other readers and change buffer entries.
Applicability:	When very high throughput in an I/O application is required.
Status:	not covered
Example:	An OLTP system where the producer listens on a socket for new requests. Then there's a reader for each of the following tasks: logging, unmarshalling, request handling.
Known applications:	LMAX Exchange uses this pattern for their trading platform.
Relation to other patterns:	Similar to Half-Synch/Half-Asynch [HS/HA], but buffer entries may be modified in place and accessed by several threads.
References:	[20]

2.4 Miscellaneous patterns

Active Object

Identifier:	AO
Intent:	Pair an object with its own thread. Clients access the active object through a proxy which transforms feature calls to asynchronous messages. The active object runs a main loop where it schedules requests from clients and runs its own code.
Applicability:	When access to a shared resource can be guarded by an object, or when an object should execute its own main loop.
Status:	Implemented language mechanism. Implemented library component.
Example:	A logging service may be implemented as an active object. Clients call `log ("something")` on the proxy which forwards the message to the active object.
Known applications:	The Java Timer class is implemented as an active object. SCOOP separate calls correspond to feature invocation on an active object.
Relation to other patterns:	The Future pattern [FUT] is usually used for active object functions that return a result.
References:	[9, p. 369], [12, p. 367]

Thread-local storage

Identifier:	TLS
Intent:	Provide private heap data for each thread.
Applicability:	When multiple threads run the same code, but each one needs a different set of data, or when the synchronization overhead for shared heap objects is undesirable.
Status:	Implemented language mechanism
Example:	Store the last exception raised in the current thread.
Known applications:	Java and C# both have a class `ThreadLocal<T>`.
Relation to other patterns:	-
References:	[9, p. 475], [8, p. 45], [12, p. 105], [21, p. 53]
Comment:	Native support in SCOOP: Use `once("THREAD")` and a non-separate return type.

Publish / Subscribe

Identifier:	P/S
Intent:	Provide a hook to subscribe to events. In the concurrent context there's often an intermediate broker which receives events from a publisher and forwards them to all subscribers.
Applicability:	When the publisher doesn't need to know the subscribers, and vice versa with the broker solution.
Status:	Implemented library component
Example:	A GUI button has an event "clicked". The application logic can subscribe to it with a handler function.
Known applications:	Event driven programming, GUI frameworks like Java Swing or EiffelVision.
Relation to other patterns:	Similar to the Observer pattern by Gamma et al.[10, p. 293], but events may come with arguments. The Eiffel agent mechanism may be used for Publish / Subscribe.
References:	-

Transactions

Identifier:	TRA
Intent:	Avoid a deadlock by reserving a set of objects one at a time. Abort if an object is already reserved by another thread.
Applicability:	When multiple operations need to be locked and no proper locking order can be established.
Status:	Possible library component
Example:	A banking application where multiple threads apply various operations on a set of bank accounts.
Known applications:	Two-phase locking in database systems.
Relation to other patterns:	-
References:	[12, p. 249]

2.5 SCOOP patterns

Import

Identifier:	IMP
Intent:	Copy an object structure from a separate processor to the local processor.
Applicability:	When it's cheaper to clone the object instead of placing it on its own processor.
Status:	Implemented library component. Future language mechanism.
Example:	Copy the HTTP request string from the network socket listener to a request handler, such that the listener can continue.
Known applications:	The library developed in this thesis makes heavy use of this pattern.
Relation to other patterns:	-
References:	[17, p. 106]

Asynchronous Self-Call

Identifier:	ASC
Intent:	Execute the body of a main loop and then ask another processor to call back the loop body.
Applicability:	When a processor is running its own code, but others need to access data on it from time to time.
Status:	Implemented library component
Example:	A network socket listener that may be stopped by another processor.
Known applications:	The Timer: Periodic [TP] implementation and the echo server example (see Appendix C.2) use asynchronous self-calls.
Relation to other patterns:	Similar to the Active Object pattern [AO], but the Asynchronous Self-Call pattern lets other processors manipulate its data directly.
References:	[17, p. 217]

Separate Proxy

Identifier:	SP
Intent:	Simplify access to a separate object by providing a processor-local proxy.
Applicability:	When a class is reusable (i.e. library code) and usually placed on a separate processor.
Status:	Guideline
Example:	A shared queue which gets accessed by several threads. Each thread creates a processor-local proxy to avoid having to deal with a separate reference.
Known applications:	Most classes in the library have a Separate Proxy.
Relation to other patterns:	The Separate Proxy is a special version of the proxy pattern described by Gamma et al. [10, p. 207].
References:	-

Full Asynchrony

Identifier:	FA
Intent:	Perform an operation completely asynchronously.
Applicability:	When an operation can be executed in parallel and there's no need to wait for a result.
Status:	Future language mechanism
Example:	A logger service where clients just want to send a log message without having to wait.
Known applications:	A workaround is described in [17, p. 215], but it is currently broken in SCOOP.
Relation to other patterns:	-
References:	[17, p. 215]
Comment:	Will be supported natively in the new runtime developed by Scott West [22].

Universal Call

Identifier:	UC
Intent:	Provide a universal enclosing routine to perform a single call on a separate object.
Applicability:	When it doesn't matter if separate calls are interleaved with calls from other processors.
Status:	Designed language mechanism
Example:	A shared queue where producers only insert items.
Known applications:	An implementation is described in [17, p. 213], but it is currently broken in SCOOP.
Relation to other patterns:	The Separate Proxy [SP] is a workaround for the missing universal call.
References:	[17, p. 213]
Comment:	The new language mechanism will probably be a statement like: `separate a as l_a then l_a.do_something end`

2.6 Synchronization primitives

Atomic Operations

Intent:	Avoid the use of locks by using hardware-supported atomic operations.
Status:	not covered
Example:	A lock-free queue using CompareAndSwap.
Known applications:	Low-level primitive which is used to implement lock-free data structures or other synchronization primitives.
References:	[8, p. 319], [12, p. 140]

Locks

Intent:	An object where only one thread at a time succeeds in calling `lock`, and others have to wait.
Status:	Possible library component
Example:	Provide exclusive access on a certain section of code.
Known applications:	Low-level primitive which is often used to implement other synchronization primitives.
References:	[8, p. 277], [12, p. 148]

Try Lock

Intent:	Try to acquire a lock with the option to back off after a certain amount of time.
Status:	Possible library component
Example:	Database transactions may get aborted due to a timeout if they can't lock a resource after a certain amount of time.
Known applications:	Applications with real-time requirements.
References:	[8, p. 277], [12, p. 148]

Read / Write lock

Intent:	Allow multiple concurrent readers but provide exclusive access to a writer.
Status:	Language limitation
Example:	An array with frequent concurrent reads can make use of a read / write lock.
Known applications:	Shared, read-mostly data structures.
References:	[8, p. 286], [12, p. 157]

Semaphore

Intent:	Make sure that only a certain amount of threads can execute a section of code.
Status:	Possible library component
Example:	The dining philosophers pattern, where at most (N-1) philosophers can eat.
Known applications:	Can be used to implement other synchronization primitives.
References:	[8, p. 98], [12, p. 220]

Single Exclusive Access

Intent:	Make sure that at most one thread has access to exactly one shared object or resource.
Status:	Implemented language mechanism
Example:	A counter variable that should only be incremented by one thread at a time to avoid lost updates.
Known applications:	The Java *`synchronized`* and C# *`lock`* statements implement single exclusive access for sections of code.
References:	[8, p. 25], [12, p. 76]

Multiple Exclusive Access

Intent:	Make sure that at most one thread has access to several shared objects or resources.
Status:	Implemented language mechanism
Example:	A money transfer between two bank accounts.
Known applications:	Databases can provide exclusive access over all data items previously used in the same transaction.
References:	-

Barrier

Intent:	Provide a synchronization point where several threads have to meet before continuing.
Status:	Possible library component
Example:	If the computation of a matrix multiplication is divided among threads, a barrier can be used to make sure that all threads finish before the result is used.
Known applications:	Parallel matrix operations, parallel loop processing.
References:	[8, p. 99], [12, p. 362]

Monitor

Intent:	Ensure that only one thread has access to an object. The thread may also wait for a condition to become true.
Status:	Implemented language mechanism
Example:	A shared buffer with conditions `is_empty` and `is_full`.
Known applications:	Java with a combination of `synchronized`, `wait()` and `notifiyAll()`
References:	[9, p. 399], [12, p. 184]
Comment:	The monitor pattern is a combination of single exclusive access and condition variables.

Condition Variables

Intent:	Wait for a certain condition to become true.
Status:	Implemented language mechanism. Possible library component.
Example:	When a buffer is empty, consumers can wait on the `is_not_empty` conditon variable. Producers will send a signal on this variable when a new item is available.
Known applications:	Preconditions in SCOOP are effectively condition variables due to their wait semantics.
References:	[8, p. 298 and 306]

Synchronous Message Passing

Intent:	Send a message from a sender to a receiver synchronously, where both have to wait until the operation has completed.
Status:	Possible library component
Example:	Make a flight reservation with the implicit guarantee that the booking system has received the message.
Known applications:	Main synchronizaton mechanism in message passing systems.
References:	[12, p. 369]

3 The SCOOP model

SCOOP is an extension to the Eiffel programming language that aims to make concurrent programming easier. The basic idea is that every object can be accessed by exactly one computational unit only. This unit is called processor or handler of an object.

The keyword **`separate`** is used to indicate that an object may be handled by a different processor than the handler for **`Current`**. Calls to a separate object ("separate calls") then correspond to sending a message to the foreign processor. There are two types of separate calls: synchronous and asynchronous. If the called feature returns a result the call is synchronous, which means that the current processor has to wait for the foreign processor to finish its task. An asynchonous call happens when the feature is a command, i.e. not returning any result. In that case both processors can proceed concurrently.

A separate call is only allowed if its target is "controlled". Controlling an object means that the user has exclusive access to that object. In that sense controlling corresponds a bit to locking in other languages. In order to control an object it has to appear as a formal argument in the enclosing routine.

SCOOP guarantees that all messages sent by the current processor are handled in the correct order by the foreign processor. The exclusive access and order guarantee ensure that a controlled separate object behaves just like an object in a sequential program. This is the reason why the SCOOP model is so simple: It allows reasoning about a feature body without the need to consider all possible interleavings of two parallel executions.

A new processor is created by calling a creation instruction on a variable which is declared as separate. The new object is then handled by the new processor.

Preconditions in SCOOP have a special role. In a concurrent setting there's often the problem that a correctness condition may change due to unfortunate interleaving, e.g. between checking that a buffer is not empty and then removing an item, the buffer actually becomes empty due to interference from another thread. Therefore SCOOP turns preconditions into wait conditions if they reference a separate object.

There are many advantages to the SCOOP model, such as easier reasoning and absence of data races, but it also has some shortcomings. It is for example often necessary to write lots of little helper functions that just take a separate object and perform a single call on it, because SCOOP enforces that every target of a separate call needs to be controlled.

SCOOP also has performance problems because it transforms every separate call into a message to another processor. This is rather expensve, especially for small functions like array access.

Furthermore, a processor is currently implemented as an operating system thread and creating them is a costly operation that involves context switches. The SCOOP model however encourages the creation of many processors which is not ideal for performance reasons.

4 Challenges in SCOOP

4.1 Object migration

Passing data from one processor to another is often necessary when programming in SCOOP. The most obvious example is the Producer / Consumer pattern [P/C], but it also applies to other situations like providing an argument to an asynchronous command.

There are two categories of objects which can be passed as arguments: expanded and reference types. Passing expanded objects, which also includes basic types such as *INTEGER*, is not a problem in SCOOP due to their copy-semantics property. However, passing a reference object from one processor to another is a bit more tricky, because bad things such as starvation or unintentional lock passing may happen if done wrong.

There are essentially three ways to safely move reference objects from a sender to a receiver processor. The first and easiest solution is to create the data on its own, separate processor:

```
class SENDER feature
  send (a_receiver: separate RECEIVER)
      -- Invoke an asynchronous operation with
      -- an argument on 'a_receiver'.
    local
      args: separate ANY
    do
      create args
      a_receiver.do_something (args)
    end
end

class RECEIVER feature
  do_something (args: separate ANY)
      -- Perform some operation with 'args'.
    do
      print (args)
    end
end
```

Listing 1: *Migrate objects on a separate processor.*

This approach is conceptually easy but not very efficient, especially when the argument object is small. We'll call this solution the Data Processor approach.

Another solution is to create the object on the same handler as the sender object:

```
class SENDER feature
  send (a_receiver: separate RECEIVER)
      -- Invoke an asynchronous operation with
      -- an argument on 'a_receiver'.
    local
      args: ANY
    do
      create args
      a_receiver.do_something (args)
    end
end

class RECEIVER feature
  do_something (args: separate ANY)
      -- Perform some operation with 'args'.
    do
      print (args)
    end
end
```

Listing 2: *Migrate objects with lock passing.*

This solution (the Lock Passing approach) looks almost like the first one. The only change is a missing separate keyword. The semantics however are radically different:

- The feature *do_something* is executed synchronously due to the lock passing mechanism [17, p. 152][7]. This means that the sender class needs to wait for it to finish.
- *RECEIVER* can't lock the argument object any more after *do_something* finishes. In particular this means that the receiver class should not store the argument in one of its attributes, because any attempt to access it later will likely result in starvation. The reason for this is that the handler of *SENDER* will continue its execution, and as long as there's still work to do no other processor can access objects on it .
- Compared to the first approach no new processor is created.

The last method makes use of a special SCOOP mechanism called *import*:

```
class SENDER feature
  send (a_receiver: separate RECEIVER)
      -- Invoke an asynchronous operation with
      -- an argument on 'a_receiver'.
    local
      args: ANY
    do
      create args
      a_receiver.receive_args (args)
      a_receiver.do_something
    end
end

class RECEIVER feature

  received: ANY

  receive_args (args: separate ANY)
      -- Receive some arguments
    do
      received := import (args)
    end

  do_something
      -- Perform some operation.
    do
      print (received)
    end
end
```

Listing 3: *Migrate objects with import.*

The *import* feature copies its argument along with all non-separate references to the local processor. It is somewhat similar to {*ANY*}.*deep_clone*, except that it doesn't follow separate references.

The import solution has several advantages. There is no need for a new processor and the receiver can also keep the argument and do the operation asynchronously. The drawback is that the data needs to be copied. However, for small data items this is usually faster than creating a new processor.

Note that *receive_args* is executed synchronously just as in the Lock Passing approach. To execute *do_something* asynchronously it has therefore been divided into an execution and argument receiving part.

The feature *import* was first described in [17, p. 106], but unfortunately it is currently not implemented in SCOOP. It is possible however to implement it manually with some user support.

4.2 Processor communication

It is often the case that two threads need to communicate with each other. An example would be a user interface with a background download task. The user interface needs to be able to cancel the download, and the download task has to inform the GUI when it is finished.

In SCOOP this is not easily done. Both processors are performing a long-running execution, which doesn't allow other processors to do separate calls on them. Specifically, the GUI processor is in an infinite loop to receive input and repaint the window, whereas the download task is busy receiving chunks of data. Cancellation will not work in this case because the user interface processor will have to wait for the download processor to finish until it can actually access the download task to call *cancel*, which kind of defeats the purpose of the cancellation button. Worse yet, the user interface will freeze until the GUI processor finally gets the lock.

The solution to this kind of problem is to introduce a third processor which is "passive", meaning that it doesn't have a task to perform and only waits for incoming requests. This new processor is known to the other two, "active" processors and handles an object which can be used for communication. In our example the "passive" processor has an object with an *is_cancelled* and *is_finished* boolean flag. The "active" processors then regularly need to check the status of these flags.

The solution to the task cancellation problem comes from a previous paper by the Software Engineering group [11].

4.3 Processor termination

When an application terminates it is necessary to stop any running thread. Sometimes this can be done with processor communication as seen in Section 4.2. A problem arises however when a processor is stuck in a wait condition.

One example of this could be a producer / consumer situation where a consumer is waiting for the buffer to become non-empty. If the producers have terminated already, the consumer never gets the chance to break out of its wait condition and therefore cannot terminate successfully.

The solution is to add a query *is_stop_requested* in the shared buffer and to adapt the wait condition to include the stop request:

```
class
  CONSUMER
```

```

feature -- Status report

 buffer: separate BUFFER

 last_item: INTEGER

 is_stopped: BOOLEAN

feature -- Basic operations

  start
      -- Start the main loop
    do
      from
        fetch (buffer)
      until
        is_stopped
      loop
          -- Do something, e.g.
        print (last_item)

        fetch (buffer)
      end
    end

feature -- Implementation

 fetch (buf: separate BUFFER)
      -- Get the next item from 'buf'.
    require
      not buf.is_empty or buf.is_stop_requested
    do
      if buf.is_stop_requested then
        is_stopped := True
      else
        last_item := buf.item
        buf.remove
      end
    end
end
```

Listing 4: *Breaking out of a wait condition.*

This allows a consumer to leave the wait condition even when the buffer is empty. The drawback of this approach is that it clutters the application code with some additional if-else constructs, but it is often possible to hide them in a `fetch` function, as shown in our example.

5 Library

5.1 Goals

The goal of the library is to provide a set of classes that simplify programming in SCOOP. Specifically, we want to provide implementations for common concurrency patterns like the worker pool. The result should be a new Eiffel library similar to the standard concurrency libraries in Java [5] or C# [4].

The library was developed with the following design goals:

Safety Avoid common SCOOP pitfalls like deadlocks, starvation of a processor or unintentional lock passing.

Convenience Shield the user from having to write many little "wrappers", i.e. features that just lock an object for a single separate call.

Performance Reduce the overhead of thread creation, especially for code that deals with a lot of small separate objects.

5.2 Concepts

This section describes two core concepts of the library: Import and Separate Proxies. The import concept deals with the problem of how to pass data from one processor to another. It is useful to achieve the performance and to some extent the safety design goal in Section 5.1.

The Separate Proxy [SP] is a pattern to hide separate references behind a proxy object. It provides a solution to the convenience design goal.

5.2.1 Import

The import concept is a central part of the library. It was developed to let users choose between two object passing strategies, namely the Data Processor and the Import approach (see Section 4.1).

The main class is *CP_IMPORT_STRATEGY*, which has the simple interface:

```
deferred class interface
  CP_IMPORT_STRATEGY [G]

feature -- Status report

  is_importable (object: separate G): BOOLEAN
      -- Is 'object' importable?

feature -- Duplication

  import (object: separate G): separate G
      -- Import 'object'.
    require
      importable: is_importable (object)

end
```

Listing 5: *The deferred class CP_IMPORT_STRATEGY.*

The class has two descendants. `CP_NO_IMPORTER [G]` can be used for the Data Processor strategy. It just perform a reference copy of the object. The class `CP_IMPORTER [G]` on the other hand narrows the return type of `import` to a non-separate `G`, meaning that it actually performs an import.

As there's no general-purpose import available in SCOOP at the moment users have to implement their own import features for every class that needs this facility. Descendants of `CP_IMPORTER` simplify this task and provide predefined implementations for some standard classes such as `STRING`. Those mechanisms are described in detail in Section 6.1.

Components that want to make use of the import mechanism need an instance of `CP_IMPORT_STRATEGY` on the same processor. There are several ways how this object can be supplied to a library component. The most obvious solution - passing it as an argument in a constructor - has a big drawback in the SCOOP world: It is impossible to instantiate the component on a separate processor without having to write an extra factory class.

A better solution is to exploit the constrained genericity mechanism in Eiffel. A component that needs to import objects has to declare an additional generic argument for the import strategy. A user can then decide on the precise semantics of the import strategy by just declaring the right type.

The constraint placed on the generic argument is that it needs to be a descendant of `CP_IMPORT_STRATEGY` and that it needs to declare `default_create` as a creation procedure. The latter is not a big restriction in practice, as there are usually no attributes in an importer anyway.

A typical class header of a component using the import concept looks like this:

```
class BUFFER [G, IMPORTER ->
        CP_IMPORT_STRATEGY [G] create default_create end]

feature -- Access

 item: like {IMPORTER}.import

end
```

Listing 6: *An example component with import.*

This small code sample shows another neat little feature of Eiffel. The **`like`** statement fixes the type based on the chosen import strategy, i.e. **`separate`** `G` for `CP_NO_IMPORTER` and non-separate for descendants of `CP_IMPORTER`. This makes the handling of imported objects a lot easier.

5.2.2 Separate Proxy

The Separate Proxy pattern [SP] simplifies access to a separate reference by providing a processor-local proxy object wich forwards all requests to the actual object. The main advantage is that clients don't need to write extra "wrapper" feature to control the separate object. It is applied to all classes in the library which are meant to be shared among processors, i.e. which are usually accessed through a separate reference.

The pattern consists of three classes:

Protégé The actual business class whose objects are usually shared.

Helper A class that provides wrapper functions to access a separate protégé. The helper class is usually ending on `_UTILS`.

Proxy A proxy class with a similar interface as the protégé class, usually ending on `_PROXY`. The proxy forwards every call to its protégé, using the helper class.

It is possible to add a fourth, deferred class that just defines a common interface for the protégé and the proxy. There's an inconsistency however: All preconditions in the protégé class that reference **`Current`** (explicitly or implicitly) need to be weakened (i.e. **`require else True`**) in the proxy, and turned into wait conditions in the helper class. Furthermore, not all features in the business class may be necessary in the proxy, and the proxy itself may add some more features such as compound actions.

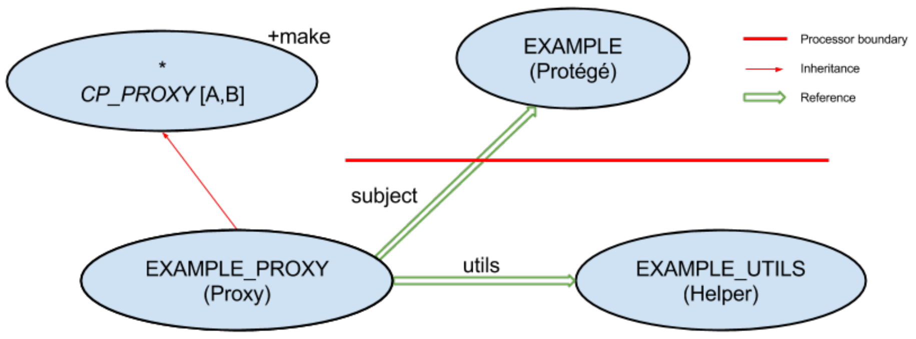

Figure 1: *The class relations in the Separate Proxy pattern.*

Unfortunately this pattern cannot be turned into a reusable module, because it is highly dependent on the precise interface of the protégé class. There is some support in the library however: `CP_PROXY` defines a creation procedure and the attributes `subject` for a separate protégé object and `utils` for a helper object.

Appendix A provides a general recipe on how to implement a separate proxy for an arbitrary protégé class.

5.3 Module overview

The library consists of several modules which implement some of the patterns described in the overview (Section 2). The source code of the library is available on GitHub [3]. All file locations in the following section are relative to the root directory of the repository.

One of the most basic modules is the import module in *library/import*. It implements the Import [IMP] pattern and is at the same time one of the core concepts of the library.

The queue module in *library/queue* implements the Prdocuer / Consumer [P/C] pattern. It depends on the import module.

The process module in *library/process* provides skeleton classes for objects with a main loop. It provides implementations for the Active Object [AO], Asynchronous Self-Call [ASC] and Timer: Periodic [TP] patterns.

The worker pool module in *library/worker_pool* implements the Worker Pool [WP] pattern. It uses the import, queue and process module.

The directory *library/promise* contains the promise module. This module provides classes to monitor and interact with an asynchronous operation.

The executor module resides in *library/executor* and provides an implementation to the Executor Framework [EF], a specialized worker pool and part of the implementation to the Future pattern [FUT].

The implementation of the future pattern is highly intertwined with other parts of the library. It makes use of the promise, executor and worker pool modules and introduces only two classes on its own: *CP_COMPUTATION* and *CP_FUTURE_EXECUTOR_PROXY*.

The class *CP_DELAYED_TASK* in *libary/util* implements the Timer: Invoke Later pattern [TIL]. The same directory also contains the class *CP_EVENT*, which implements the Publish / Subscribe pattern [P/S] in SCOOP.

6 Library modules

6.1 Import

The import module substitutes the SCOOP *import* feature, a built-in mechanism that is unfortunately not implemented at the moment. The basic concepts and ideas behind the module are described in Section 5.2.1. This section only deals with the class *CP_IMPORTER* and its descendants.

The class *CP_IMPORTER* [*G*] has a single deferred feature *import*. It does not provide a generic import mechanism. To write an importer for an arbitrary type, e.g. *STRING*, a client needs to write a new class, inheriting from *CP_IMPORTER* [*STRING*], and implement the deferred feature.

Although the library has a few predefined importers, writing an extra class for every user-defined type may quickly become tedious. Therefore the library provides another way of using the import module with *CP_IMPORTABLE*:

```
deferred class
  CP_IMPORTABLE

feature {CP_DYNAMIC_TYPE_IMPORTER} -- Initialization

  make_from_separate (other: separate like Current)
      -- Initialize 'Current' with values from 'other'.
    deferred
    end

end
```

Listing 7: *The deferred class CP_IMPORTABLE.*

Users can inherit from *CP_IMPORTABLE* and define the import function right inside their class.

There are two predefined importers which can be used for *CP_IMPORTABLE* objects:

- *CP_DYNAMIC_TYPE_IMPORTER*
- *CP_STATIC_TYPE_IMPORTER*

The latter uses constrained genericity to create an object of type *G*. The approach is pretty simple and fast but it has the drawback that the result type is always the static type *G*, even if the argument to provided to the `import` feature was of a subtype of *G*.

The *CP_DYNAMIC_TYPE_IMPORTER* on the other hand respects the dynamic type of its argument. With the help of reflection it creates a new, uninitialized object of the correct type and then calls `make_from_separate` to perform the initialization. This introduces a new problem with respect to void safety however.

As opposed to the static type importer, the feature `make_from_separate` doesn't need to be a creation procedure. This in turn means that the compiler will not check if every attribute is correctly initialized. It is therefore strongly advised to declare `make_from_separate` as a creation procedure for every descendant of *CP_IMPORTABLE*.

Another problem of the *CP_DYNAMIC_TYPE_IMPORTER* is the invariant of an object. There's a short time interval between the creation of an object (using reflection) and the call to `{CP_IMPORTABLE}.make_from_separate` where the invariant is broken. Due to this it is impossible to use classes with invariants in conjunction with the dynamic type importer.

In the future, there will hopefully exist an import routine natively supported by the SCOOP runtime. In that case *CP_IMPORTER* can be made effective and use the native import, and all its descendants will become obsolete.

6.2 Queue

The queue module provides the class *CP_QUEUE* and some support classes that implement the Separate Proxy pattern [SP]. The module can be used for the Producer / Consumer pattern [P/C].

The main challenge in the queue module is data migration, as described in Section 4.1. Therefore the module makes heavy use of the import concept. This means that, along with a generic argument for the data type, it is also necessary for clients to provide a *CP_IMPORT_STRATEGY*. The import strategy basically "teaches" the queue how to import a given object.

Internally the class *CP_QUEUE* uses an *ARRAYED_QUEUE* to store its elements.

6.3 Process

The process module provides a set of classes that implement a skeleton for a main loop with a deferred body.

The class *CP_PROCESS* defines the interface. It is a descendant of the class *CP_STARTABLE*, which means that clients have a simple way to start a separate process using *CP_STARTABLE_UTILS*.

Users need to implement the feature `step`, which should contain the body of the loop. The feature `start` is used to start the loop, and it can be terminated by setting the attribute `is_stopped` to **`True`**.

CP_PROCESS also introduces the two methods `setup` and `cleanup`. They are called in the beginning or at the end of the main loop, and must be explicitly redefined by descendants if needed.

There are two techniques to implement the main loop itself. The first technique, used by *CP_CONTINUOUS_PROCESS*, is pretty staightforward:

```
from setup
until is_stopped
loop
  step
end
```

This approach is very simple and fast. The problem however is that other processors never get a chance to access the `CP_CONTINUOUS_PROCESS` unless the main loop is exited completely. This class is an implementation of the Active Object pattern [AO].

`CP_INTERMITTENT_PROCESS` uses a different technique. The basic idea is to perform only one iteration, and then ask another processor to invoke the loop body again in **`Current`**. This ping-pong approach ensures that other processors get a chance to access and modify data in `CP_INTERMITTENT_PROCESS` after each iteration. In practice this is particularly useful to stop the process from the outside.

`CP_INTERMITTENT_PROCESS` implements the Asynchronous Self-Call pattern [ASC]. The callback service is provided by the class `CP_PACEMAKER`. Every `CP_INTERMITTENT_PROCESS` automatically creates an associated pacemaker.

The `CP_PERIODIC_PROCESS` allows to add small delays between executions. It is a descendant of `CP_INTERMITTENT_PROCESS` and an implementation of the Timer: Periodic pattern [TP]. The class also introduces the simple command `stop`, which can be used to stop the timer.

6.4 Worker pool

The worker pool module provides an implementation for the pattern of the same name. The intention of the Worker Pool pattern [WP] is to exploit the performance advantages of parallel execution while avoiding the overhead of expensive thread creation.

The main component of a worker pool is a shared buffer where clients can insert tasks to be executed. A set of worker threads then continuously retrieve tasks from the buffer and execute them.

The representation of a task varies between different languages. In Java for instance a Runnable object is used, whereas in C# the task is represented as a delegate. SCOOP however has to deal with the problem of object migration, as described in Section 4.1.

If the task object is created on its own processor, as in the Data Processor approach, the performance advantage of the worker pool cancels out. With the Lock Passing approach a task object will be executed on the processor that created the object, which makes the worker pool useless (not to mention the risks of starvation if applied wrong). This only leaves the import mechanism as a sensible solution.

The library supports two flavors of a worker pool. The first and more basic one leaves the choice on how to represent a task object open to the user through a generic argument. The second solution on the other hand defines a new class to encapsulate an operation. It is described in Section 6.6.

The basic worker pool module has three important classes:

- `CP_WORKER_POOL`

- *CP_WORKER*
- *CP_WORKER_FACTORY*

The *CP_WORKER_POOL* provides the shared buffer and some additional functionality to adjust the pool size. The type of the task object alongside its import strategy can be specified with generic arguments. *CP_WORKER_POOL* inherits from *CP_QUEUE* and therefore uses the import concept too.

The deferred class *CP_WORKER* corresponds to the worker thread in other languages. Users need to implement the feature *do_run*, which receives a task object and executes some operation on it. The exact type of the task object depends on the generic arguments of *CP_WORKER*, which must be the same as in *CP_WORKER_POOL*. The non-deferred part of *CP_WORKER* is the main loop itself, which fetches a new task, calls *do_run*, and checks if the worker has to terminate.

The last class, *CP_WORKER_FACTORY*, just provides a deferred factory function for a new worker. The factory class is necessary because the exact type of *CP_WORKER* is not known to the library in advance. *CP_WORKER_POOL* uses the factory to create new workers on demand.

An important functionality of a worker pool is to adjust the number of workers. Increasing the worker count is easily done by just creating new instances of *CP_WORKER*. To decrease the amount of workers however the module needs to apply the processor termination technique described in Section 4.3.

The Separate Proxy pattern [SP] is applied to *CP_WORKER_POOL* to make the handling of a separate worker pool object more convenient.

The basic worker pool module is very flexible. It is for example possible to use it just as an advanced producer / consumer module, where consumers are automatically created and destroyed. The drawback however is that clients need to implement two classes, the worker and the factory, to make use of the module. Section 6.6 therefore introduces a more specialized version of the worker pool which can be used to execute arbitrary operations.

6.5 Promise

The promise module contains a set of classes which can be used to monitor the state of an asynchronous operation. It is mostly used in conjunction with the executor or future module.

The main class is *CP_PROMISE*, which defines queries like *is_terminated* or *is_exceptional*. It also defines the interface to cancel an operation or to get the progress percentage (e.g. for a download task), but these mechanisms need to be supported by the asynchronous operation as well.

The Separate Proxy [SP] is available for promise objects because they are usually declared separate to the client. In this case the pattern is implemented with four classes, i.e.

- *CP_PROMISE* defines a common interface,
- *CP_SHARED_PROMISE* defines the actual separate object,
- *CP_PROMISE_UTILS* has features to access a **separate** *CP_PROMISE* and
- *CP_PROMISE_PROXY* implements the proxy object.

There's an important descendant, the `CP_RESULT_PROMISE`, which is used for asynchronous operations returning a result. It also has a set of associated classes that implement the Separate Proxy pattern.

The `CP_RESULT_PROMISE` contains a query `item` to retrieve the result as soon as it's available. A distinguishing feature of this query is that it blocks if the result is still being computed.

The return type of `item` depends on a generic argument. To move the result back to the client the class makes use of the import concept. This means that both `CP_SHARED_RESULT_PROMISE` and `CP_RESULT_PROMISE_PROXY` have an additional generic argument which defines the import strategy.

6.6 Executor

The executor module defines an interface for executing arbitrary tasks. The implementation of the execution service may vary. In most cases it is a worker pool, but it is also possible to use a single thread or to execute the task synchronously in the current thread.

The representation of a task object in Java is a Runnable object, or a delegate in C#. A SCOOP implementation also needs a class to represent an task, but with an important additional requirement: The task objects have to be importable.

The agent classes in Eiffel (i.e. `ROUTINE` and descendants) may be used to represent operations, but they can't be easily imported. Therefore we added a new, deferred class `CP_TASK` to represent an importable asynchronous operation. It also adds some additional functionality like exception handling or the ability to attach a promise object (see Section 6.5). To define a new task clients need to inherit from `CP_DEFAULT_TASK` and implement the two features `run` and `make_from_separate`.

The interface to execute tasks is provided by the class `CP_EXECUTOR`. It defines the feature `put` which takes a **`separate`** `CP_TASK` object as its argument. As an executor instance is usually placed on its own separate processor we applied the Separate Proxy pattern [SP] on `CP_EXECUTOR`.

The executor framework is pretty useless on its own, as it essentially consists of only two deferred classes. Therefore it is shipped with a worker pool implementation. The `CP_TASK_WORKER_POOL` implements the executor interface and is itself a descendant of the more basic `CP_WORKER_POOL`. The associated `CP_TASK_WORKER` then just fetches `CP_TASK` objects and executes them.

In some cases it may also be possible to use agents in conjunction with the executor module. The library has implemented a mechanism to import agents with the help of reflection. The restrictions imposed on the agent are quite harsh however, e.g. it only works with basic expanded or truly separate closed arguments. The functionality is provided by `CP_AGENT_TASK` (and `CP_AGENT_COMPUTATION` for the future module, see Section 6.7) in *library/agent_integration*.

6.7 Futures

The Future pattern [FUT] is used to perform a computation asynchronously. Instead of computing a value straight away, the computation is wrapped into an object and the user receives a handle to retrieve the value as soon as it is ready. This handle is often called future, promise or delay. In this section we'll use the term future for the whole pattern, and promise only refers to the handle.

The main advantage of the future pattern is that it allows to make use of parallelism in an easy way. Users just have to spot computations which may run asynchronously, and the future pattern then takes care of thread management, synchronization and result propagation.

The pattern consists of four building blocks:

- The promise,
- the computation,
- the execution service,
- and a "frontend" object which takes a computation, submits it to the executor, and returns a promise.

The promise object is defined by *CP_PROMISE* and its descendants. The detailed implementation is described in Section 6.5.

The representation of the computation is a Callable object in Java and a delegate in C#. Our library uses the class *CP_COMPUTATION* with the deferred feature *computed*. It is a descendant of *CP_TASK* introduced in Section 6.6.

Due to the fact that *CP_COMPUTATION* inherits from *CP_TASK* we can just use the executor module (see Section 6.6) as the execution service for the future pattern.

The "frontend" part is provided by the two classes *CP_EXECUTOR_PROXY* and *CP_FUTURE_EXECUTOR_PROXY*. This is an example for a separate proxy where the responsability has been expanded: Instead of just forwarding the computation object to the execution service, it also initializes the promise and returns it to the user.

The implementation of the future pattern hits two challenges:

- Object Migration (see Section 4.1): Operations can't be easily moved from the client to an execution service. The same is also true for the result of a computation in the reverse direction.
- Processor Communication (see Section 4.2): The promise object should neither be placed on the client processor nor on the executor service.

The first problem is already solved by the executor module. Just like a *CP_TASK* object, a *CP_COMPUTATION* is movable across processor boundaries. To bring the result back to the client the *CP_RESULT_PROMISE* also makes use of the import concept.

The second problem is more interesting however. As we've seen in Section 4.2, the promise object needs to be placed on its own processor. However, starting a new processor for every computation introduces a huge overhead.

A better tradeoff would be to create one global processor which takes care of all promise objects. This may introduce some contention if multiple futures are submitted at the same time, but we think that this is acceptable.

The global processor approach brings another problem though. A promise object has two generic arguments for the return type and the import strategy. As these arguments are not known in advance, and because SCOOP processor tags [17, p. 90] are not implemented yet, it is impossible to create a promise object on this dedicated processor.

The solution is - surprisingly - the import concept. We can create a "template" promise object with the correct types on the client processor, and then ask the global processor to import it. That way the promise ends up on the correct processor.

7 Evaluation

To evaluate the library we did a small performance benchmark. We implemented the Gaussian elimination algorithm in three different ways: sequentially, with SCOOP only, and using the future module (see Section 6.7) from our library. We chose to use the future module because it indirectly also measures many other parts of the library, like the worker pool or import mechanism.

We ran the tests with randomly generated square matrices. The matrix dimensions were always a power of two in the range from 32 to 1024. Additionally, there was one more column for the result vector in the system of linear equations. Each test was repeated 5 times. The test system was a quad-core AMD Phenom II X4 955 processor with 6 GB of RAM.

The results are shown Table 1.

Matrix Size	Future	Raw SCOOP	Sequential
32	0.36	0.19	<0.01
64	1.67	1.11	<0.01
128	8.45	9.27	0.04
256	26.45	66.56	0.33
512	102.28	515.11	2.62
1000	-	3937.2	-
1024	424.32	error	20.79

Table 1: *Average time in seconds for different algorithms.*

We can get several observations from the results:

- The raw SCOOP solution fails for the matrix with dimensions 1024x1024. This is a known bug [6]: The algorithm uses more than the maximum number of processors. The library solution doesn't suffer from this problem because it's using a fixed amount of processors. To nevertheless get a comparable result the test was repeated with a slightly smaller matrix.
- Futures are faster than raw SCOOP for large data sets.
- For smaller data sets, raw SCOOP beats the library.
- Sequential execution is a lot faster than SCOOP.

The last observation is probably the most fundamental. The SCOOP runtime really needs to be improved in order to make it competitive to threaded systems, or even sequential ones. Fortunately an improved version [22] is being developed at the time of writing. It will probably be integrated into a future EiffelStudio release.

Another improvement which might be useful for the library is the Passive Processor concept [16]. If both the worker pool and the promise processor could be declared as passive, the computation might speed up a lot.

8 Conclusion

Concurrent programming is increasingly becoming the norm despite its difficulties. The SCOOP model provides a solid foundation to concurrent programming in Eiffel, but it is hard to learn for developers due to the paradigm shift and the lack of a concurrency library.

In this thesis we've worked out many methods that simplify concurrent programming in SCOOP. We did a broad survey of popular concurrency patterns and present our findings in a detailed list. The list can be used to search for a pattern that fits a particular problem, which may even be useful to programmers of threaded languages.

From our survey we selected some patterns which we thought to be especially useful. The selection was based on the study of other concurrency libraries as well as some input from the Software Engineering research group at ETH. The seleted patterns were then implemented and are now available as a new Eiffel library. Besides the pattern implementations, the library also provides some workarounds for current SCOOP limitations, such as the missing import statement.

Performance measurements for the Future pattern showed that the library is actually faster for large data sets and uses less threads than the native SCOOP approach.

Finally, we also described some challenges when programming in SCOOP for the first time and how to solve them. This is especially useful to developers experienced in thread programming who want to try SCOOP out.

8.1 Future work

The library provides several opportunities for future work.

More patterns The library can be extended with additional patterns. It may be useful to include Pipeline or Dataflow Network.

Separate Proxies The Separate Proxy pattern may be applied to some EiffelBase classes, such as `ARRAYED_LIST`, `HASH_TABLE` or `ROUTINE`.

Separate Proxy Wizard Writing a Separate Proxy is tedious. Most of it could be automated with a wizard however.

Concurrent Data Structures Sometimes it may be useful to have truly concurrent data structures for performance reasons. The Array Slicing technique [18] is an example how arrays with concurrent read access can be implemented in SCOOP.

SCOOP itself also provides a rich offering of possible improvements.

Faster runtime The SCOOP runtime needs to become faster. This is currently being developed [22].

Native Import A native SCOOP import mechanism is a great tool to deal with a lot of small objects. It will also make the library API and implementation simpler, as the manual import workaround can be removed.

Passive Processors The passive processors concept [16] could be integrated into EiffelStudio. It can make a big performance improvement to situations where one needs to pass data from one processor to another.

Separate references The handling of separate references should become more convenient. At the moment programmers are forced to write a lot of small, annoying features to perform separate calls. Some syntactic sugar would be really helpful.

Appendices

A How-To: Separate Proxy

This appendix shows how to implement a separate proxy based on a small class *EXAMPLE*.

```
class interface
  EXAMPLE [G]

feature -- Status report

  is_available: BOOLEAN
      -- Is 'item' available?

feature -- Access

  item: separate G
      -- Item in 'Current'.
    require
      available: is_available

feature -- Element change

  put (a_item: separate G)
      -- Set 'item' to 'a_item'.

end
```

Listing 8: *The example class (protégé) where the separate proxy should be applied.*

First we need to create the helper class. This is done according to these rules:

- The name should end in *_UTILS*, i.e. *EXAMPLE_UTILS*.
- The generic arguments are the same as in *EXAMPLE*.
- Any feature to access the separate *EXAMPLE* object should be prefixed with *example_*. This helps to avoid name clashes if someone wants to inherit from *EXAMPLE_UTILS*.
- The first argument of each feature is *example*: **separate** *EXAMPLE* [*G*]. All other arguments are the same as the ones in the corresponding feature in *EXAMPLE*.
- Preconditions in *EXAMPLE* should be rewritten as wait conditions with the same meaning in *EXAMPLE_UTILS*.
- If there's a non-expanded return type to a feature, you can decide if it should be declared separate in *EXAMPLE_UTILS* or if it should be imported.

```
class
  EXAMPLE_UTILS [G]

feature -- Access

  example_item (example: separate EXAMPLE [G]): separate G
      -- Get the item from 'example'.
      -- May block if not yet available.
    require
      available: example.is_available
    do
      Result := example.item
    end

feature -- Element change

  example_put (example: separate EXAMPLE [G];
      item: separate G)
      -- Put 'item' into 'example'.
    do
      example.put (item)
    end
end
```

Listing 9: *The helper class for a separate EXAMPLE.*

In this example we also dropped the feature `is_available`, because it's not considered to be important for separate clients.

The proxy class also has some simple rules:

- The name should be `EXAMPLE_PROXY`.
- The generic arguments are the same as in `EXAMPLE`.
- Inheriting from `CP_PROXY [EXAMPLE [G], EXAMPLE_UTILS [G]]` is recommended. That way one can avoid having to write the creation procedure `make`.
- The feature names and arguments are the same as in `EXAMPLE`.
- Preconditions in `EXAMPLE` are not present in `EXAMPLE_PROXY`. The class `EXAMPLE_UTILS` defines them as wait conditions instead.
- Every feature body makes use of `utils` to forward its requests to the `subject`.

```
class
  EXAMPLE_PROXY [G]

inherit
  CP_PROXY [EXAMPLE [G], EXAMPLE_UTILS [G]]

create
  make

```

```
feature -- Access

  item: separate G
      -- Item in the example object.
      -- May block if not yet available.
    do
      Result := utils.example_item (subject)
    end

feature -- Element change

  put (a_item: separate G)
      -- Set `item' to `a_item'.
    do
      utils.example_put (subject, a_item)
    end

end
```

Listing 10: *The proxy class for a separate EXAMPLE.*

B SCOOP and EiffelVision

At the core of most GUI toolkits is an event dispatching thread (EDT). The EDT is running an infinite loop where it listens for user input events.

Only the EDT is allowed to modify user interface objects. Background threads therefore have to enqueue agents to be executed by the EDT if they want to update the GUI.

EiffelVision also follows this design with the event dispatching code defined in `EV_APPLICATION` and the enqueuing feature `do_once_on_idle` in the same class. Due to this design one might think that combining SCOOP with EiffelVision is impossible.

This is not true however. EiffelVision implements a variation of the Asynchronous Self-Call pattern [ASC]. The body of the loop is defined by the feature `{EV_APPLICATION_I}.process_event_queue`, whereas the actual loop is implemented in `EV_APPLICATION_HANDLER`. Therefore, a user interface object can subscribe to events from separate processors in the same manner as it can subscribe to events from the same processor.

In fact the SCOOP model makes GUI programming a lot easier. In threaded languages programmers constantly have to worry that only the EDT manipulates user interface objects. SCOOP however gives this guarantee for free. This completely eliminates a source of randomly occurring errors which are usually very hard to find.

We'll show a small example application which can download a file in the background. The application has a simple graphical user interface and uses the executor module from the library. Some highlights of the example are event propagation and the cancellation mechanism. The full source code can be found in *examples/eiffelvision_downloader* in the repository [3].

Let's first look at the business logic. The class `DOWNLOAD_TASK` in Listing 11 defines the background downloader. It inherits from `CP_DEFAULT_TASK` such that it can be used in conjunction with an executor.

```
class
  DOWNLOAD_TASK

inherit
  CP_DEFAULT_TASK

    -- Initialization omitted.

feature -- Access

  url: STRING

feature -- Basic operations

  run
      -- <Precursor>
    local
      download_fragments: ARRAYED_LIST [STRING]
      http_downloader: detachable HTTP_PROTOCOL
      size: INTEGER
    do
      create download_fragments.make (50)
      create http_downloader.make (create {HTTP_URL}.make(url))

      from
          -- Start the download
        http_downloader.set_read_mode
        http_downloader.open
        http_downloader.initiate_transfer
        size := http_downloader.count
      until
          -- Terminate when the download is finished
          -- or the user cancels the download manually.
        http_downloader.bytes_transferred = size
        or attached promise as l_promise
        and then is_promise_cancelled (l_promise)
      loop
          -- Receive a single packet.
        http_downloader.read
        if attached http_downloader.last_packet as l_packet then
          download_fragments.extend (l_packet)
        end
          -- Update the progress information in the UI.
        if attached promise as l_promise then
          promise_set_progress (l_promise,
            http_downloader.bytes_transferred / size)
        end
      end

        -- Discard result. A real application would
        -- probably write it to a file
      download_fragments.wipe_out
      http_downloader.close
    rescue
      if attached http_downloader as dl and then dl.is_open then
        dl.close
      end
    end
end
```

Listing 11: The background download task.

The code is structured such that the loop body only handles a small chunk of the total download. This allows to check for a cancellation request in regular intervals, and to publish the current progress to the shared promise object.

The rescue clause at the end ensures that the connection is correctly closed. Note that this is the only exception handling which needs to be done, and it's not necessary to "catch" an exception with **retry**. The user interface is still safe though because the exception gets caught later, transformed to an unsuccessful termination event, and the GUI will be informed automatically.

Another major component is the class which defines the main window. It contains a lot of boring EiffelVision initialization code however. Listing 12 therefore only shows the event handling part.

```
class
  MAIN_WINDOW
inherit
  EV_TITLED_WINDOW

    -- Initialization and GUI elements omitted.

feature -- Access

  executor: CP_EXECUTOR_PROXY
    -- An executor to submit background tasks to.

  download_handle: detachable CP_PROMISE_PROXY
    -- A handle to a possible background download.

  formatter: FORMAT_DOUBLE
    -- A formatter for progress values.

feature -- Status report

  is_cancelling: BOOLEAN
    -- Is the download about to terminate?

feature {NONE} -- Button press events

  on_download_button_clicked
    -- Handler for clicks on download button.
   local
    downloader: DOWNLOAD_TASK
    l_promise: CP_PROMISE_PROXY
   do
    if not attached download_handle then

      create downloader.make (url_text.text)
      l_promise := executor.new_promise
```

```
        download_handle := l_promise

        l_promise.progress_change_event.subscribe (agent
            on_progress)
        l_promise.termination_event.subscribe (agent
            on_terminated)

        downloader.set_promise (l_promise.subject)
        executor.put (downloader)
      end
    end

  on_cancel_button_clicked
      -- Handler for clicks on cancel button.
    do
      if
         not is_cancelling and
         attached download_handle as l_download
      then
        l_download.cancel
        is_cancelling := True
        status_text.set_text ("Cancelling download...")
      end
    end

feature {NONE} -- Background download events

  on_terminated (is_successful: BOOLEAN)
      -- Handler for termination events from download task.
    do
      if not is_successful then
        result_text.set_text ("Download aborted.")
      elseif is_cancelling then
        result_text.set_text ("Download cancelled.")
        is_cancelling := False
      else
        result_text.set_text ("Download finished.")
      end

      status_text.set_text ("No download in progress.")
      download_handle := Void
    end

  on_progress (progress: DOUBLE)
      -- Handler for progress change events from download task.
    do
      if not is_cancelling then
        status_text.set_text ("Download progress:" +
            formatter.formatted (progress * 100) + "%%")
      end
    end
end
```

Listing 12: *The main window.*

The most interesting feature is `on_download_button_clicked`. It starts a background task and sets up all event handlers such that the user interface can react to progress change or termination events. The events are predefined in `CP_PROMISE` and sent automatically to all subscribers whenever the associated task changes the progress value or terminates.

The `on_terminated` handler function is used to receive an event that the background task has finished. There are three cases that need to be distinguished: normal termination without cancellation, normal termination with cancellation, and exceptional termination due to an error.

The last component is the class `DOWNLOAD_APPLICATION`.

```
class
  DOWNLOAD_APPLICATION

create make

feature {NONE} -- Initialization

  make
      -- Start the download application example.
    local
      app: EV_APPLICATION
      main_window: MAIN_WINDOW
      worker_pool: separate CP_TASK_WORKER_POOL
    do
        -- Create a worker pool which can be
        -- used to execute background downloads.
      create worker_pool.make (10, 1)
      create executor.make (worker_pool)

        -- Create application and main window.
      create app
      create main_window.make (executor)

        -- Don't forget to tear down the worker pool at the end.
      app.destroy_actions.extend (agent executor.stop)

        -- Start the GUI.
      main_window.show
      app.launch
    end

feature -- Access

  executor: CP_EXECUTOR_PROXY
      -- The worker pool for background tasks.

end
```

Listing 13: *Download application root class.*

The class just creates and launches the event loop, the worker pool and the main window. The most interesting part is how the worker pool can be stopped. As the statement `app.launch` just kick-starts the event loop and then returns we can't destroy the worker pool after this statement as in a sequential program. It is therefore necessary to install a handler to stop the executor when the event loop terminates.

C API tutorial

The library has several independent components which can be used for various tasks in concurrent programming. This tutorial therefore consists of three unrelated sections. Each section describes a concurrency problem and how the library can be used to solve it.

All programming code used in this section originally comes from the example applications in the repository [3].

C.1 Producer / Consumer

The producer / consumer example is pretty common in concurrent programming. At its core is a shared buffer. A producer can add items to the buffer, whereas a consumer removes items from the buffer.

The library class *CP_QUEUE* can be used for the shared buffer. If we want to pass *STRING* objects from producers to consumers, we have to declare the queue like this:

```
class PRODUCER_CONSUMER feature

  make
      -- Launch producers and consumers.
    local
      queue: separate CP_QUEUE [STRING, CP_STRING_IMPORTER]
  -- ...
    do
      create queue.make_bounded (10)
  -- ...
    end
end
```

Note that there are two generic arguments:

- The first argument (*STRING*) denotes the type of items in the queue.
- The second argument (*CP_STRING_IMPORTER*) defines the import strategy (see Section 5.2.1). It teaches the queue how to import a string object.

In our example we decided to import every string object. An alternative would be to use *CP_NO_IMPORTER [STRING]* and deal with separate references instead.

The next step is to define the producer and consumer classes.

```
class
  PRODUCER

inherit
  CP_STARTABLE

```

```
create
  make

feature {NONE} -- Initialization

  make (a_queue: separate CP_QUEUE [STRING, CP_STRING_IMPORTER];
        a_identifier: INTEGER; a_item_count: INTEGER)
      -- Initialization for 'Current'.
    do
      identifier := a_identifier
      item_count := a_item_count
      create queue_wrapper.make (a_queue)
    end

  queue_wrapper: CP_QUEUE_PROXY [STRING, CP_STRING_IMPORTER]
      -- A wrapper object to a separate queue.

  identifier: INTEGER
      -- Identifier of 'Current'.

  item_count: INTEGER
      -- Number of items to produce.

feature -- Basic operations

  start
      -- Produce 'item_count' items.
    local
      i: INTEGER
      item: STRING
    do
      from
        i := 1
      until
        i > item_count
      loop
          -- Note that there's no need to declare 'item' as
          -- separate, because it will be imported anyway.
        item := "Producer: " + identifier.out + ": item " + i.out
        queue_wrapper.put (item)
        i := i + 1
      end
    end

end
```

Listing 14: *The producer class.*

You may notice three things in this example:

- *PRODUCER* inherits from *CP_STARTABLE*.
- The *PRODUCER* uses a *CP_QUEUE_PROXY* instead of *CP_QUEUE*.
- The generated strings are not separate.

The classes *CP_STARTABLE* and *CP_STARTABLE_UTILS* are a useful combination. They allow to start some operation on a separate object without the need for a specialized wrapper function.

CP_QUEUE_PROXY is part of the Separate Proxy pattern [SP] (see Section 5.2.2). It is useful to access a separate queue without having to write a lot of small wrapper functions.

The fact that strings can be generated on the local processor is probably the most interesting observation. Usually it is necessary when using SCOOP to create shared data on a dedicated processor. As we're using the import mechanism however this is not necessary and would even be wasteful.

The consumer is the same as the producer except for the feature *start*:

```
class
  CONSUMER

inherit
  CP_STARTABLE

  -- Initialization omitted...

feature -- Basic operations

  start
      -- Consume 'item_count' items.
    local
      i: INTEGER
      item: STRING
    do
      from
        i := 1
      until
        i > item_count
      loop
        queue_wrapper.consume

        check attached queue_wrapper.last_consumed_item as l_item
             then

            -- Note that 'item' is not declared as separate
          item := l_item
          print (item + " // Consumer " + identifier.out
            + ": item " + i.out + "%N")
        end
        i := i + 1
      end
    end
end
```

Listing 15: *The consumer class.*

Again, there's no need to declare the consumed string as separate, thanks to the import mechanism.

The only thing left now is to create and launch the producers and consumers in the main application. Note that *PRODUCER_CONSUMER* inherits from

CP_STARTABLE_UTILS such that it can use *async_start* to start both the consumer and producer threads.

```
class
  PRODUCER_CONSUMER

inherit
  CP_STARTABLE_UTILS

create
  make

feature {NONE} -- Initialization

  make
      -- Launch the producer and consumers.
    local
      l_queue: separate CP_QUEUE [STRING, CP_STRING_IMPORTER]
      l_producer: separate PRODUCER
      l_consumer: separate CONSUMER
    do
      print ("%NStarting producer/consumer example. %N%N")

        -- Create a shared bounded queue for data exchange.
      create l_queue.make_bounded (queue_size)

        -- Create and launch the consumers.
      across 1 |..| consumer_count as i loop
        create l_consumer.make (l_queue, i.item,
            items_per_consumer)
        async_start (l_consumer)
      end

        -- Create and launch the producers.
      across 1 |..| producer_count as i loop
        create l_producer.make (l_queue, i.item,
            items_per_producer)
        async_start (l_producer)
      end
    end

feature -- Constants

  queue_size: INTEGER = 5
  producer_count: INTEGER = 10
  consumer_count: INTEGER = 10
  items_per_producer: INTEGER = 20
  items_per_consumer: INTEGER = 20

invariant
  equal_values: producer_count * items_per_producer =
      consumer_count * items_per_consumer
end
```

***Listing 16:** The producer / consumer application root class.*

C.2 Server thread

In network server programming it is common to have a dedicated thread listening on a socket. In a SCOOP environment it is not hard to create such a processor, but it is hard to stop it. The main problem is that the server will run an infinite loop, and other processors never get exclusive access to call a feature like `stop`.

The library addresses this issue with *CP_INTERMITTENT_PROCESS*. The class defines a special main loop using the Asynchronous Self-Call pattern [ASC].

To use *CP_INTERMITTENT_PROCESS* you need to inherit from it and implement the deferred feature `step`. The following example defines a simple echo server that just listens on a socket and replies with the same string:

```
class
  ECHO_SERVER

inherit

  CP_INTERMITTENT_PROCESS
    redefine
      cleanup
    end

create
  make

feature {NONE} -- Initialization

  make
      -- Initialization for `Current'.
    do
        -- Create the socket on the specified port.
      create socket.make_server_by_port (2000)
        -- Set an accept timeout.
      socket.set_accept_timeout (500)
        -- Enable the socket.
      socket.listen (5)
    end

feature -- Basic operations

  cleanup
      -- <Precursor>
    do
      socket.cleanup
    end

  stop
      -- Stop the current processor.
    do
      is_stopped := True
    end
  step
      -- <Precursor>
    local
      l_received: STRING
    do
        -- Accept a new message.
      socket.accept

        -- In case of an accept timeout 'accepted' is Void.
      if attached socket.accepted as l_answer_socket then

          -- Read the message.
        l_answer_socket.read_line
        l_received := l_answer_socket.last_string

          -- Generate and send the answer.
        l_answer_socket.put_string (l_received)
        l_answer_socket.put_new_line
        l_answer_socket.close
      end
    end

feature {NONE} -- Implementation

  socket: NETWORK_STREAM_SOCKET
      -- The server network socket.

end
```

***Listing 17:** The echo server class.*

The accept timeout is important in this example. It ensures that the server processor periodically breaks free of its wait condition while listening and therefore has a chance to finish the `step` feature.

The echo server can be started with `{STARTABLE_UTILS}.async_start` and stopped with the feature `stop`. Thanks to the special loop construct used in `CP_INTERMITTENT_PROCESS` stopping the echo server also works when called from another processor.

C.3 Worker Pool and Futures

This section describes how to use a worker pool for I/O tasks. The example application defines two operations: reading a file and appending a string to a text file. The classes to represent these operations are `FILE_APPENDER_TASK` and `FILE_READER_TASK`.

The file reader task is implemented with the future module from the library. The library has the class `CP_COMPUTATION` to represent futures, i.e. asynchronous tasks that return a result. The `FILE_READER_TASK` therefore needs to inherit from `CP_COMPUTATION`.

The file appender task doesn't return a result. Therefore it has to inherit from `CP_DEFAULT_TASK`. This inheritance is necessary to be able to submit it to a worker pool later.

The two classes are shown in Listing 18.

```
class
  FILE_READER_TASK
inherit
  CP_COMPUTATION [STRING]

create
  make, make_from_separate

feature {NONE} -- Initialization

  make (a_path: STRING)
      -- Create a new task to read the content from 'a_path'.
    do
      path := a_path
    end

feature {CP_DYNAMIC_TYPE_IMPORTER} -- Initialization

  make_from_separate (other: separate like Current)
      -- <Precursor>
    do
      create path.make_from_separate (other.path)
      promise := other.promise
    end

feature -- Access

  path: STRING
      -- The path of the file to read from.

feature -- Basic operations

  computed: STRING
      -- <Precursor>
    local
      l_file: PLAIN_TEXT_FILE
      l_content: STRING
    do
      create l_file.make_open_read (path)
      l_file.read_stream (l_file.count)
      Result := l_file.last_string
      l_file.close
    end
end

class
  FILE_APPENDER_TASK
inherit
  CP_DEFAULT_TASK

    -- Initialization similar to FILE_READER_TASK.

feature -- Access

```

```
 path: STRING
   -- The path of the file to write to.

 content: STRING
   -- The content to be written.

feature -- Basic operations

 run
   -- <Precursor>
  local
   l_file: PLAIN_TEXT_FILE
  do
   create l_file.make_open_append (path)
   l_file.put_string (content)
   l_file.close
  end
end
```

Listing 18: *The file reader and appender classes.*

The main algorithm needs to be defined in `computed` or `run`, respectively. Additionally, the feature `make_from_separate` has to be implemented. This feature is required to import task objects from the client to the worker pool processor (see Section 5.2.1 for the import concept).

`CP_EXECUTOR` defines an interface to submit and execute task objects. The most important implementation is `CP_TASK_WORKER_POOL`. The executor class is shipped with a Separate Proxy [SP], which means that clients can access it via `CP_EXECUTOR_PROXY` or `CP_FUTURE_EXECUTOR_PROXY`. These two proxy classes also initialize the promise object, which is a handle to the asynchronous operation that can be used to wait for its termination or to retrieve the result when it's available.

Listing 19 shows how a worker pool is used to submit the previously defined file reader and appender tasks.

```
class
 IO_WORKER_POOL

create
 make

feature -- Constants

 path: STRING = "a.txt"

 hello_world: STRING = "Hello World%N"

feature {NONE} -- Initialization

 make
   -- Initialization for 'Current'.
  do
    -- Create the worker pools.
   create worker_pool.make (100, 4)
```

```
      create executor.make (worker_pool)

        -- Run the example
      single_read_write

        -- Stop the executor. This is necessary such that
        -- the application can terminate.
      executor.stop
    end

feature -- Basic operations

   single_read_write
      -- Perform a single write operation on a file.
    local
      write_task: FILE_APPENDER_TASK
      write_task_promise: CP_PROMISE_PROXY

      read_task: FILE_READER_TASK
      read_task_promise: CP_RESULT_PROMISE_PROXY [STRING,
          CP_STRING_IMPORTER]

      l_result: detachable STRING
    do
        -- Execute a file append task first.
      create write_task.make (path, hello_world)

        -- Submit the task and get a promise object.
      write_task_promise := executor.put_with_promise(write_task)

        -- Wait for the task to finish.
      write_task_promise.await_termination

        -- It is possible to check for IO exceptions.
      check no_exception:
        not write_task_promise.is_exceptional
      end


        -- Now execute a read task.
      create read_task.make (path)

        -- Submit the task and get a promise.
      read_task_promise := executor.put_future (read_task)

        -- The promise can be used to retrieve the result.
        -- Note that the statement blocks if the result
        -- is not ready yet.
      l_result := read_task_promise.item

        -- Check if the read-write cycle worked as expected.
%      check correct_result: l_result ~ hello_world end
    end

```

```
feature {NONE} -- Implementation

 worker_pool: separate CP_TASK_WORKER_POOL
    -- The worker pool to execute tasks.

 executor: CP_FUTURE_EXECUTOR_PROXY[STRING, CP_STRING_IMPORTER]
    -- The executor proxy used to submit tasks.

end
```

Listing 19: *Using a worker pool for futures and asynchronous tasks.*

Submitting tasks to the executor and dealing with the asynchronous result is pretty straightforward. Some calls to the promise object are blocking however, e.g. *await_termination* or *item*, if the asynchronous task has not finished yet.

The library ensures that no exception can escape the task object and crash the worker pool or the client code. Clients can check if an exception happened with the query *is_exceptional* on the promise object. The exception trace, if any, is also available to the client with the query *last_exception_trace*.

An important measure is to stop the executor when the application wants to terminate. Otherwise the workers will continue to wait for incoming tasks, preventing the process to shut down.

References

[1] Borealis distributed stream processing. http://cs.brown.edu/research/borealis/public/.

[2] Eiffel ECMA-367 Standard. http://www.ecma-international.org/publications/standards/Ecma-367.htm.

[3] Github code repository. https://github.com/romasch/scoop_patterns.

[4] Microsoft Task Parallel Library. http://msdn.microsoft.com/en-us/library/dd460717(v=vs.110).aspx.

[5] Oracle Java concurrency package. http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html.

[6] Scoop: Known limitations. https://docs.eiffel.com/book/solutions/scoop-implementation.

[7] Scoop: Official website. http://docs.eiffel.com/book/solutions/concurrent-eiffel-scoop.

[8] J. Bloch J. Dowbeer D. Holmes D. Lea B. Goetz, T. Peierls. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[9] H. Rohnert H. Buschmann D. Schmidt, M. Stal. *Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects*. Wiley, 2000.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[11] Alexey Kolesnichenko, Sebastian Nanz, and Bertrand Meyer. How to cancel a task. In *Proceedings of the 2013 International Conference on Multicore Software Engineering, Performance, and Tools (MUSEPAT'13)*, Lecture Notes in Computer Science, pages 61–72. Springer, 2013.

[12] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, second edition, 2000.

[13] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 227–242, New York, NY, USA, 2009. ACM.

[14] J. Reinders M. McCool, A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann, 2012.

[15] B. Meyer. *Touch of Class*. Springer, 2009.

[16] Benjamin Morandi, Sebastian Nanz, and Bertrand Meyer. Safe and efficient data sharing for message-passing concurrency. In *Proceedings of the 16th International Conference on Coordination Models and Languages (COORDINATION 2014)*, volume 8459 of *Lecture Notes in Computer Science*, pages 99–114. Springer, 2014.

[17] Piotr Nienaltowski. *Practical Framework for Contract-based Concurrent Object-oriented Programming*. PhD thesis, ETH Zürich, 2007.

[18] Mischael Schill, Sebastian Nanz, and Bertrand Meyer. Handling parallelism in a concurrency model. In *Proceedings of the 2013 International Conference on Multicore Software Engineering, Performance, and Tools (MUSEPAT'13)*, volume 8063 of *Lecture Notes in Computer Science*, pages 37–48. Springer, 2013.

[19] Douglas C Schmidt, Carlos O'Ryan, Michael Kircher, Irfan Pyarali, et al. Leader/Followers - a design pattern for efficient multi-threaded event demultiplexing and dispatching. In *University of Washington. http://www.cs.wustl.edu/s̃chmidt/PDF/lf.pdf*. Citeseer, 2000.

[20] Martin Thompson, Dave Farley, Michael Barker, Patricia Gee, and Andrew Stewart. Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads, 2011.

[21] S. Toub. Patterns for parallel programming, 2010. Microsoft Corporation.

[22] Scott West. *Correctness and Execution of Concurrent Object-Oriented Programs*. PhD thesis, ETH Zürich, 2014.