Chair of
Software Engineering

Reproducible executions of SCOOP programs

Technical Report

By: Andrey Nikonov
Andrey Rusakov

Supervised by: Benjamin Morandi
Dr. Sebastian Nanz
Scott West
Prof. Dr. Bertrand Meyer

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

inf
Informatik
Computer Science

Abstract

SCOOP (Simple Concurrent Object-Oriented Programming) [2, 3, 4] is a model and practical framework for building concurrent applications. It comes as a refinement of the Eiffel [7] programming language and has been integrated into the research version of EiffelStudio, called EVE [8].

Many examples have proven SCOOP's simplicity [3], but the development of concurrent programs in SCOOP can still be troubled by problems. One of the main difficulties is debugging of concurrent programs because the run-time behavior, in particular context switching, depends on the operating system scheduler. To handle this problem for general concurrent programs we have to capture all types of synchronization events or have to modify the operating system scheduler, which is impossible for commercial system. The SCOOP program model allows to alleviate this problem because an intermediate scheduler and a simple synchronization model is used.

We developed and implemented a record/replay technique to help users to recognize concurrent bugs in SCOOP programs and provided tools for debugging. We demonstrated its functionality by applying to an example program with deadlocks.

Also we provided a tool for visualizing SCOOP program behavior, particularly, SCOOP processors structure, separate calls sequences and locked processors lists for each separate call.

Keywords

Concurrent programming, SCOOP, Eiffel, debugging, record and replay of concurrent programs.

Contents

1 Introduction

Testing is the most important technique to ensure the production of quality software in industry, but is not applied to concurrent programs in a satisfactory way. Since the nondeterministic execution of concurrent programs leads to very large state spaces and subtle variations in run-time behavior, testing in a concurrent setting has to be based on repeatable execution schedules, which should be generated in a systematic way to cover the program's state space. Without such a technique, program errors that are discovered are difficult to track down and to reproduce, making debugging a nightmare.

In this paper we present a technique and software for recording and replaying concurrent programs runtime behavior written in SCOOP, based on the idea of controlling execution schedules. In other words these techniques make it possible to eliminate the non-determinism of concurrent execution. We also provide a tool for saving and visualizing the SCOOP program model. The tool allows plotting a separate call graph, SCOOP processors relationships and locked processors lists.

Using these tools together you can record, replay and display unexpected program runtime behavior and observe undesirable sequences of separate calls for debugging.

The main goal of this work is to develop and implement SCOOP program debugging techniques and demonstrate their benefits by applying them to SCOOP programs with bugs. As main problems we consider deadlocks and some rarely arising bugs depending on scheduler’s work.

The rest of the paper is organized as follows: Section 2 briefly discusses SCOOP historical notes and different techniques for debugging concurrent programs. Section 3 discusses the main notions that we used such as logical schedule identification during a SCOOP program execution, describes our approach for recording and replaying logical schedule information. Section 4 discusses visualizing the runtime structure of SCOOP programs. Section 5 shows on a couple of examples that used record/replay tool with SCOOP programs. Section 6 describes a software implementation and developer’s guide. Section 7 is the user guide. Finally, Section 8 concludes the paper.

2 Related work

A lot of previous approaches for replaying multithreaded applications were based on capturing of all shared memory accesses. We can achieve the same runtime behavior of multithreaded program by reproducing the sequence of shared memory accesses. But these approaches were large overheaded (in time and in space) in program traces generating [5, 6].

Leblanc and Mellor-Crummy [5] tried to reduce the log size, so they assumed that programs use consistent, coarse-grained operations, called CREW, for concurrent-read-exclusive-write. Only CREW flavors were used to shared objects access and program's traces were generated only from these access operations.

Carver and Tai [6] offered approach that is very similar to the previous. They also generate traces only for coarse-grained synchronization events, assuming that shared variables are well guarded within well-defined critical-sections.

Both approaches are very similar to ours because we also have coarse-grained synchronization operations in SCOOP, but we use more compact scheme for storing program traces. This scheme was developed by Choi and Srinivasan [1]. The main idea is to capture only sequences of synchronization events and to represent program's trace like a logical schedule. It does not require making modifications to the operating system, and is, therefore, independent of the operating system. Only the SCOOP library and SCOOP compiler are affected. Particularly, we used SCOOP's own scheduler to collect separate calls – SCOOP's synchronization events.

3 A record/replay technique for SCOOP programs

In this chapter we discuss some theoretical and practical notions for recording and replaying SCOOP runtime behavior.

To replay a multithreaded program we have to record a thread's schedule information first and then replay exactly the same schedule for the next run. The threads schedule is a sequence of time intervals. Each interval contains execution events of a single thread. Interval boundaries correspond to moments of threads context switching. Let's refer to the thread schedule information obtained from operating system scheduler as the *physical schedule*. Likely, we can figure out the *logical schedule* information without any help from the system thread scheduler.

3.1 Logical schedules

To better understand the notion of logical schedules, consider a simple SCOOP program shown below:

```
class
	MAIN

create
	make

feature
make
	local
		j : INTEGER
	do
		create shared_obj
		create thread1.make ( shared_obj )
		run ( thread1 )
		j := 20
		io.put_string ( "shared = " + shared_obj.value.out + ", j = " + j.out )
	end
run ( a_thread : attached separate MYTHREAD )
	do
		a_thread.run
	end

feature {NONE}
	thread1 : separate MYTHREAD
	shared_obj : OBJECT
end
```

```
class
	MYTHREAD
create
	make

feature
	make ( an_object : OBJECT )
		do
			obj := an_object
		end

	run
		local
			k : INTEGER
		do
			k := 5
			obj.add ( k )
		end

feature {NONE} -- Implementation
	obj : OBJECT
end
```

```
class
	OBJECT

feature
	add ( a_value : INTEGER )
		do
		   value := value + a_value
		end

	value : INTEGER

end
```

Here, thread *MAIN* starts a child thread, *thread1*. Both *MAIN* and *thread1* can access the shared object, *shared_obj – MAIN* reads *shared_obj* and *thread1* reads and writes *shared_obj*. Variables *k* and *j* are thread-local variables, while *shared_obj* is thread-shared variable.

Figure 1 depicts a few execution instances (physical schedules) of the example program on a uniprocessor machine. Time is marked in the vertical direction:

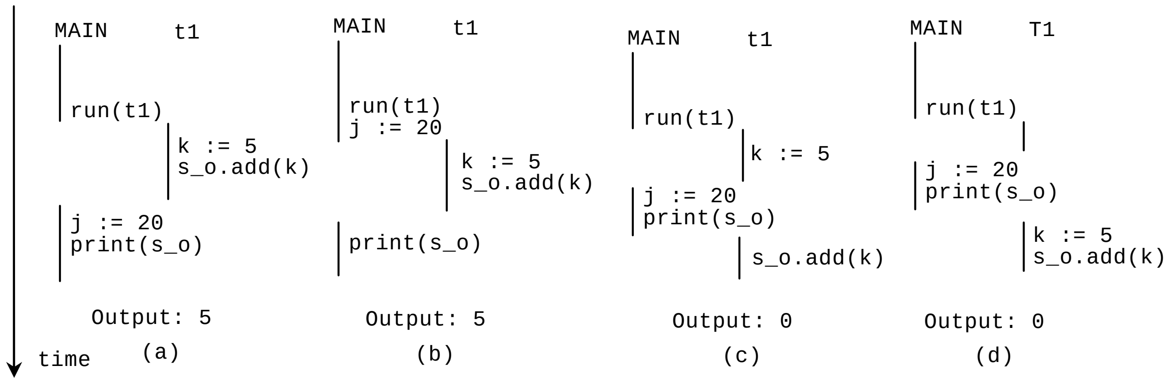


Figure 1. Different execution examples.

You can see the results of programs execution are different and depend on the order of shared variable accesses. Let’s classify physical schedules with the same order of shared variable accesses to one equivalence class. At example shown below, one equivalence class contains schedules (a) and (b). Schedules (c) and (d) belong to another one equivalence class. We denote all the *physical schedules* in the same equivalence class as a *logical schedule*.

Synchronization events can potentially affect the order of shared variable accesses, and thus affect the possible logical schedules. A simple SCOOP model doesn’t provide any explicit statements for synchronization, but automatically does it when any separate call occurs [2]. So we can count the separate call like a synchronization event. Actually, a compulsory condition for any record/replay tool is to capture order of all synchronization events and shared variable accesses to reproduce exactly the same execution behavior of the program.

Let say some additional words about synchronization in SCOOP, particularly potential data races. An absence of low-level data races is guaranteed by SCOOP model. Actually only one processor can execute features on a particular object. It means unique access to particular object. What about high-level data races? One can easily create them artificially but they are also simple to recognize, neutralize and avoid in future. So let make one general assumption that we have no low- and high-level data races in SCOOP. It simplifies the recording-replaying procedure indeed. Otherwise we have to use low-level programming approach to reproduce data-races that accompanied with a lot additional troubles.

In SCOOP every separate object is associated with processor and every processor is associated with a physical thread. An execution order of asynchronous separate calls is managed by SCOOP's scheduler. We changed the SCOOP scheduler main-loop to collect the order of separate calls for evaluating a logical schedule.

3.2 Recording of runtime behavior

The approach to capture logical schedule information is based on a global clock (just an integer counter) for the SCOOP scheduler and one local clock for each SCOOP processor.

The global clock and local clock of the SCOOP processor start with the same time value, the local clock stays behind the global clock when a different processor executes a separate call: when the processor is scheduled out, the global clock continues to count for each separate call executed by other processors while the local clock stays still. We use this observation in capturing the logical intervals for each processor. Figure 2 presents a block-scheme of algorithm for capturing logical schedule for SCOOP program:

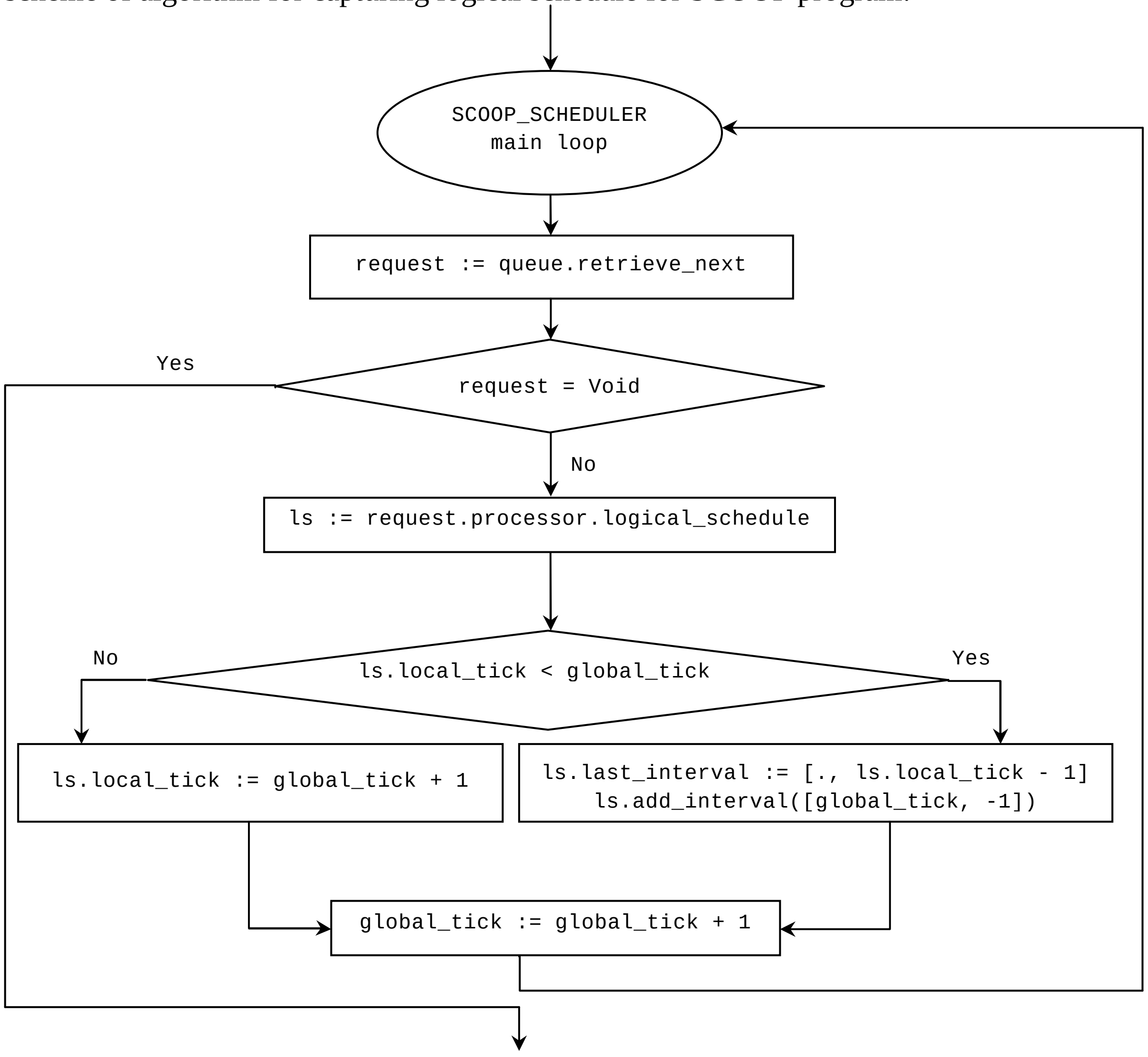
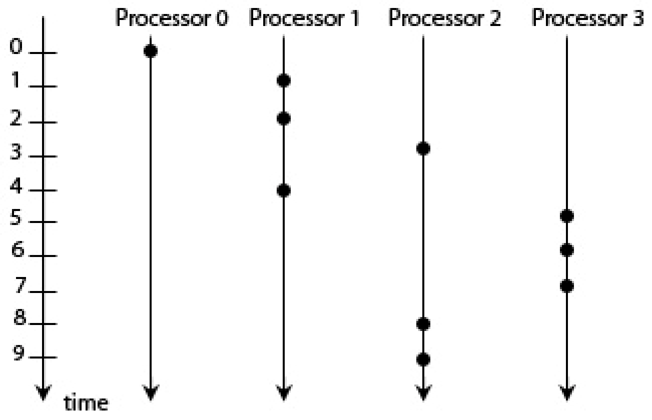

SCOOP_SCHEDULER
main loop
request := queue.retrieve_next
request = Void
Yes
No
ls := request.processor.logical_schedule
ls.local_tick < global_tick
No
Yes
ls.local_tick := global_tick + 1
ls.last_interval := [., ls.local_tick - 1]
ls.add_interval([global_tick, -1])
global_tick := global_tick + 1

Figure 2. Logical schedule capturing algorithm.

1. At the beginning, all the local tick counters and the global tick have the same value (say, zero).
2. When a separate call is tried to be executed by a processor, the SCOOP scheduler catches it first and compares the processor's local clock with the global tick.
3. If the two tick values are different, the scheduler has just detected the end of the previous logical interval and also the start of a new logical interval.
4. The scheduler increments the global tick counter, synchronizes the processor's local tick counter with the global tick and says "ready" for the current separate call.

The output of algorithm is a sequence of logical intervals. Each logical interval is a set of maximally consecutive separate calls of a SCOOP processor. Figure 3 presents an example of the working of the algorithm:



Processor 0: [0, 0]
Processor 1: [1, 2], [4, 4]
Processor 2: [3, 3], [8, 9]
Processor 3: [5, 7]

Figure 3. Identifying Logical Schedules.

Here the time axis is associated with SCOOP scheduler. Each black point denotes a separate call that assigns with the certain SCOOP processor. We have only to know in which moment the SCOOP scheduler maintains the current separate call. But it does not matter when the separate call will physically arise and when it will be finished.

3.3 Matching of SCOOP processors

The logical schedule does not contain enough information to reproduce the execution behavior of the program during a replay. The main reason is that SCOOP processors identifiers are not fixed. Each SCOOP processor associates with operating system thread and has the same identifier. So from run to run of program we will get different identifiers of SCOOP processors that will make it impossible to reproduce the program behavior.

From the first look the problem is connected to another – threads can be created in arbitrary order; we cannot fix the order and cannot give unique number for each thread. Luckily, the SCOOP model provides a mechanism to control the SCOOP processors creation – creation procedure is managed by SCOOP scheduler. So we can fix this problem having a parent for every child processor. Let’s take a look at an example:

```
class
     APPLICATION

create
     make

feature
     make
            do
                   create obj1.make
                   --creating new separate object inside a make feature
                   create obj2.make
                   --creating new separate object inside a make feature
                   create obj3.make
                   --creating new separate object inside a make feature

            end

feature {NONE}

     obj1, obj2, obj3 : separate OBJECT

end
```

SCOOP processors sequentially creating is guaranteed by the SCOOP model. Next, creation features can be executed concurrently in a different order. But it really does not matter because we already have frozen stable numbers for child processors and fixed a parent for that.

An example of different order SCOOP processors creation is depicted in Figure 4. We can see that processors tree structure is identical. Numbers over the graph’s edges denote a creation time moment for child processor. Numbers in rounded brackets denote a local id of processor. Local id is a processor number that used to distinguish processors at the same level of the processors’ tree:

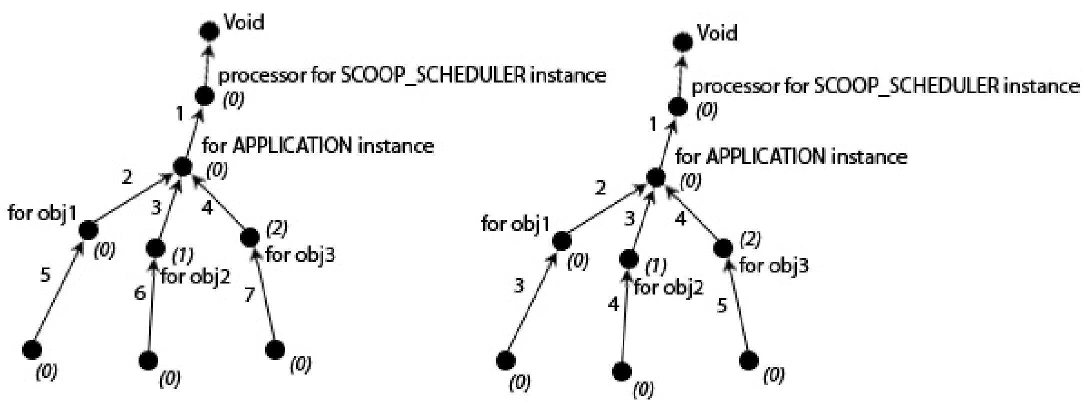


Figure 4. SCOOP processors creation mechanism. Regardless of the creation order we obtain the same processors identifiers.

Now we can assign unique identifier to each SCOOP processor that does not change from runtime to runtime. For example, let’s take a look on the left part of Figure 4 and obtain the identifier for processor that holds *obj2*. We simply go from this node to root of the tree and pick up all processors numbers. For processor that holds *obj2* we will get the identifier {1, 0, 0} and so on.

3.4 Replaying of runtime behavior

After processors identifying and logical schedule capturing we have enough information for replaying SCOOP programs. Now we will describe a replay algorithm built-in into the SCOOP scheduler:

1. At the beginning $global\_tick$ counter has to be set to a zero value.
2. Increment a global tick counter and get the next request from SCOOP scheduler tasks rounded queue. Each request contains a processor that has to be used for execution the request. Denote it like a current processor.
3. Retrieve previously recorded logical schedule intervals, LSI, for the current processor, $LSI = \{[a_1, b_1], [a_2, b_2], \dots, [a_n, b_n]\}$. Here we use unique processors identifiers to match the current processor with some processor in the logical schedule.
4. If $\exists i \in \{1, .., n\} \, / \, [a_i, b_i] \in LSI \wedge global\_tick \in [a_i, b_i]$ then execute the current request else go to the second step and repeat.

Now we can simply record and replay SCOOP runtime to recreate the same behavior for debugging purposes. If we capture the runtime example with some bug we would like to reproduce it and to observe what occurred at the SCOOP model level.

4 Visualizing the runtime structure of SCOOP programs

Visualizing the SCOOP runtime structure can be useful for debugging concurrent programs. The tool can be used for observing a sequence of separate calls and some additional information that we need, like a processors structure, names of classes, separate features that have executed and locked processors that have locked for separate calls execution. All this information is collected during two stages: processors creating and SCOOP scheduler main loop executing. All collected information is serialized to dot-format just before finishing the SCOOP program. Dot is a special text format for visualizing any directed graphs and diagrams.

We presented an example of feature work shown below in a Figure 5. We applied the feature for “quick sort” algorithm written in SCOOP. Here we use 5 elements for sorting. The classes relevant to implementation of this algorithm are located in the cluster ***scoop/quick_sort*** in the ***examples*** folder.

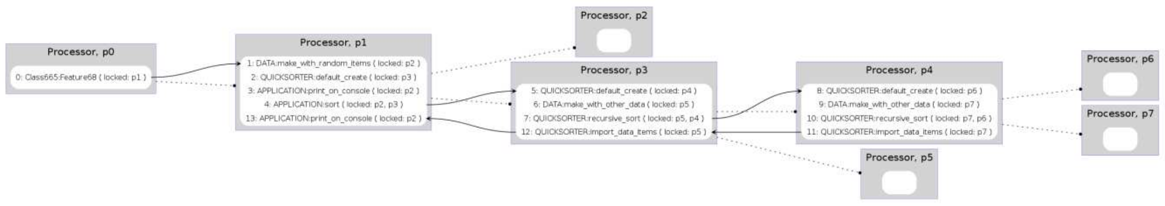

Processor, p0
0: Class665:Feature68 (locked: p1)
Processor, p1
1: DATA:make_with_random_items (locked: p2)
2: QUICKSORTER:default_create (locked: p3)
3: APPLICATION:print_on_console (locked: p2)
4: APPLICATION:sort (locked: p2, p3)
13: APPLICATION:print_on_console (locked: p2)
Processor, p2
Processor, p3
5: QUICKSORTER:default_create (locked: p4)
6: DATA:make_with_other_data (locked: p5)
7: QUICKSORTER:recursive_sort (locked: p5, p4)
12: QUICKSORTER:import_data_items (locked: p5)
Processor, p4
8: QUICKSORTER:default_create (locked: p6)
9: DATA:make_with_other_data (locked: p7)
10: QUICKSORTER:recursive_sort (locked: p7, p6)
11: QUICKSORTER:import_data_items (locked: p7)
Processor, p5
Processor, p6
Processor, p7

Figure 5. The SCOOP diagram for “quick sort”.

Here, the dotted arrows express the parent-child relationship for processors, while the solid arrows show the sequence of separate calls in the particular run.

5 Examples

In this chapter we will describe some examples and techniques that could be used in a SCOOP development. First we will show an example that demonstrates record/replay technique. Next we will show how we can recognize a deadlock in a SCOOP program.

5.1 Stock exchange

Consider a model of stock exchange. Traders are independent subjects and can buy or sell some quantity of stocks arbitrarily. They make transactions by the use of brokers and each trader is randomly attached to one broker. Brokers accept the trader's orders and carry out it on the stock exchange. Every trader's transaction influences the stock price. Also we have two types of traders – bulls and bears. The number of each type of trader is randomly set at the beginning.

We would like to implement this model in SCOOP. So as all transaction events can arise asynchronously we declare all traders as "separate", hence brokers and stock exchange we have to declare "separate" too.

The following figure presents the principal model of stock exchange written in SCOOP:

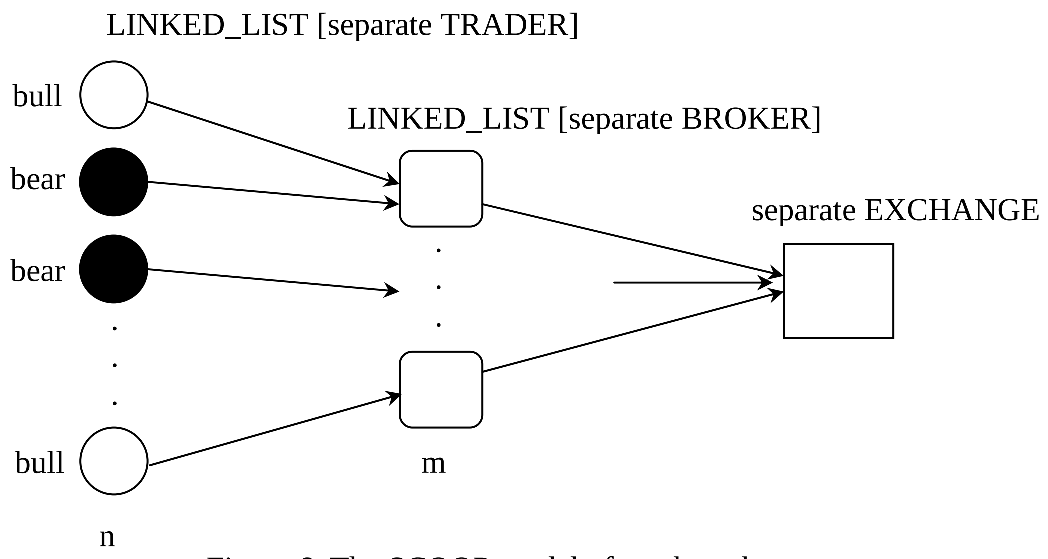


Figure 6. The SCOOP model of stock exchange.

The EXCHANGE class contains the initial stock price and `is_bullish_trend` feature which is true if we have a bullish trend (current price higher than previous) now and false when bearish trend (otherwise). Now we describe how every trader's transaction influence to stock price. Price evaluating law is $stock\_price := stock\_price + impact\_factor * quantity$. Quantity of stock is the same for every transaction. Let's describe a simple algorithm for $impact\_factor$ evaluating that implemented in EXCHANGE class:

If trader is bull (and hope for price growing always) there are two branches:

- If current trend is bullish then $impact\_factor = 2$
- If current trend is bearish then $impact\_factor = 1$

If trader is bear (and hope for price falling always) there are also two branches:

- If current trend is bullish then $impact\_factor = -1$
- If current trend is bearish then $impact\_factor = -2$

You can simply realize – the output stock price "randomly" depends on trader's transactions sequence (bulls-bears sequence) that depends on system scheduler work by-turn.

Using record/replay tool you can repeat the sequence of trader's transactions and achieve the same execution behavior. The classes relevant to implementation of this model are located in the cluster ***scoop/exchange*** in the ***examples*** folder.

5.2 Deadlock

One of the main problems of SCOOP development is deadlock. The SCOOP model developers elaborated a really simple approach for concurrent programming but still, there is no built-in mechanism for recognizing deadlock. We know that deadlocks have an asynchronous nature and it's very difficult to reproduce and debug them. Our record/replay tool allows do that. With the visualizing tool you can observe interactions between the SCOOP model abstractions to understand why the deadlock has occurred. Let's take a look at an example with deadlock.

Consider an example with two bank accounts and at least two clients. One client wants to make transaction from account 1 to account 2 and another client also wants to make transaction, but first withdraw funds from account 2 and then deposit to account 1. Of course client's transactions can occur asynchronously and at the same moment. Hence we get a deadlock:

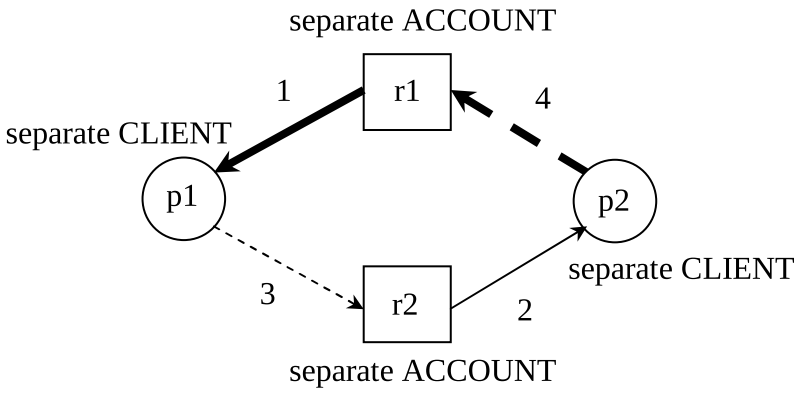


Figure 7. The deadlock for SCOOP model of bank account.

First at time moment 1 the object r1 is allocated and locked by SCOOP processor that holds p1. Next, at moment 2 SCOOP processor that holds r2 locked by processor that holds p2. Next, at moment 3, p1 try to catch object r2 and then blocked because r2 has already locked by p2. The same for p2 and r1. We have got an endless waiting for this case of processor-resource sequence of allocations. Using record/replay and visualizing tool we can reproduce this deadlock and visualize a scheme of SCOOP objects interactions to localize the deadlock. Let's take a look on the next Figure 8:

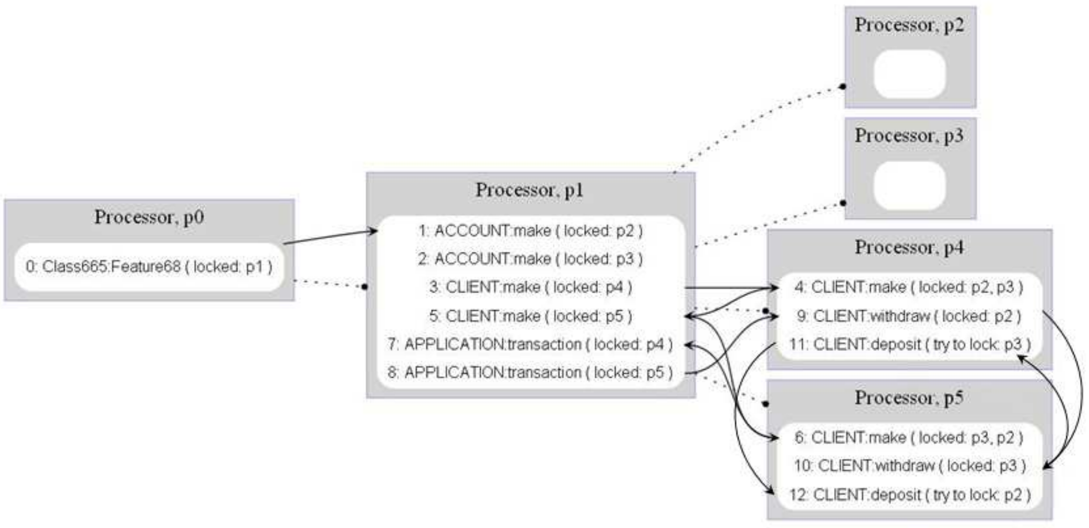

Processor, p0
0: Class665:Feature68 (locked: p1)
Processor, p1
1: ACCOUNT:make (locked: p2)
2: ACCOUNT:make (locked: p3)
3: CLIENT:make (locked: p4)
5: CLIENT:make (locked: p5)
7: APPLICATION:transaction (locked: p4)
8: APPLICATION:transaction (locked: p5)
Processor, p2
Processor, p3
Processor, p4
4: CLIENT:make (locked: p2, p3)
9: CLIENT:withdraw (locked: p2)
11: CLIENT:deposit (try to lock: p3)
Processor, p5
6: CLIENT:make (locked: p3, p2)
10: CLIENT:withdraw (locked: p3)
12: CLIENT:deposit (try to lock: p2)

Figure 8. SCOOP model diagram with deadlock. Visualized by GraphViz.

Separate calls 9, 10, 11 and 12 bring to deadlock. Call 11 tries to lock p3 but it's already locked on 10th call. Call 12 tries to lock p2 but it's already locked on 9th call. We have got an endless waiting.

The classes relevant to implementation of this model are located in the cluster ***scoop/bank_deadlock*** in the ***examples*** folder.

6 Developer's guide

In this chapter we provide information for developers who want to know technical details or want to extend the implementation of SCOOP record/replay.

The whole implementation of SCOOP record/replay can be divided into three groups: Eiffel Studio integration, SCOOP compiler integration and SCOOP Library integration.

All changes in touched classes enclosed in special comment lines and have the following structure:

```
-- SCOOP REPLAY

       Changes and additions

-- SCOOP REPLAY end
```

6.1 SCOOP library integration

The main part of record/replay techniques was implemented in the SCOOP library. Related classes can be found in the cluster ***replay*** in the ***scoopli*** library.

Names and constants

Configuration options with default values contained in class SCOOP_LIBRARY_CONSTANTS. Listing 1 shows description for each option:

Listing 1:

```
REPLAY_directory_name: STRING = "scoop_replay"
       -- Replay directory.
REPLAY_file_extension: STRING = "sls"
       -- Extension for replay file.
REPLAY_diagram_file_extension: STRING = "dot"
       -- Extension for diagram file.
REPLAY_file_header: STRING = "scoop_replay_file"
       -- Header that every replay file has to contain at the beginning.
REPLAY_command_line_argument_beginning: STRING = "-REC_REP"
       -- Left bound of replay command line arguments.
REPLAY_command_line_argument_end: STRING = "+REC_REP"
       -- Right bound of replay command line arguments.
REPLAY_command_line_argument_record: STRING = "RECORD"
       -- Replay command line argument that activates record mode.
REPLAY_command_line_argument_replay: STRING = "REPLAY"
       -- Replay command line argument that activates replay mode.
REPLAY_command_line_argument_diagram: STRING = "DIAGRAM"
       -- Replay command line argument that activates diagram generation.
REPLAY_command_line_argument_verbose_info: STRING = "VERBOSE_INFO"
```

```
        -- Replay command line argument that activates verbose replay
information output.
```

Record/replay tool

The entry point for making changes to the SCOOP record/replay is class SCOOP_REPLAY_LOGICAL_SCHEDULE_COLLECTOR that collects logical schedules of application's executions and reproduce their while replay. The classes relevant to implementation of record/replay techniques located in the cluster ***replay*** in the ***scoopli*** library. Table 1 describes the functionality of each class.

Class name	**Functionality**
SCOOP_REPLAY_LOGICAL_SCHEDULE_COLLECTOR	Class that collects logical schedules for scoop programs and executes replay.
SCOOP_REPLAY_LOGICAL_SCHEDULE	Class that represents logical schedule for current scoop processor.
SCOOP_REPLAY_LOGICAL_INTERVAL	Class that represents interval of separate calls, that current processor executes without interleaving.
SCOOP_REPLAY_FILE	Class that saves record/replay data on hard drive.

Table 1. Record/replay classes.

To implement the record/replay algorithm in the SCOOP library, the following general classes had been modified: SCOOP_STARTER_IMP, SCOOP_PROCESSOR, SCOOP_SCHEDULER. Table 2 shows main changes:

Class name	**Modifications**
SCOOP_STARTER_IMP	Added parsing of record/replay command line arguments.
SCOOP_PROCESSOR	Added ***local_id***, ***parent_processor*** and ***children_number*** features for collecting processors relation's tree. Added references on processor's logical schedules for record and replay.
SCOOP_SCHEDULER	Added flags for identifying record and replay execution modes. Record and replay logic had been integrated into scheduler's main loop.

	Record/replay information saving after finish of execution or after unhandled exception.

Table 2. Main changes in SCOOP library classes.

Visualizing tool

The classes related to the visualizing the runtime structure of SCOOP programs with diagram can be found in cluster ***replay/diagram*** in the ***scoopli*** library. Table 3 shows their main functionality:

Class name	**Functionality**
SCOOP_REPLAY_DIAGRAM	Class that collects information about processor's structure, separate calls and locked processors and save this information into diagram file.
SCOOP_REPLAY_DIAGRAM_NODE	Class that represents the node of SCOOP processor's structure.
SCOOP_REPLAY_DIAGRAM_FEATURE	Class that represents the separate call's feature with a list of locked processors.

Table 3. Visualizing the runtime structure classes.

6.2 SCOOP compiler integration

This part of the integration applies to SCOOP code generation. To be able to hold SCOOP processor's structure and to track down processor's relations for every execution we added *parent_processor* feature into SCOOP_PROCESSOR class. The only change in SCOOP compiler is creating new SCOOP processors via *new_processor_* creation procedure with *parent_processor* as a parameter. Thus in class SCOOP_CLIENT_CONTEXT_AST_PRINTER that generates SCOOP code we replaced the *new_processor_* feature with *new_processor_ (a_parent_processor),* where *a_parent_processor* is current processor.

Class SCOOP_CLIENT_CONTEXT_AST_PRINTER is located in ***scoop2scoopli*** library in cluster ***ast_visitor***.

6.3 Eiffel Studio integration

This group of changes and new classes related to EVE IDE and provides user interface of record-replay facilities.

Names and constants

The names, titles and texts for record/replay Graphical User Interface are contained in class INTERFACE_NAMES.

Menu items

The classes relevant to main menu can be found in cluster ***interface/new_graphical/scoop_replay_tool*** defined in target bench of the EVE project. Table 4 shows functionality that each class provides:

Class name	**Functionality**
EB_SCOOP_EXECUTION_RECORDING_MODE_CMD	Command that activates/deactivates SCOOP execution recording
EB_SCOOP_EXECUTION_REPLAY_CMD	Command that launches SCOOP execution replay wizard
EB_SCOOP_EXECUTION_DIAGRAM_CMD	Command that activates/deactivates SCOOP execution diagram generation

Table 4. Menu items classes.

To integrate these commands into IDE several classes had been touched. Table 5 shows main changes in their functionality:

Class name	**Modifications**
DEBUGGER_MANAGER	Added two boolean flags that indicate record mode and diagram generation mode: *scoop_execution_recording_enabled* *scoop_execution_diagram_enabled*
EB_DEBUGGER_MANAGER	Group of record/replay items in the Execution menu became to depend on project type (SCOOP project or regular Eiffel project). Three items added into Execution menu for SCOOP projects: “Activate SCOOP Execution Recording”, “Replay SCOOP Execution…” and “Activate SCOOP Execution Diagram Generation” according to commands described above.
EB_EXEC_DEBUG_CMD	Group of record/replay items in the Run toolbar drop down button became to depend on project type. Three items added into Run toolbar drop down button for SCOOP projects: “Activate SCOOP Execution Recording”, “Replay SCOOP Execution…” and “Activate SCOOP Execution Diagram Generation” according to commands

	described above.
EB_DEBUG_RUN_CMD	Added passing of record/replay execution parameters before launching application by clicking Run button. If SCOOP recording or SCOOP Execution Diagram Generation activated, parameters pass via command line arguments.

Table 5. Main changes in Eiffel Studio classes.

Replay Wizard

The classes related to SCOOP Execution replay wizard can be found in cluster ***interface/new_graphical/scoop_record_replay_tool/wizard*** defined in target bench of the EVE project.

The entry point for changes to the wizard is class EB_SCOOP_REPLAY_WIZARD_MANAGER, where wizard information creates and initial state shows. Table 6 contains information about wizard classes.

Class name	**Functionality**
EB_SCOOP_REPLAY_WIZARD_MANAGER	Class which is launching SCOOP record/replay wizard.
EB_SCOOP_REPLAY_WIZARD_INFORMATION	Class that contains information associated with a state in the wizard.
EB_SCOOP_REPLAY_WIZARD_SHARED_INFORMATION	Shared class to access the information for the wizard.
EB_SCOOP_REPLAY_WIZARD_INITIAL_STATE	Initial window with the description of SCOOP replay facilities.
EB_SCOOP_REPLAY_WIZARD_SELECT_STATE	State for selecting a file for replay.
EB_SCOOP_REPLAY_WIZARD_FINAL_STATE	State for launching replay with selected logical schedule.
EB_SCOOP_REPLAY_WIZARD_DIRECTORY_ERROR_STATE	Error state if directory doesn't exist or contains no replay files.
EB_SCOOP_REPLAY_WIZARD_FILE_ERROR_STATE	Error state if selected file has wrong format or corrupted.

Table 6. SCOOP replay wizard classes.

7 User guide

This chapter explains how to use SCOOP record/replay tool and visualizing SCOOP runtime structure tool.

7.1 Using command line arguments

All record/replay features applied by passing parameters to the application as command line arguments. Hence for the work with the tool recompiling of the SCOOP program is not needed. Here is an example of command line arguments, which activates Recording mode and Diagram Generation mode and saves logical schedule into ***my_replay.sls*** file and saves execution diagram into ***my_diagram.dot*** file:

```
-rec_rep record my_replay.sls diagram my_diagram.dot +rec_rep
```

Table 7 contains description of each command line parameter:

Parameter	**Description**
–rec_rep	Left bound of record/replay command line arguments.
+rec_rep	Right bound of record/replay command line arguments.
record	Argument that activates Recording mode. File name for recorded logical schedule should be the next parameter. If file name is not specified date and time are using.
replay	Argument that activates Replay mode. File name for replay logical schedule has to be the next parameter. If file name is not specified error occurs.
diagram	Argument that activates Diagram Generation mode. File name for recorded execution diagram should be the next parameter. If file name is not specified date and time are using.
verbose_info	Argument that activates verbose information output.

Table 7. Record/replay command line arguments.

All record/replay arguments have to be enclosed between ***–rec_rep*** and ***+rec_rep***. Presence of considered arguments activates respective execution mode. Arguments are non-sensitive to the register and can be written in different order.

7.2 Using the Graphical User Interface

Considered tools are integrated into EVE IDE, thus application can be launched with suitable arguments from Eiffel Studio.

After compiling SCOOP program you can activate or deactivate execution modes by accessing the Execution menu or Run drop-down button, as shown in Figure 9. Click Run to launch application with selected parameters.

Logical schedule file that has ***.sls*** extension and execution diagram file that has ***.dot*** extension will be saved by default into ***scoop_replay*** directory with names according to starting execution time. Directory can be found in the target folder, inside the ***EIFGENs*** folder.

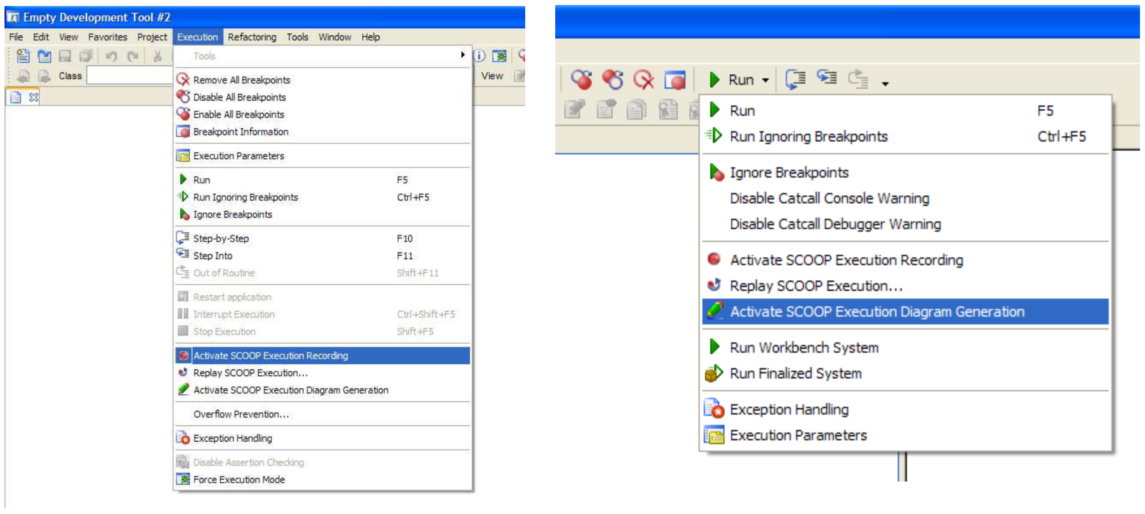
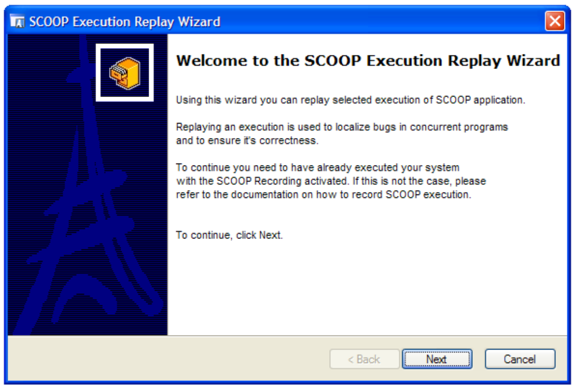

Empty Development Tool #2
File Edit View Favorites Project Execution Refactoring Tools Window Help
Class
View
Tools
Remove All Breakpoints
Disable All Breakpoints
Enable All Breakpoints
Breakpoint Information
Execution Parameters
Run F5
Run Ignoring Breakpoints Ctrl+F5
Ignore Breakpoints
Step-by-Step F10
Step Into F11
Out of Routine Shift+F11
Restart application
Interrupt Execution Ctrl+Shift+F5
Stop Execution Shift+F5
Activate SCOOP Execution Recording
Replay SCOOP Execution...
Activate SCOOP Execution Diagram Generation
Overflow Prevention...
Exception Handling
Disable Assertion Checking
Force Execution Mode
Run
Run F5
Run Ignoring Breakpoints Ctrl+F5
Ignore Breakpoints
Disable Catcall Console Warning
Disable Catcall Debugger Warning
Activate SCOOP Execution Recording
Replay SCOOP Execution...
Activate SCOOP Execution Diagram Generation
Run Workbench System
Run Finalized System
Exception Handling
Execution Parameters

Figure 9. Record/replay integration into IDE.

Running replay wizard

Start Replay wizard by selecting ***Replay SCOOP Execution…*** in the ***Execution*** menu or ***Run*** drop-down button, as shown in Figure 10. Choose ***Next*** at the welcome screen:


SCOOP Execution Replay Wizard

Welcome to the SCOOP Execution Replay Wizard

Using this wizard you can replay selected execution of SCOOP application.

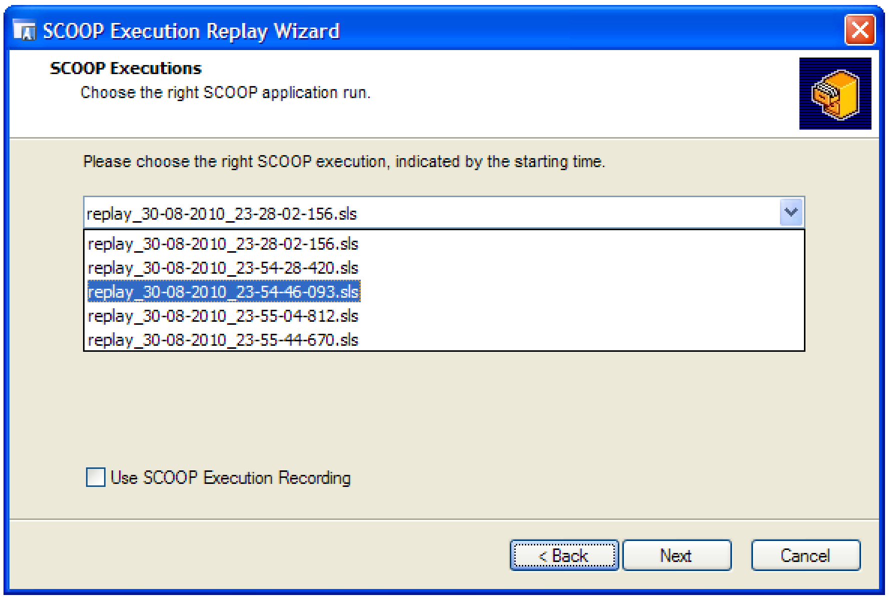
Replaying an execution is used to localize bugs in concurrent programs and to ensure it's correctness.

To continue you need to have already executed your system with the SCOOP Recording activated. If this is not the case, please refer to the documentation on how to record SCOOP execution.

To continue, click Next.

Figure 10. Starting the Replay wizard.

If you have already recorded executions of your SCOOP application, the next step will ask you to select logical schedule for replay. Choose the right replay file that represented by starting time and select ***Use SCOOP Execution Recording*** check box if you also want to record replay execution into new file. Then click ***Next***, as shown in Figure 11:



Choose the right SCOOP application run.

Please choose the right SCOOP execution, indicated by the starting time.

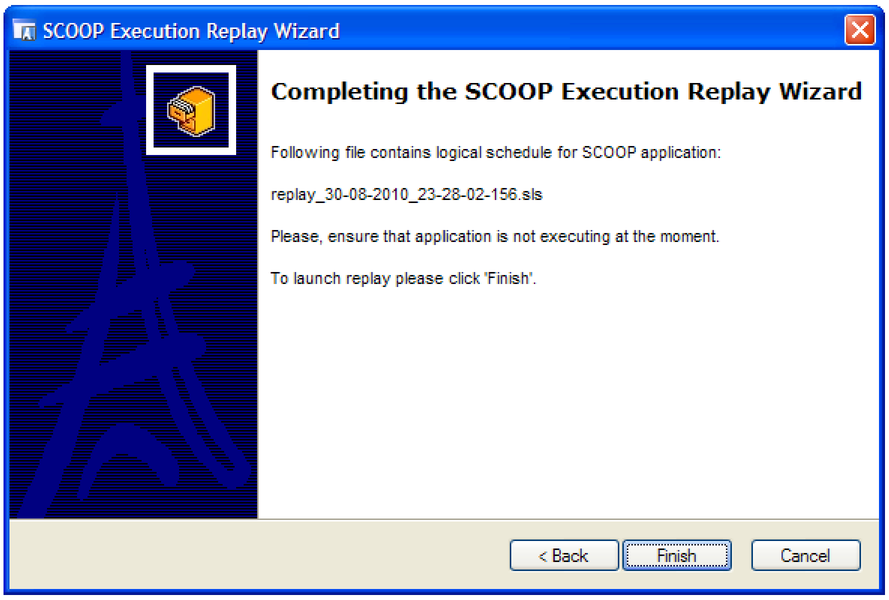
replay_30-08-2010_23-28-02-156.sls

replay_30-08-2010_23-28-02-156.sls
replay_30-08-2010_23-54-28-420.sls
replay_30-08-2010_23-54-46-093.sls
replay_30-08-2010_23-55-04-812.sls
replay_30-08-2010_23-55-44-670.sls

☐ Use SCOOP Execution Recording

Figure 11. Replay file selection.

If selected replay file exists and can be executed you will see the final step of the Replay wizard. Click ***Finish*** to launch replay execution, as shown in Figure 12:



Completing the SCOOP Execution Replay Wizard

Following file contains logical schedule for SCOOP application:

replay_30-08-2010_23-28-02-156.sls

Please, ensure that application is not executing at the moment.

To launch replay please click 'Finish'.

Figure 12. Launching Replay execution.

8 Conclusion

We end this technical report with some considerations about the project, the goals that we achieved and some personal comments.

To conclude the project we briefly discuss about main goals that we achieved during the work:

1. Developed and implemented record/replay technique for reproducing SCOOP programs behavior.
2. Implemented tool for visualizing SCOOP diagram for program runtime.
3. Integrated record/replay technique and visualizing tool into EVE. All the code is integrated into EVE and is part of the last releases that include SCOOP.
4. Clearly documented the design choices and concepts that we used.
5. Clearly documented the user and developer guides that allow to extend and properly use the record/replay features.
6. Properly documented and tested all the code. All the code has been tested with a few examples and from the ones included in EVE.

We hope that this project's outcomes can stay a good base stone for the future work on SCOOP project, particularly in testing and debugging directions for concurrent programs written in SCOOP.

References

[1] J.-D. Choi and H. Srinivasan. Deterministic Replay of Java Multithreaded Applications. IBM T. J. Watson Research Center, Hawthorne, NY, USA, 10532.

[2] Meyer B.: Object-Oriented Software Construction, chapter 31, 2nd edition, Prentice Hall, 1997.

[3] B. Morandi, S. Bauer, and B. Meyer. SCOOP – a contract-based concurrent object-oriented programming model. Technical report, ETH Zurich and Ludwig-Maximilians-Universitat Munchen, 2009.

[4] Nienaltowski P.: Practical framework for contract-based concurrent object-oriented programming, PhD dissertation 17061, Department of Computer Science, ETH Zurich, February 2007.

[5] Thomas J. Leblanc and John M. Mellor-Crummy. Debugging parallel programs with instant replay. IEEE Transactions on Computers, C-36(4):471-481, April, 1987.

[6] K. C. Tai, Richard H. Carver, and Evelyn E. Obaid. Debugging concurrent Ada programs by deterministic execution. IEEE transactions on Software Engineering, 17(1):45-63, January 1991.

[7] SCOOP website. http://scoop.origo.ethz.ch/, 2010.

[8] EIFFEL website. http://www.eiffel.com/, 2010.

[9] EVE website. http://eve.origo.ethz.ch/, 2010.