Implementing Synchronous Reactive Programming in RxJava
- Bachelor thesis -

Student: Alexandru Babeanu s3004872
First supervisor: Msc. Mauricio Cano
Second supervisor: Dr. Jorge A. Pérez

July 26, 2018

Abstract

Programming languages with reactive features have become common practice for the design and implementation of reactive and interactive systems. This paper explores the relationship between the main principles and design decisions behind tow commonly used paradigms that offer reactive features: the Synchronous Reactive Programming (SRP) paradigm and reactive programming. It describes the implementation of an SRP language using constructs for reactive programming offered by the RxJava library. The primary focus of this thesis is to demonstrate that reactive programming is at least as expressive as synchronous reactive programming.

Contents

List of Figures

List of Tables

1 Introduction

In recent years, programming languages with reactive features, such as event streams, data-flows, and propagation of change, have become common in the design of interactive and reactive systems, like web applications or graphical user interfaces, for example.

Reactive programming [5] has become a popular choice in the implementation of interactive systems, and even languages that are not intrinsically reactive are being extended with constructs [13] that adhere to the reactive programming paradigm. The key feature of this paradigm is the immediate reaction to events; program components will react independently from one another, as soon as events become available. While this paradigm offers benefits like higher computational speed or program modularity, it intrinsically forbids deterministic semantics for concurrent behaviour [8]. This makes formal reasoning, such as the program analysis and verification, more difficult.

The Synchronous Reactive Paradigm (SRP) [1] also offers reactive features, and has been used for designing reactive systems with real-time constraints. The key feature that differentiates SRP from reactive programming is the coordination between the reactive components within a program. The threads of a program will not react to events independently from each other, they will instead use signals to synchronize their executions and ensure a deterministic order of operations. In programs that adhere to this paradigm, change is propagated only when all the threads have reached a so-called *suspended state*, from which execution cannot continue. This property allows for the design of synchronous reactive languages with deterministic semantics, which in turn facilitates formal analysis and verification of programs.

This paper describes the implementation of an interpreter for a synchronous reactive language (SRL), built through the use of reactive constructs. The source code for this interpreter is available at `https://github.com/babeanu-dorian/InterpreterSRL`. The model in [3] serves as the starting point for the implemented language. It is a relaxation of the Esterel model, [9], a robust synchronous programming language designed for the development of complex reactive systems.

The main objective of this thesis is to show that reactive programming is sufficiently expressive to implement synchronous reactive programming. By using the construction of this interpreter as a basis, this project seeks to study the underlying design decisions behind reactive programming and synchronous reactive programming, as well as the relationship between the two.

The "Preliminaries" section of the paper covers the theory behind the Synchronous Reactive Paradigm and reactive programming, introduces a model for an SRP language used as the basis for the implemented language, and describes the RxJava constructs that will be used in the implementation. Section 3 covers the design and implementation of the interpreter. It lists and explains the design decisions made regarding the implemented language and the implementation of the interpreter. It also offers a detailed explanation of the source code. The "Evaluation" section contains an analysis of the implemented language with

regards to its relation to the original model, its limitations and properties like determinism and concurrency. Section 5 offers a comparison between SRL and already established synchronous reactive languages.

2 Preliminaries

2.1 Synchronous Reactive Programming

The principle of Synchronous Reactive Programming [1] is to make the same abstraction for programming languages as the synchronous abstraction in digital circuits, where the timing characteristics of the electronic transistors are neglected (each gate is assumed to compute its result instantaneously, each wire is assumed to transmit its signal instantaneously). A synchronous circuit is clocked and at each tick of its clock, it computes instantaneously its output values and the new values of its memory cells from its input values and the current values of its memory cells.

This abstraction, known as the Synchronous Hypothesis [2], makes reasoning about program behaviour a lot easier. The main properties offered by the Synchronous Hypothesis are the fragmentation of program execution into logical ticks, and the use of signals for propagating information. A synchronous program reacts to its environment in a sequence of ticks, and computations within a tick are assumed to be instantaneous. Logical ticks, also named instants, are implemented as follows:

1. All threads execute until they all reach a suspended state.
2. The signals are evaluated once all threads are blocked.
3. Then execution proceeds to the next instant.

The term "instantaneous" does not refer to real time here, but to the actual execution of program instructions. In practice, this is modelled by the following two constraints:

1. A program will not accept new input until it finished reacting to the current input.
2. The real-time order of parallel instructions does not change the output of the program.

These properties make it possible to design synchronous reactive languages with deterministic semantics, which in turn simplifies the process of reasoning about program behaviour.

Another important characteristic of Synchronous Reactive Programming is the use of signals to achieve this synchronization of parallel instructions. They are similar to the current in a circuit wire: either present or absent. In practice, a synchronous reactive language has a set of threads running in parallel, or executed in an arbitrary order, since the actual order has to be irrelevant to the

output of the program. These threads use signals to coordinate their actions: a thread can wait for a signal to be made present or absent by a different thread before continuing execution.

2.2 A Basic Model for a Synchronous Reactive Language

The model in [3], is used as the base for the language described in this paper. It is largely inspired by the SL synchronous language introduced by Boussinot and De Simone in 1996, [4], which is itself a relaxation of the Esterel model, [9]. In this model, the reaction to the absence of a signal is postponed until the following instant in order to avoid the problem of causality cycles present in the Esterel model.

Causality cycles can occur if the model allows an instruction that tests the value of a signal to also alter the value of that signal. For example, in the program "`when A is absent, emit A`", if the signal `A` is considered present then it is absent, because it was not emitted, and if it is considered absent then it is emitted and made present. This program is erroneous, because there is has no solution (no signal environment that is consistent with its instructions).

To formally describe the operational semantics a few assumptions and notations are required. A countable set of signals s, s',... is assumed to exist. The notation Int represents a finite set of signal names representing an observable interface. The notation $[s/x]T$ refers to the thread resulted from replacing all instances of x in T with s. The predicate $(T,\ E) \Downarrow^P (T',\ E')$ is used to denote that the thread T in environment E executes an atomic, possibly empty, sequence of instructions, resulting in the thread T' in environment E', and the spawning of the multi-set of threads P.

Definition 1. A *thread* is defined as an expression written according to the following grammar:

$$T ::= () \mid (emit\ s) \mid (local\ s\ T) \mid (thread\ T) \\ \mid (when\ s\ T) \mid (watch\ s\ T) \mid A(s) \mid (T;\ T)$$

Definition 2. A signal environment E is a partial function from signal names to the boolean values $true$ and $false$, whose domain of definition $dom(E)$ contains Int.

The constructs that compose this model are as follows:

1. () is the terminated thread.
2. The construct $A(s)$ is used to execute threads that require parameters, defined as $T = A(x)$.
3. The $(emit\ s)$ construct is used to set the mapping of a signal in the environment to $true$.
4. The $(local\ s\ T)$ construct is used to declare a new signal, which is mapped to $false$ by default, bind it to the name s in T, then execute T.

5. The $(thread\ T)$ construct is used to execute a block of instructions T in parallel with the current thread.

6. The $(T_1;\ T_2)$ construct represents the sequential composition of instructions.

7. The construct $(when\ s\ T)$ will suspend the current thread until the signal s is mapped to $true$ and execute T afterwards.

8. The construct $(watch\ s\ T)$ will start the execution of T, but abort it at the end of the first instant in which the signal s is mapped to $true$.

The operational semantics are formalized as follows:

$$\frac{}{((),\ E) \Downarrow^{\emptyset} ((),\ E)} \tag{1}$$

$$\frac{A(x) = T,\ ([s/x]T,\ E) \Downarrow^{P} (T',\ E')}{(A(s),\ E) \Downarrow^{P} (T',\ E')} \tag{2}$$

$$\frac{}{((emit\ s),\ E) \Downarrow^{\emptyset} ((),\ E[s := true])} \tag{3}$$

$$\frac{s' \notin dom(E),\ ([s'/s]T,\ E \cup \{s' \to false\}) \Downarrow^{P} (T',\ E')}{((local\ s\ T),\ E) \Downarrow^{P} (T',\ E')} \tag{4}$$

$$\frac{}{((thread\ T),\ E) \Downarrow^{\{|T|\}} ((),\ E)} \tag{5}$$

$$\frac{(T_1,\ E) \Downarrow^{P_1} ((),\ E_1),\ (T_2,\ E_1) \Downarrow^{P_2} (T',\ E')}{((T_1;\ T_2),\ E) \Downarrow^{P_1 \cup P_2} (T',\ E')} \tag{6}$$

$$\frac{(T_1,\ E) \Downarrow^{P} (T',\ E'),\ T' \neq ()}{((T_1;\ T_2),\ E) \Downarrow^{P} ((T';\ T_2),\ E')} \tag{7}$$

$$\frac{E(s) = false}{((when\ s\ T),\ E) \Downarrow^{\emptyset} ((when\ s\ T),\ E)} \tag{8}$$

$$\frac{E(s) = true,\ (T,\ E) \Downarrow^{P} ((),\ E')}{((when\ s\ T),\ E) \Downarrow^{P} ((),\ E')} \tag{9}$$

$$\frac{E(s) = true,\ (T,\ E) \Downarrow^{P} (T',\ E'),\ T' \neq ()}{((when\ s\ T),\ E) \Downarrow^{P} ((when\ s\ T'),\ E')} \tag{10}$$

$$\frac{(T,\ E) \Downarrow^{P} ((),\ E')}{((watch\ s\ T),\ E) \Downarrow^{P} ((),\ E')} \tag{11}$$

$$\frac{(T,\ E) \Downarrow^{P} (T',\ E'),\ T' \neq ()}{((watch\ s\ T),\ E) \Downarrow^{P} ((watch\ s\ T'),\ E')} \tag{12}$$

A program P is a finite non-empty multi-set of threads. The notations $sig(T)$ and $sig(P)$ are used to denote the set of signals that are free in a thread T, and respectively, the set of signals that are free in a program P. Through the course of an instant, all threads are scheduled non-deterministically to progress according to the semantics described above. In order for an instant to end, all threads must reach a state from which progress can no longer be made. A thread T in an environment E is suspended (denoted as $(T,\ E)\ddagger$) when it is either terminated ($T = ()$) or its execution was suspended by a *when* statement ($T = (when\ s\ A),\ E(s) = false$). This is represented via the following rule:

$$\frac{(T,\ E) \Downarrow^{\emptyset} (T,\ E)}{(T,\ E)\ddagger} \tag{13}$$

During the transition from one instant to the next, all active $(watch\ s\ A)$ instructions are replaced by the terminated thread if $E(s) = true$, then all signals in the environment are mapped to $false$.

Definition 3. The *abort* operation associated with the *watch* construct is defined as the function $\lfloor\_\rfloor_E$:

$$\lfloor P \rfloor_E = \{\lfloor T \rfloor_E \mid T \in P\} \quad \lfloor () \rfloor_E = () \quad \lfloor T;\ T' \rfloor_E = \lfloor T \rfloor_E;\ T'$$

$$\lfloor when\ s\ T \rfloor_E = \begin{cases} (when\ s\ \lfloor T \rfloor_E) & \text{if } E(s) = true \\ (when\ s\ T) & \text{otherwise} \end{cases}$$

$$\lfloor watch\ s\ T \rfloor_E = \begin{cases} () & \text{if } E(s) = true \\ (watch\ s\ \lfloor T \rfloor_E) & \text{otherwise} \end{cases}$$

The execution of a program P in the environment E is formalized using the following two transition rules:

$$\frac{\forall T \in P\ (T,\ E)\ddagger}{(P,\ E) \Downarrow (\lfloor P \rfloor_E, E)} \tag{14}$$

$$\frac{\exists T \in P\ \neg((T,\ E)\ddagger),\ (T,\ E) \Downarrow^{P'} (T',\ E'),\ (P \backslash \{|T|\} \cup \{|T'|\} \cup P', E) \Downarrow (P'', E'')}{(P,\ E) \Downarrow (P'',\ E'')} \tag{15}$$

Finally, the input-output behaviour of a program is described by labelled transitions $P \xrightarrow{I/O} P'$, where $I, O \subseteq Int$ are the signals declared in the interface of the program. The transition means that a program in state P with input I moves to state P' with output O:

$$\frac{(P,\ E_{I,P}) \Downarrow (P',\ E'),\ O = \{s \in Int \mid E'(s) = true\}}{P \xrightarrow{I/O} P'} \tag{16}$$

$$\text{where: } E_{I,P} = \begin{cases} true & \text{if } s \in I \\ false & \text{if } s \in (Int \cup sig(P)) - I \ \} \\ undefined & \text{otherwise} \end{cases}$$

The following constructs, which are essential to synchronous reactive programming, are derived from the previously mentioned basic constructs:

$$\begin{array}{lcl}
(await\ s) & = & (when\ s\ ()) \\
(loop\ T) & = & A(s) \text{ where } \{s\} = sig(T),\ A(s) = (T;\ A(s)) \\
(now\ T) & = & (local\ s\ ((emit\ s); (watch\ s\ T))) \\
(pause) & = & (local\ s\ (now\ (await\ s))) \\
(exit\ s) & = & ((emit\ s); (pause)) \\
(trap\ s\ T) & = & (local\ s\ (watch\ s\ T)) \\
(present\ s\ T\ T') & = & (local\ t\ ((\\
 & & (thread\ (watch\ s\ (((pause);\ (thread\ T')); (emit\ t))))); \\
 & & (now\ (((await\ s);\ thread(T));\ (emit\ t)))); \\
 & & (await\ t)))
\end{array}$$

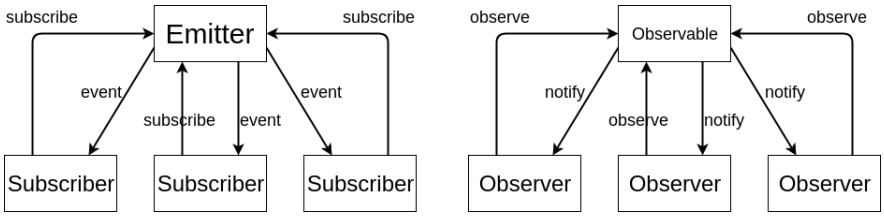
Figure 1: Derived SRP Constructs

The instruction $(await\ s)$ blocks the current thread until the signal s becomes present. The instruction $(loop\ T)$ executes a block of instructions repeatedly and indefinitely. The instruction $(now\ T)$ executes a block of instructions until the end of the current instant. The instruction $(pause)$ blocks the current thread for the rest of the instant. The *exit* and *trap* constructs provide a simple exception throwing mechanism, where $(trap\ s\ T)$ represents a $try{-}catch$ block and $(exit\ s)$ is represents the *throw* instruction. The instruction $(present\ s\ T\ T')$ will either run T in the current instant, if the signal s is present, or run T' in the following instant, if the signal s is absent. The *present* construct is especially important, because it shows that the model allows threads to react to the absence of a signal. In order to avoid causality cycles, the reaction can only happen in the following instant. For example, the program $(present\ s\ ()\ (emit\ s))$ is not erroneous, because it is consistent with a signal environment in which s is absent in one instant and present in the next.

2.3 RxJava

Reactive programming is a programming paradigm oriented around the concepts of data flow and propagation of change [5]. It is a layer of abstraction that makes it easier to reason about dependencies between components of a program. The core principle of this paradigm is that program components should not be concerned with the components that depend on them, but with the components that they depend on. Some components, called emitters, are sources of data or

have an internal state that can change. An emitter will propagate such events to registered subscribers, which perform event-specific operations called reactions. In reactive programming these reactions happen asynchronously [6]: subscribers do not coordinate and act independently from each other.



(a) Reactive Programming (b) The Observer Pattern

Figure 2: Reactive programming and the Observer pattern

Reactive Extensions [13] is a library that offers reactive constructs by extending the Observer Pattern from object oriented programming with operators for composing and manipulating data from functional programming. RxJava [14] is the implementation of this library for the Java programming language. In order to allow its users to use reactive programming concepts, the library offers the following:

- observables - sources of data
- subscribers - objects that listen to the Observables and react to the data
- operators for modifying and composing the data (*map*, *zip*, $filter$ etc.)

The main constructs from RxJava used to implement the interpreter are subjects. In RxJava, a subject is a combination between an observable and a subscriber. A subject can subscribe to another observable and propagate the data to its own subscribers, while optionally performing some intermediary operations on it. The library offers many types of subjects, but the two that were used in the implementation are:

- *PublishSubject* (*Fig.* 3)- emit all observed data to subscribed *Observer* objects
- *BehaviorSubject* (*Fig.* 4) - emits the most recently observed item upon subscription, and all subsequently observed items to each subscribed *Observer*

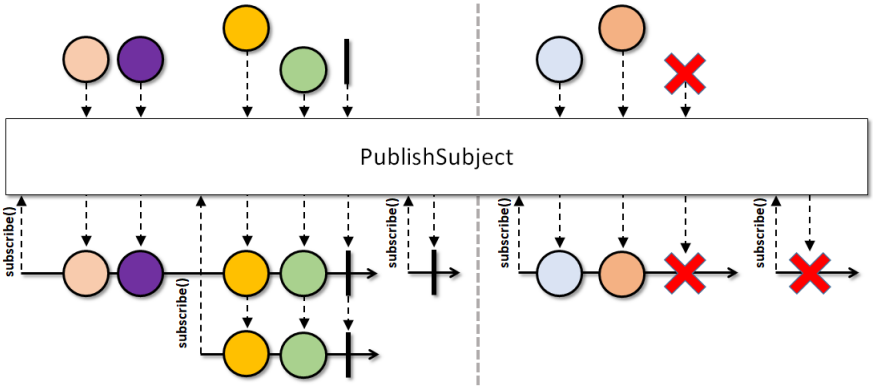


Figure 3: PublishSubject

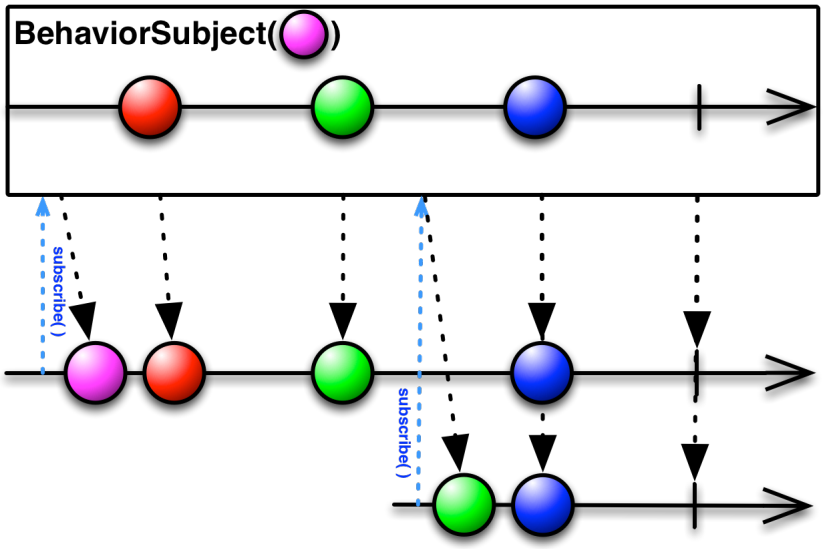


Figure 4: BehaviorSubject

3 Implementing the Interpreter

3.1 Design Decisions

To convert the theoretical model into a programming language, some constructs need to be added to the language in order to cover the parts of the model that are considered implementation details (i.e, how to declare the signal interface of a program):

- The *signal_domain* construct is introduced to allow a programmer to declare the signal interface of a program. The signals declared with *signal_domain* will be part of the signal environment for the entire runtime of the program. They can be set through input and will determine which emitted signals will be included in the output.
- The *let* keyword is introduced to allow a programmer to declare functions. This is needed to implement the construct $A(s)$ from the basic model, where A is an identifier for a thread that accepts parameters. A function's parameters can be either signals or program statements, including other functions.

Data types such as numbers, strings and maps (used as objects, inspired from JavaScript) were added to the language, along with relevant operators, in order to increase its expressiveness and make it a more robust proof of concept.

3.1.1 Additional Data Types

Two constructs were introduced to allow a programmer to declare variables that will hold non-signal values:

- The construct *shared_data* is used to declare global variables, with the same scope as the signals declared with *signal_domain*.
- The construct *shared* is used in a similar way as *local*, it binds a name in an instruction sequence to a non-signal variable.

Since these variables are shared between the threads of a program, their inclusion comes at a cost. For the language to allow for true parallelism while maintaining its property of determinism, access to resources that are shared between multiple threads needs to be synchronized in a deterministic fashion. For this purpose, the *lock* construct was introduced with the following behaviour:

- The instruction (*lock* r A) will request the "lock" for the shared variable r and run A in the same thread as soon as the request was granted by the scheduler. After running A, the thread yields its ownership of the "lock" of the resource r. A thread with an active *lock* instruction that did not receive the "lock" for the requested resource is considered suspended.

- In order to read or write to a shared variable, a thread must "own its lock" (i.e, be recognized by the scheduler as the only thread allowed to access said resource). Attempts to access a resource outside a *lock* instruction involving that resource will result in an error.

- The scheduler will keep track of all the resource requests and wait for all threads to block. Once that happens, the scheduler will fulfil these requests based on the following approach:

  1. If a resource was requested by only one thread, that thread receives the "lock" for that resource.
  2. If a resource was requested by more than one thread, its "lock" is granted to the thread with the highest priority.

  In order for the scheduler to be deterministic, this approach has to be deterministic, which it is, provided that the priority of the threads is chosen in a deterministic fashion. In this implementation, the following deterministic function was chosen to determine the priority of the threads:

  Definition 4. *Given two distinct threads, T_1 and T_2, T_1 has a higher priority than T_2 if one of the following statements is true:*

  - T_1 is an ancestor of T_2 (parent of parent of...)
  - T_1 has the same parent as T_2, but T_1 was sequentially spawned before T_2 (the instruction (*thread* T_1) appears before the instruction (*thread* T_2) in the instruction sequence of their parent thread)
  - T_1 has an ancestor that was sequentially spawned before T_2 or an ancestor of T_2

- A thread will release all its resources before blocking, this can be caused by *lock* or *when* instructions nested within a *lock* instruction. In case of a nested *lock* instruction, the thread will release its resources then make a request for both the resources it had and the ones mentioned in the *lock* instruction. In the case of a nested *when*, the thread will request its released resources after the *when* instruction allows execution to proceed.

- It is important to mention that an instant cannot end while some threads are suspended by a *lock* instruction. The scheduler should fulfil their requests and allow execution to proceed within the same instant.

The primitive data types added to the language are integers, floating-point numbers, booleans and strings. To allow a programmer to combine these types into data structures, the construct (*struct* x) was introduced, which will store a map into the variable x. Tables 1 and 2 show the operators that were implemented on these data types.

The following constructs are also introduced in order to allow decisions to be taken based on variable values:

- The (*loop cond T*) construct will run the instruction block T if the condition *cond* evaluates to *true*, and repeat until the condition evaluates to $false$.
- The (if *cond* Tif $Telse$) construct will run one of the two instruction blocks depending of the value of *cond* (Tif if *cond* is *true*, $Telse$ if *cond* is $false$).

The construct ($structure\_fields$ S L) will produce a list (implemented using a structure) of strings representing the fields of the structure S and store that list in L.

The (*print msg*) construct is used to write the contents of a variable to the output stream.

3.1.2 Syntactic Sugar

To make programming in this language more comfortable, most constructs will use a function notation and some will accept a variable number of arguments to replace multiple consecutive uses of the same construct. The redesigned constructs are shown in table 3.

Note : Because the scheduler is designed to grant requests from multiple threads at once, the construct $lock(r_1, ...r_n,\ T)$ behaves differently from the instruction block (*lock* r_1 (...(*lock* r_n T)...).

To allow users to create libraries and share code between multiple files, the *import* construct was introduced. This construct allows users to specify other files to parse for function definitions. For example, the code in figure 5 represents the standard library of the language, which includes the composite constructs mentioned in figure 1 from section 2.2.

Another useful syntactic shortcut is the ability to specialize functions by binding some of their parameters. Passing x parameters in a function call when the function takes n parameters will result in a function that takes $n - x$ parameters. For example:

```
let foo(a, b, c) =...;

let bar(T) = local(s, T(s));

bar(foo(x, y)); // will result in foo being called with
                // parameters x, y, s
```

Operator	Description
=	Assignment. The left operand must be a pointer (variable or structure field).
+	Number addition and string concatenation. Type promotion follows the order string > float > int. A boolean can also be concatenated to a string.
-	Subtraction. The result will be an integer only if both operands are integers. The only other accepted operand type is float.
*	Multiplication. The result will be an integer only if both operands are integers. The only other accepted operand type is float.
/	Division. Performs integer division if both operands are integers and floating-point division if at least one is a float. No other data types are accepted.
%	Modulo. Can only be applied to integers.
.	Index operator. The left operand must be a structure, the right operand must be an identifier. Returns a pointer to the data in a structure field.
[ ]	Index operator. Behaves the same as ".", but the expression inside the brackets can return any type, which will then be converted to a string.

Table 1: SRL operators - part 1

Operator	Description
==	Equality. Can be applied to any types except structures. Returns false if the operands have different types or values. Returns a boolean.
!=	Inequality. Can be applied to any types except structures. Returns true if the operands have different types or values. Returns a boolean.
<	Less than. Can only be applied to integers or floats. Returns a boolean.
<=	Less than or equal. Can only be applied to integers or floats. Returns a boolean.
>	Greater than. Can only be applied to integers or floats. Returns a boolean.
>=	Greater than or equal. Can only be applied to integers or floats. Returns a boolean.
&& and	Logical and. Can only be applied to booleans. Returns a boolean.
|| or	Logical or. Can only be applied to booleans. Returns a boolean.
! not	Logical not. Unary operator. Can only be applied to booleans. Returns a boolean.

Table 2: SRL operators - part 2

Notation	Description
$emit(s_1, ...s_n)$	Shortcut for $(emit\ s_1; ...; emit\ s_n)$.
$local(s_1, ...s_n,\ T)$	Shortcut for $(local\ s_1\ (...(local\ s_n\ T)...)$.
$thread(T)$	Different notation for $(thread\ T)$.
$when(s_1, ...s_n,\ T)$	Shortcut for $(when\ s_1\ (...(when\ s_n\ T)...)$.
$watch(s_1, ...s_n,\ T)$	Shortcut for $(watch\ s_1\ (...(watch\ s_n\ T)...)$.
$lock(r_1, ...r_n,\ T)$	Allows the request of multiple resources at once.
$shared(r_1, ...r_n,\ T)$	Shortcut for $(shared\ r_1\ (...(shared\ r_n\ T)...)$.
$struct(r)$	Assigns a map to the variable r.
$struct\_fields(str,\ list)$	Different notation for $(structure\_fields\ S\ L)$.
$loop(cond,\ block)$	Repeats the execution of *block* while *cond* evaluates to $true$.
$if(cond,\ Tif,\ Telse)$	Executes Tif is *cond* evaluates to $true$, otherwise it executes $Telse$.
$print(msg)$	Prints the string representation of the value in msg to the standard output.

Table 3: SRL constructs

```
// The standard library of the Synchronous Reactive
// Language:

let await(s) = when(s,());

let now(T) = local(s,
    emit(s);
    watch(s, T);
);

let pause() = local(s, now(await(s)));

let exit(s) = (
    emit(s);
    pause();
);

let trap(s, T) = local(s,
    watch(s, T);
);

let present(s, Tif, Telse) = local(t,
    thread(
        watch(s,
            pause;
            thread(Telse);
            emit(t);
        );
    );
    now(
        await(s);
        thread(Tif);
        emit(t);
    );
    await(t);
);
```

Figure 5: SRL standard library: "stdlib.srl"

3.1.3 The Grammar

The resulting language can be described with the following grammar:

- The following tokens are ignored wherever they are matched:

```
WHITESPACE  ::=  " " | "\t" | "\n" | "\r";
COMMENT     ::=  "//" (~["\n", "\r"])*;
```

- The following tokens are accepted with priority from top to bottom:

```
IMPORT         ::=  "import";
SIGNAL_DOMAIN  ::=  "signal_domain";
SHARED_DATA    ::=  "shared_data";
NULL           ::=  "null";
TRUE           ::=  "true";
FALSE          ::=  "false";
LET            ::=  "let";
NOT            ::=  ("!" | "not");
OR             ::=  ("||" | "or" );
AND            ::=  ("&&" | "and");
COMP           ::=  ("==" | "!=" | "<" | "<=" | ">" | ">=")
ADDOP          ::=  ("+" | "-");
MULOP          ::=  ("*" | "/" | "%");
ID             ::=  ["a"-"z","A"-"Z","_"]
                    (["a"-"z","A"-"Z","0"-"9","_"])*;
STRING         ::=  "\""  ("\\" ("\\" | "n" | "r" | "\"")
                            | ~["\\","\n","\r","\""] )* "\"";
INT            ::=  ["0"-"9"]+;
FLOAT          ::=  (<INT>)?"."<INT>( ("e"|"E")("+"|"-")<INT> )?;
```

- A program in this language is described by the following production rules:

```
Program                  ::=  (ImportList)? SignalDomain SharedData
                              (StatementWithSemicolon | Declaration)* EOF;
ImportList               ::=  IMPORT ":" StringList ".";
SignalDomain             ::=  SIGNAL_DOMAIN ":" IdentifierList ".";
SharedData               ::=  SHARED_DATA ":" IdentifierList ".";
Declaration              ::=  LET ID "(" IdentifierList ")"
                              "=" StatementWithSemicolon;
StatementWithSemicolon   ::=  Statement ";";
Statement                ::=  OrExpression ("=" Statement )?;
OrExpression             ::=  AndExpression (OR AndExpression)*;
AndExpression            ::=  LogicalAtom (AND LogicalAtom)*;
LogicalAtom              ::=  (Comparison | NOT Comparison);
Comparison               ::=  Calculation (COMP Calculation)?;
Calculation              ::=  Term (ADDOP Term)*;
```

```
Term                ::=  Factor (MULOP Factor)*;
Factor              ::=  (Constant | "(" StatementSequence ")"
                         | ID (ParameterList | (Index)+ )? );
Index               ::=  ("." ID | "[" Statement "]");
Constant            ::=  (INT | FLOAT | STRING | NULL | TRUE
                          | FALSE);
StatementSequence   ::=  Statement (";"
                         (StatementWithSemicolon)* )?;
ParameterList       ::=  "(" (StatementSequence
                         ("," StatementSequence)* )? ")";
IdentifierList      ::=  ( ID ("," ID)* )?;
StringList          ::=  ( STRING ("," String)* )?;
```

- All functions that are implicit in the language (the redisigned constructs from section 3.1.2) will be matched by the `(Identifier ParameterList)` rule in `Factor`.
- In this language, a library is a collection of functions written into a different file. It is described by the following production rule:

  ```
  Library  ::=  (ImportList)? (Declaration)* EOF;
  ```

3.2 Implementation

To achieve actual parallelism, the *Thread* class from the Java standard library is used to run program instructions on different processor cores. To synchronize these threads in the manner described by the model in [3], the *Condition* class, which offers mechanisms for suspending and activating threads, is used in combination with RxJava constructs. The two components of the model that are depended on by a multitude of other components, such that they can conform to the Observer Pattern, are the signal environment and the scheduler. These two components will act as *Observables* and the threads will act as *Observers*. The two events that require propagation are signal emission and instant transition. The signal environment will notify threads whenever signals are emitted, while the scheduler will keep count on the number of active threads and, once they are all suspended, notify them about the transition into the next instant.

3.2.1 Abstract Syntax Tree

JavaCC [15] was used to generate a parser for the grammar described in section 3.1.3. This parser will produce an abstract syntax tree in the form of an object of class *Program*, that can execute the instructions described in the parsed code-file. An abstract syntax tree [12] is an intermediate representation that captures the essential structure of the source code in a tree form, while omitting unnecessary syntactic details.

In order to encode program instructions, a common interface must be made for all program statements. The *Statement* interface was created to serve this

purpose. Every subclass of *Statement* must implement the *execute* method that takes a *SignalGuard* and a translation table as parameters and returns an *Object*. The *SignalGuard* object is responsible for managing the execution flow in relation to the environment (signals and resources). Its characteristics are discussed in detail in section 3.2.3. The translation table maps a *String* (the name of a variable) to its corresponding *Statement*. This is needed because statements can change names when passed to functions. For example, in the following code snippet, the s variable inside foo needs to be translated to the global signal a:

```
signal_domain: a.
let foo(s) = emit(s);
foo(a);
```

All statements must return something, but it can be of any type, since *Object* is the superclass of every other class in Java. This is particularly useful for statements such as mathematical operations, but all *Statement* subclasses are subjected to this requirement to allow for future extensions - such as function return values for example. The classes that implement the *Statement* interface are:

- *StatementSequence* - contains a list of *Statement* objects. Its *execute* method will execute each statement in the list, making use of the *executeStatement* method in *SignalGuard* (see 3.2.3). This is necessary because the execution of a *StatementSequence* can be paused or aborted after or during any *Statement* in the list. The method returns *null*, but this can be changed to extend the language to allow the assignment of lambda functions to variables.

- $Identifier$ - represents the name of a variable. It offers a *translate* method that will return the statement corresponding to the variable name in the provided scope (translation table). Its *execute* method will construct a $FunctionCall$ object and call *execute* on it.

- *Parameter* - binds a *Statement* object to a translation table. This is needed because function parameters have to use the scope of the caller, for example:

```
// this code emits signals a and b
signal_domain: a, b.
let foo(b, T) = (
    emit(b); // emits a (passed as parameter)
    T(); // T is evaluated to emit(b),
         // but b refers to the global signal,
         // not the local variable b
);
foo(a, emit(b));
```

- *Value* - holds a value of any type (*Object* type) and its *execute* method returns it. It is used for constants and data variables (pointers). The empty statement () is encoded as a *Value* object that holds *null*.

- *FunctionCall* - represents an identifier followed by a list of parameters. At this stage, the parameters are of type *Statement*, and are not yet bound to a scope. The *execute* method will translate the *Identifier* and call execute on the obtained *Statement* if it is a *Parameter* or a *Value*. The former is useful for when a function or a *StatementSequence* is passed as parameter, while the latter is simply necessary because of how the *Identifier* class is designed. If the translation result is an *Identifier*, then a function is called through the method *callFunction* in class *Program* (explained in detail further in this section). Because parameters might be passed to a *Parameter* object, which could pass them on to an *Identifier*, all 3 classes that implement function calls, *Identifier*, *Parameter* and *FunctionCall*, also offer an *execute* method that takes in a list of parameters (*Statement* objects) as an additional argument. This also implements the partial function calls described in section 3.1.2.

- *ThreadedStatement* - used to run a scope bound *Statement* (a *Parameter*) on a different thread. Upon instantiation, a new *SignalGuard* is created, since new threads do not carry on the constraints (when, watch or lock) of their parent threads. The *execute* method will register a new thread with the program's *Scheduler* (see section 3.2.2), then start a new thread that will execute the bound *Statement* with the new *SignalGuard*.

- *BinaryOperation* - is actually an abstract class that acts as a superclass for the implementations of the operators described in section 3.1.1. It contains two data fields of type *Statement*, and offers two methods for returning the result of each operand by calling their respective *execute* methods. The classes that derive from *BinaryOperation* are: *Assignment*, *IndexOperation*, *Addition*, *Subtraction*, *Multiplication*, *Division*, *Modulo*, *Equality*, *LessThan*, *GreaterThan*, *LogicalNot*, *LogicalOr*, *LogicalAnd*.

A *Program* will store the functions that are implicitly defined within the language, as well as the functions defined using the *let* construct, in a data field called *functions*, which maps a function name to its corresponding implementation. The interface *FunctionDefinition* was created to be implemented by all classes that encode function implementations. These derived classes must offer a *call* method that takes as parameters a *SignalGuard* and a list of *Parameter* objects.

The *UserFunctionDefinition* class is used to encode functions defined with the *let* construct. It has two data fields: a list of names for the function parameters and a *Statement* object encoding the function body. Its *call* method, which implements the function *call* from the *FunctionDefinition* interface, will create a new translation table for the local scope of the function, then

pass it to the *execute* method of the *Statement* representing the function body, along with the *SignalGuard* argument. When creating the translation table, the function will start from the global scope, which can be retrieved by calling the method *getGlobalScope* on the *SignalGuard* argument, which in turn uses the *getGlobalScope* method in *Program*. A local structure referred to by the identifier *here* is added to allow local function computations. This version of the table will then be extended by mapping every name in the list of parameter names to its corresponding *Parameter* object from the list passed as argument to *call*. All naming conflict are solved by overwriting, for example:

```
signal_domain: a, b, c.
let foo(s, s, s) = emit(s);
foo(a, b, c); // emits c
```

The encodings of functions that are implicitly declared within the language are described in detail in section 3.2.5.

The *Program* object will store all the program statements in a *ThreadedStatement* object, so that the interpreted program will run on a different thread than the main thread of the interpreter. This separation is required because when all program threads block, another thread needs to perform the transition from one instant to the next. Once the *Program* object is constructed, calling *execute* on it will run the encoded instructions. This function consists of a loop that samples the input signals, executes one instant (the program is run until all threads block), then outputs all the signals from the signal interface that were emitted during that instant. In the current implementation, the interpreted program will use the input and output streams of the interpreter.

3.2.2 Scheduler

The *Program* object also contains a *Scheduler* object. This object is responsible for assessing when an instant has ended, perform the transition from one instant to the next, and implement the synchronized locking mechanism described in section 3.1.1. The *Scheduler* class contains the following data members:

- *threadOrder* - an ordered list that holds the ID's of the threads that make up the interpreted program, ordered from the highest to the lowest priority (see the priority function in section 3.1.1)
- *threads* - a table that maps each thread id to relevant information about the thread, such as its number of children, its parent id, its corresponding *SignalGuard* object and the list of resource requests. All this is needed to establish the priority of the threads and handle their resource requests.
- *resources* - a table that maps each resource (pointer) to whether it is available or currently owned by a thread
- *activeThreads* - a counter for the number of active threads in the implemented program

- *instantLock* - a lock used to synchronize the methods of the *Scheduler* class, which modify the number of active threads, the thread data and the requests and ownership of resources

- *endInstantCondition* - a condition variable that uses the *instantLock*. The scheduler thread (the main thread of the interpreter) will sleep on this condition until the *activeThreads* counter reaches 0. When that occurs, the scheduler thread is notified to wake up and perform the transition to the next instant.

To perform its attributed tasks, the *Scheduler* class offers the following methods:

- *registerThread* - places a new entry in the *threads* table and the *threadOrder* list. For the former operation, the priority of the new thread is calculated, based off of its parent, then its id is placed at the correct position in the list.

- *deregisterThread* - called by a thread once it finishes execution. It simply removes the entries in *threads* and *threadOrder* corresponding to said thread. It also decrements its parent's number of children, if the parent is still present in the *threads* table.

- *incrementActiveThreadCount* - increments *activeThreads*. This method should always be called by the thread that is about to spawn or awake another, not by the thread that is spawned or awoken. This is required to avoid the case where *activeThreads* reaches 0 before the awoken thread can call *incrementActiveThreadCount*.

- *decrementActiveThreadCount* - decrements *activeThreads*. If *activeThreads* reached 0, then the *signal* method of *endInstantCondition* is called to signal the scheduler thread to wake up.

- *requestResources* - appends a list of resource requests to the calling thread's list of requests in the *thread* table

- *releaseResources* - takes a list of resources (pointers) as parameters and sets their state as available in the *resources* table

- *nextInstant* - causes the calling thread to sleep until the number of active threads in the interpreted program reaches 0. When that happens, the *distributeResources* method is called to handle resource requests. If any threads were awoken, the current thread is again put to sleep and the same actions are executed upon wakeup. If no threads awaken, which means that no threads are waiting on *lock* statements, the function will perform the actions associated with the end of the instant, then terminate. This function was designed to be called by the scheduler thread. The actions associated with the end of the instant are passed as a parameter of type *Runnable* and are executed by calling *run* on it.

- *distributeResources* - goes through the request lists of the registered threads in the order of their priority (their order in the *threadOrder* list) and grants, for each thread, either all the requests, if all the requested resources are available, or none.

The *Program* class contains a data field of type *PublishSubject* (see section 2.3) called *endInstantSignaler* and offers a method called *subscribeToInstantSignaler* that allows *Observers* to subscribe to it. Each thread's *SignalGuard* will subscribe to this *Subject* upon construction. The function *execute* in *Program* will call *nextInstant* on the *Scheduler* data field to execute an instant of the interpreted program. The actions associated with the end of an instant (passed as a parameter of type *Runnable*) are:

1. Print the output.
2. Have *endInstantSignaler* notify its subscribers that the instant has ended and present them with the current signal environment. This needs to be done before resetting the environment so that threads can abort their activated *watch* statements.
3. Reset the signal environment (make all signals absent).

3.2.3 Signals

The signal environment (Def. 2) is encoded by the *signalTable* data field in class *Program*, which is an instance of class *SignalTable*. This class was designed as a wrapper class that appends the functionality of an *Observable* to a *Map*. This class consists of a table that maps signal names to boolean values (present or absent) and a *Subject* that allows *Observers* to subscribe and react to changes in the signal table. The type of *Subject* used here is *BehaviourSubject* (see section 2.3). This will ensure that subscribers are presented with the state of the signal environment at least once, even when no changes occur during subscription, which is useful for the case when a thread subscribes to wait on a signal that is already present. The data in the signal table can be changed via the following methods:

- *put* - adds a mapping from a signal name to a boolean value, overwrites the existing mapping if the signal name is already in the table
- *putAll* - takes a map from signal names to boolean values as parameter and adds all its contents to the signal table
- *resetSignals* - maps all signal names in the table to $false$ (absent)

All these methods will also cause the *BehaviourSubject* to notify its subscribers and present them with the modified signal environment. For observers to subscribe to it, the *SignalTable* class offers a *getObservable* method, which the threads can access via the *subscribeToSignalTable* method in *Program*.

Each thread of the interpreted program will have a *SignalGuard* object associated with it, that will interrupt the execution flow when the thread has to wait on some signals or resources and resume execution when the conditions are met or when a new instant starts. These object will make use of the following data members:

- *environment* - a reference to the *Program* object, used to interact with the signal environment and the scheduler
- *checks* - a list of the *when* and *watch* instructions which guard the execution of the current statement, along with their corresponding signals
- *remainingWhenChecks* - the set of signals the thread is waiting on due to *when* instructions. When a signal becomes present it is removed from the set. At the start of a new instant, when all signals are made absent, all signals from *checks* that are associated with a *when* instruction are copied into the set. Its main use is improving performance.
- *resources* - the list of resource locks needed for continuing execution. The thread may be holding these locks or waiting to receive them.
- *hasResources* - a boolean that takes note on whether the thread holds all the required resource locks, or none.
- *globalScope* - a translation table for the global scope of the thread, used as starting block for the scope of a function defined with the *let* construct. The global scope of a thread contains the shared variables declared with the *shared_data* construct as well as a *private* variable that is unique in each thread and does not require locking to access (the thread it corresponds to always holds its lock).
- *signalTableSubscriber* - the *Disposable* object obtained when subscribing to the signal environment, used to unsubscribe when no longer waiting on signals.
- *endInstantSubscriber* - the *Disposable* object obtained when subscribing to the scheduler for notifications about instant transitions, used to unsubscribe when the thread finishes execution.
- *blockCondition* - the condition variable used to wait on signals or resources
- *lock* - the lock used by *blockCondition*, also used to synchronize some methods that may be called from different threads
- *stopCount* - the number of instruction blocks guarded by nested *when* and *watch* instructions to abort due to an activated *watch* instruction

When a *Statment* object has to run a set of instructions, as is the case with *StatementSequence* or with a *loop* instruction, they have to use the *executeStatement* method offered by *SignalGuard*, because the instruction

block may contain instructions that affect the execution flow, such as *when*, *watch* and *lock*. This method takes the *Statement* object to be executed and the translation table encoding its scope as parameters. Before calling the *execute* method on the received *Statement* object, the method will perform the following checks and take action accordingly:

1. If the set of remaining when checks is not empty and *stopCount* is 0, then the thread needs to wait for some signals to become present before execution can proceed. To do so, the private method *waitOnSignalTable* is called to subscribe to the signal environment through the *subscribeToSignalTable* method in *Program*. This method takes a callable as parameter, to call when the signal environment changes, for which the *SignalGuard* object will pass its *checkWhen* method. Because the signal environment uses a *BehaviourSubject* to notify its subscribers, the *checkWhen* method will be called once upon subscription and then every time the signal environment changes. This happens either when a signal was emitted or when all signals were reset at the end of the instant. The *Subject* will pass a view of the signal table to the *checkWhen* method so that it can remove all the fulfilled when checks from the *remainingWhenChecks* set. If the initial call removes all the checks from the set, the *waitOnSignalTable* method allows execution to continue without waiting. Otherwise, the method will block the current thread by using the *wait* method of the condition variable *blockCondition*, until the number of remaining when checks reaches 0 or until *stopCount* differs from 0. In both these cases, the *checkWhen* method will use the *signal* method of *blockCondition* to wake up the sleeping thread. Once execution is allowed to continue, the *SignalGuard* will unsubscribe from the signal environment by calling *dispose* on its *signalTableSubscriber* field.

2. If *stopCount* differs from 0, no waiting will occur and the execution of the *Statement* object is skipped. A *stopCount* greater than 0 means that the current instruction has been aborted by a *watch* instruction.

3. If the boolean value *hasResources* is *false*, then the *SignalGuard* object will use the the *waitOnResources* method to register its resource requests with the scheduler (via the *requestResources* method) and the wait on *blockCondition* until these requests are met. When the scheduler grants resources to a thread, it calls the *grantResources* method of the *SignalGuard* object corresponding to that thread. This method will set *hasResources* to *true* and signal the sleeping thread to wake up.

The aforementioned methods *waitOnSignalTable* and *waitOnResources* will first call the *decrementActiveThreadCount* method of *Scheduler* before putting the thread to sleep, while the methods *checkWhen* and *grantResources* will call *incrementActiveThreadCount* before notifying the thread to wake up.

The *SignalGuard* class also offers the following methods for adding elements to the collections of signal checks and needed resources before executing a *Statement*:

- *executeWhen*
- *executeWatch*
- *executeLock*

The methods *executeWhen* and *executeLock* will also release the resource locks held by the thread before proceeding with the condition checks and statement execution. *executeLock* will also release the resources that are relevant to its call after the associated instruction block is completed or aborted. This way, a nested *lock* instruction will only release its share of resources.

The *SignalGuard* also performs the termination of instruction blocks within activated *watch* instructions. Upon construction, all *SignalGuard* objects subscribe to the scheduler via the *subscribeToInstantSignaler* method offered by the *Program* class, in order to receive notifications about transitions from one instant to the next. The *Callable* object passed to this function, to be called when the instant changes, is the *nextInstant* method of the subscribing *SignalGuard* object. The *Scheduler* object will call this method at the end of each instant, and pass the signal environment to it, it is reset. This allows threads to consider which *watch* conditions were satisfied during that instant, and abort their corresponding instruction blocks. The *nextInstant* method will search the *checks* list for the first activated *watch* instruction. According to the *abort* function described in section 2.2, Def. 3, a *watch* instruction is activated when the its corresponding signal is present and all the *when* instructions it is nested within are activated (their corresponding signals are also present). This is why the *checks* list preserves the order in which these instructions are nested within each other. If an activated *watch* instruction is found, *stopCounter* will be set to the number of checks that need to be removed from the list (from last to first, like a stack) so that the activated *watch* instruction is removed. While *stopCount* is greater than 0, the calls to *executeStatement* will not wait, nor execute the associated *Statement*. At the end of their execution, both *executeWhen* and *executeWatch* functions will remove their checks from the list and decrement *stopCount* accordingly. This way, all the remaining instructions in the block associated with the top-most activated *watch* instruction, including nested *watch* and *when* instructions, will be skipped.

3.2.4 Data Types

The non-signal variables of this language, encoded by *Value* objects in the abstract syntax tree, will behave similarly to those in Java, in the sense that they all represent pointers to the data. In order to implement these pointers, that also limit the access to their data to at most one thread at a time, the *LockedPointer* class was designed as a wrapper around the objects used for the data types of the language. A *LockedPointer* object will consist of an owner id,

which holds the id of the thread that currently has the resource lock, and a value, representing the referenced data, which is of type *Object*, so that it can refer to any type of data. The methods $getValue$ and $setValue$ restrict access to the data such that they throw an error when run by a thread that isn't the owner. Since a variable can contain a structure (produced by the *struct* construct), and all the structure fields are also *LockedPointer* objects, all pointers within a structure need to have a common owner. This is done by having all pointers share the same $OwnerId$ value, which is basically a wrapper around an integer variable, used to allows both sharing and modification. When the scheduler grants the lock of a resource to a thread, it'll call the *setOwner* method on the corresponding *LockPointer*, and pass the id of the thread that is receiving the lock. This method will set the value of the owner id for the object on which it was called, which will also affect all other *LockPointer* objects that are within the same structure.

The data primitives present in the language are integers, floating point numbers, boolean values, strings and structures. In order to implement them, it is sufficient to use the $Integer$, $Double$, $Boolean$, $String$ and $HashMap$ classes respectively, all present in the Java standard library.

A structure is implemented as a map from string values, representing the names of the structure fields, to $LockedPointers$. When adding a field to a structure, the new *LockedPointer* object needs to be constructed via the factory method $makeWithinStruct$, so that it uses the same $OwnerId$ object as the rest of the structure. The function *sameStructure* offered by the *LockedPointer* class can be used to check if two pointers belong within the same structure. Structure assignment to variables or structure fields, which would cause multiple pointers to reference the same data, is only allowed if the pointers are within the same structure. This is needed in order to disallow code that bypasses the deterministic locking mechanism, such as:

```
signal_domain: .
sared_data: a, b.
lock(a,
    struct(a);
    struct(a.x);
    lock(b,
        b = a.x; // error
    );
    a.x.y = 0;    // would modify b
);
```

3.2.5 Language Constructs

The language constructs are split into two categories: the ones that are used by the parser to interpret the code and build the abstract syntax tree and the ones that are implemented by classes that implement the $FunctionDefinition$ interface.

The former category consists of the following constructs:

- *import* - The files mentioned with this construct will be parsed using the *library* grammar rule described in section 3.1.3 and all the function definitions will become entries in the $functions$ table in $Program$.
- *signal_domain* - The identifiers listed with this construct will become keys in *signalTable*
- *shared_data* - The identifiers listed with this construct become $LockedPointer$ objects stored in the table for the global scope of the program
- *let* - The parser will encode the functions declared with this construct as $UserFunctionDefinition$ objects and store them in the $functions$ table in $Program$
- The various operators described in section 3.1.1 are parsed into objects from classes that extend the $BinaryOperation$ class. Each operation has its own class.

The latter category contains the following constructs, each having an associated class that implements $FunctionDefinition$:

- *emit* - Encoded as an instance of the $EmitFunctionDefinition$ class, its *call* method expects a list of $Parameter$ objects that contain $Identifier$ objects which translate to signal names. It will then pass each signal name to the *emitSignal* method from class $Program$, setting them as present in the signal environment.

- *local* - Encoded as an instance of the $LocalFunctionDefinition$ class, its *call* method expects a list of $Parameter$ objects that contain $Identifier$ objects, except for the last one, which should contain the instruction block to be run. It will then use the *addLocalSignal* method in class $Program$, which will produce a new signal name and add an entry for it into the signal table. The generated names will contain the symbol "#", followed by a number, in order to avoid name clashes with user-declared signals. New translation rules are introduced into the translation table of the last $Parameter$ object, in order for the identifiers of the locally declared signals to be mapped to the newly generated names. The last $Parameter$ is then executed.

- *thread* - Encoded as an instance of the $ThreadFunctionDefinition$ class, its *call* method expects a list containing a single $Parameter$ object. This object is used to construct a $ThreadedStatement$ object, whose *execute* method will execute the $Parameter$ object in a different thread.

- *when* - Encoded as an instance of the $WhenFunctionDefinition$ class, its *call* method expects a list of $Parameter$ objects that contain $Identifier$ objects, except for the last one, which should contain the instruction block to be run. It then uses the *translate* method from the $Identifier$ class to obtain the signal names, which will get passed to the $executeWhen$ method of $SignalGuard$ together with the instruction block to be executed.

- *watch* - Encoded as an instance of the $WatchFunctionDefinition$ class, its *call* method expects a list of $Parameter$ objects that contain $Identifier$ objects, except for the last one, which should contain the instruction block to be run. It then uses the *translate* method from the $Identifier$ class to obtain the signal names, which will get passed to the $executeWatch$ method of $SignalGuard$ together with the instruction block to be executed.

- *shared* - Encoded as an instance of the $LocalFunctionDefinition$ class, its *call* method expects a list of $Parameter$ objects that contain $Identifier$ objects, except for the last one, which should contain the instruction block to be run. It will then construct a new $LockedPointer$ object for each $Identifier$, and introduce new entries into the translation table of the last $Parameter$ object so that the identifiers are mapped to $Value$ objects containing the newly constructed $LockedPointer$ objects. Lastly, the method *declareResources* from class $Scheduler$ is used to register the new shared resources with the scheduler.

- *lock* - Encoded as an instance of the $LockFunctionDefinition$ class, its *call* method expects a list of $Parameter$ objects that contain $Value$ objects, except for the last one, which should contain the instruction block to be run. It then uses the *execute* method on the $Parameter$ objects that refer to the shared variables to obtain the $LockedPointer$ objects

representing the resources to be locked. These will get passed to the *executeLock* method of *SignalGuard* together with the instruction block to be executed.

- *struct* - Encoded as an instance of the *StructFunctionDefinition* class, its *call* method expects a list containing a single *Parame-ter* object, which in turn should contain a *Value* object. The *execute* method of the *Parameter* object is run to obtain the *LockedPointer* object that will store the structure. A new *HashMap* object, which maps strings to *LockedPointer* objects, is constructed and passed to the *setValue* method of the *LockedPointer* received as parameter.

- *struct_fields* - Encoded as an instance of the *StructFieldsFunctionDefinition* class, its *call* method expects a list containing exactly two *Parameter* objects, which in turn should contain *Value* objects. The *execute* methods of the *Parameter* objects are run to obtain the two *LockedPointer* objects, the first one being a structure and the second one being the pointer that will store the resulting list. The fields of the structure are retrieved by using the *keySet* method on the map that encodes it. A structure that imitates an array-list is then constructed and stored in the second *LockedPointer* object. The names of the fields of the first structure are stored as strings in this list.

- *if* - Encoded as an instance of the *IfFunctionDefinition* class, its *call* method expects a list containing exactly three *Parameter* objects. The first one should contain a *Value* object representing the condition, the second one should represent the instruction block to be run if the instruction is true and the third one should represent the instruction block to be run if the condition is false. The *execute* method will be called on the first *Parameter* object to obtain the value of the condition. This value should be of boolean type, and, based on its truth value, one of the remaining two parameters is executed.

- *loop* - Encoded as an instance of the *LoopFunctionDefinition* class, its *call* method expects a list containing exactly two *Parameter* objects. The first one should contain a *Value* object representing the condition, the second one should represent the instruction block to be run. Inside an infinite loop, the first parameter is executed to obtain the value of the condition, which is expected to be of boolean type. If this value is *true*, the second parameter is executed and the loop starts from the beginning. If the condition is *false*, the loop is aborted. The *loop* instruction can also be aborted by an enclosing *watch* statement. To account for this, the method *isAborted* of *SignalGuard* is called before each iteration of the loop. If it returns *true*, the loop is aborted.

- *print* - Encoded as an instance of the *PrintFunctionDefinition* class, its *call* method expects a list containing a single *Parameter* object, which in turn should contain a *Value* object. The *execute* method of the *Parameter*

object is run to obtain the *LockedPointer* object that contain the data to be printed. In order to preserve the deterministic property of the language, this function needs to obtain the lock of the output stream before writing to it. The resource pointer is obtained through the *getGlobalScope* method from *SignalGuard*. This is then passed to the *executeLock* method of the guard, together with a $PrintWithLock$ statement that will print the data once the guard executes it.

4 Evaluation

4.1 Implementation of the Basic Model

This section covers the relation between the operational semantics of the initial model, presented in section 2.2 and the behaviour of the interpreter. When discussing this relation, a translation needs to be made between the program states described by these semantics, and the states of the interpreter. Furthermore, it has to be shown that the interpreter also follows the presented transition rules, so if the model semantics describe a transition from program state A into program state B, then the interpreter needs to transition from the translation for state A into the translation for state B. The operational semantics for the language construct are described by the rules 1-16 and the definition of the *abort* function (Def. 3) from section 2.2.

$$\frac{}{((),\ E) \Downarrow^{\emptyset} ((),\ E)} \textit{ (rule 1)}$$

Rule 1 describes the terminated thread, which is interpreted as the completion of a call to the *execute* method of a *Statement* object. Once the *execute* method returns, the instruction represented by the *Statement* object can be considered completed, which is denoted in the model as the terminated thread, "()".

$$\frac{(T_1,\ E) \Downarrow^{P_1} ((),\ E_1),\ (T_2,\ E_1) \Downarrow^{P_2} (T',\ E')}{((T_1;\ T_2),\ E) \Downarrow^{P_1 \cup P_2} (T',\ E')} \textit{ (rule 6)}$$

$$\frac{(T_1,\ E) \Downarrow^{P} (T',\ E'),\ T' \neq ()}{((T_1;\ T_2),\ E) \Downarrow^{P} ((T';\ T_2),\ E')} \textit{ (rule 7)}$$

Rules 6 and 7 describe the behaviour of the sequential composition of instructions. This composition is converted by the parser into a *StatementSequence* object, which contains an ordered list of *Statement* objects representing the sequential instructions. The *execute* method of *StatementSequence* will call the *execute* method of each *Statement* in the list, one by one, in their respective order. The transition in rule 6 is implemented by the fact that after an instruction has terminated, the execution of the next one in the list will begin. The transition in rule 7 is implemented by the fact that an instruction will not be executed unless it is the first one in the list, or the previous instruction has finished execution (the *execute* method returned).

The constructs used in the following rules are interpreted as $FunctionCall$ objects, whose *execute* methods will use the $callFunction$ method in $Program$ to run the *call* method of the $FunctionDefinition$ objects that match the function identifiers.

$$\frac{A(x) = T,\ ([s/x]T,\ E) \Downarrow^P (T',\ E')}{(A(s),\ E) \Downarrow^P (T',\ E')}\ \textit{(rule 2)}$$

Rule 2 describes the execution of a thread that requires parameters, modelled by the interpreter via the method *call* in class $UserFunctionDefinition$. The parser will produce a $UserFunctionDefinition$ for each function declared via the *let* construct, and store them in the $functions$ table within the $Program$ object. This models the premise $A(x) = T$, where A is the function identifier, x is a parameter and T is the function body, stored as a $Statement$ object. When the statement $A(s)$ is executed, the *call* method of the corresponding $UserFunctionDefinition$ will first produce a translation table that maps the parameter names to their values. Afterwards, it will pass this translation table to a call to the *execute* method of the $Statement$ encoding T, effectively running the instruction $[s/x]T$. This way, as described by transition rule 2, the state reached by running the instruction $A(s)$ will coincide with the state reached by running the instruction T in which the identifier x was replaced by s.

$$\frac{}{((emit\ s),\ E) \Downarrow^{\emptyset} ((),\ E[s := true])}\ \textit{(rule 3)}$$

Rule 3 describes the behaviour of the *emit* construct. This construct is modelled by the *call* method in $EmitFunctionDefinition$, which will map the signal passed as argument to true in the $signalTable$ in the $Program$ object, then terminate. As $signalTable$ represents the signal environment, the execution of the $(emit\ s)$ instruction will lead to a state equivalent to $((),\ E[s := true])$, where E is the initial mapping within $signalTable$.

$$\frac{s' \notin dom(E),\ ([s'/s]T,\ E \cup \{s' \to false\}) \Downarrow^P (T',\ E')}{((local\ s\ T),\ E) \Downarrow^P (T',\ E')}\ \textit{(rule 4)}$$

Rule 4 describes the behaviour of the *local* construct, which is modelled by the interpreter via the method *call* in class $LocalFunctionDefinition$. This method will first create a new unique name for the local signal via the method $addLocalSignal$ in $Program$. This models the premise $s' \notin dom(E)$. The method will then produce a translation table that maps the local signal identifier to the newly created identifier and pass it to a call to the *execute* method of the $Statement$ encoding T, effectively running the instruction $[s'/s]T$. This way, as described by transition rule 4, the state reached by running the instruction $(local\ s\ T)$ will coincide with the state reached by running the instruction T in which the identifier s was replaced by a new, unique identifier.

$$\frac{}{((thread\ T),\ E) \Downarrow^{\{|T|\}} ((),\ E)}\ \textit{(rule 5)}$$

Rule 5 describes the behaviour of the *thread* construct, which is responsible for the parallel composition of instructions. The instruction $(thread\ T)$ will spawn another thread to run the instruction T in parallel, then terminate. This is modelled by the interpreter via the *call* method in class $ThreadFunctionDefinition$, which constructs a $ThreadedStatement$ object that encodes the instruction block T. Calling the *execute* method on a $ThreadedStatement$ will start a new Java $Thread$, [16], that executes the $Statement$ object encoding T in parallel with the other threads.

$$\frac{E(s) = false}{((when\ s\ T),\ E) \Downarrow^{\emptyset} ((when\ s\ T),\ E)}\ \textit{(rule 8)}$$

$$\frac{E(s) = true,\ (T,\ E) \Downarrow^{P} ((),\ E')}{((when\ s\ T),\ E) \Downarrow^{P} ((),\ E')}\ \textit{(rule 9)}$$

$$\frac{E(s) = true,\ (T,\ E) \Downarrow^{P} (T',\ E'),\ T' \neq ()}{((when\ s\ T),\ E) \Downarrow^{P} ((when\ s\ T'),\ E')}\ \textit{(rule 10)}$$

Rules 8, 9, and 10 describe the behaviour of the *when* construct, which is modelled by the interpreter via the method *call* in class $WhenFunctionDefinition$. This method will add a *when* check to the list of signal checks in the *SignalGuard* object corresponding to the current thread, then subscribe to the *SignalTable* object representing the signal environment. Rule 8 shows that the execution will halt while the signal mentioned in the *when* instruction is not present. The interpreter mimics this through the use of the *wait* method in class *Condition* [16], which will suspend the current thread until a different thread calls *signal*. Since the signal table is a *BehaviorSubject* 2.3, both on subscription and every time a signal is emitted, the method *checkWhen* in *SignalGuard* is called to check the state of the signal environment. If the required signals have become present, the execution proceeds, as indicated in rules 9 and 10, otherwise the thread remains suspended, as indicated by rule 8. Rule 10 shows that, although the instruction block T is executed, the transitions that compose its execution will carry the signal check corresponding to the *when* instruction until the terminated thread is reached. This is modelled in the interpreter by using a list for the signal checks and by having the *SignalGuard* object be passed to subsequent instructions, so that their execution is "guarded" by all the signals used in enclosing *when* and *watch* instructions. To adhere to rule 9, when the execution of T reaches the terminated thread, so does the *when* instruction, which is modelled by the removal of the corresponding check from the list.

$$\frac{(T,\ E) \Downarrow^{P} ((),\ E')}{((watch\ s\ T),\ E) \Downarrow^{P} ((),\ E')}\ \textit{(rule 11)}$$

$$\frac{(T,\ E) \Downarrow^{P} (T',\ E'),\ T' \neq ()}{((watch\ s\ T),\ E) \Downarrow^{P} ((watch\ s\ T'),\ E')}\ \textit{(rule 12)}$$

Rules 11 and 12 describe the behaviour of the *watch* construct, which is modelled by the interpreter via the method *call* in class $WatchFunctionDefinition$. This method will add a *watch* check to the list of signal checks in the *SignalGuard* object corresponding to the current thread, then execute the *Statement* that encodes the instruction block T. As shown by rule 12, the transitions that compose the execution of T will carry the signal check corresponding to the *watch* instruction until the terminated thread is reached. This is modelled in the interpreter by using a list for the signal checks and by having the *SignalGuard* object be passed to subsequent instructions, so that their execution is "guarded" by all the signals used in enclosing *when* and *watch* instructions.

$$\frac{(T,\ E) \Downarrow^{\emptyset} (T,\ E)}{(T,\ E)\ddagger} \textit{ (rule 13)}$$

$$\frac{\forall T \in P\ (T,\ E)\ddagger}{(P,\ E) \Downarrow (\lfloor P \rfloor_E, E)} \textit{ (rule 14)}$$

Rule 14 defines the program transition from one instant to the next. It makes use of rule 13 and the *abort* operation (Def. 3). Rule 13 defines the condition for a thread to be considered suspended. The only transition rules that fit this definition are rules 1 and 8. In the interpreter, the *Scheduler* object keeps the count of how many threads are active. This count is decreased when a thread goes into waiting due to a *when* instruction or when a thread finishes the execution of its instruction block. The count is increased when a thread is spawned or when a thread is resumes execution. It can be seen that the premise $\forall T \in P\ (T,\ E)\ddagger$ is equivalent with the count of active threads reaching 0, which is when the *Scheduler* performs the transition into the next instant. To perform the *abort* operation, the *Scheduler* will call the $nextInstant$ method of each *SignalGuard* object in the program, which in turn will call the $checkWatch$ method. This method will identify the top-most activated *watch* instruction, in which the signal is present and all the enclosing *when* instructions are also activated. If an activated *watch* was found, the *stopCount* value of the corresponding *SignalGuard* is set to the number of *when* and *watch* instructions that need to terminate so that the activated *watch* will terminate. While the value of *stopCount* is above 0, the interpreter will skip executing statements, decrementing *stopCount* with each terminated *when* or *watch* instruction. The value will reach 0 once the activated *watch* instruction was aborted along with its corresponding instruction block.

$$\frac{\exists T \in P\ \neg((T,\ E)\ddagger),\ (T,\ E) \Downarrow^{P'} (T',\ E'),\ (P \backslash \{|T|\} \cup \{|T'|\} \cup P', E) \Downarrow (P'', E'')}{(P,\ E) \Downarrow (P'',\ E'')}$$

(rule 15)

The instantaneous transitions, described by rule 15, are performed by simply letting the Java threads run until the count of active threads reaches 0.

$$\frac{(P,\ E_{I,P}) \Downarrow (P',\ E'),\ O = \{s \in Int \mid E'(s) = true\}}{P \xrightarrow{I/O} P'} \textit{(rule 16)}$$

$$\text{where: } E_{I,P} = \begin{cases} true & \text{if } s \in I \\ false & \text{if } s \in (Int \cup sig(P)) - I \ \} \\ undefined & \text{otherwise} \end{cases}$$

Rule 16 formalizes the input-output behaviour of a program. The input is represented by the set of signals in the interface that are present at the beginning of an instant, while the output represents the set of signals in the interface that are present at the end of the instant. In the interpreted language, the signal interface is declared using the $signal_domain$ construct. The signals declared using this construct are easily distinguishable from local signals because of their naming conventions. The names of local signals start with the symbol "#", while signals in the interface can only start with a letter or the symbol "_". This behaviour is modelled by the interpreter via the method *execute* in class *Program*, which contains a loop that will read the input signals, set them as present in the signal environment, allow execution to proceed until the end of the instant, then print all present signals that are part of the signal interface.

4.2 Parallelism

Some implementations of synchronous reactive languages, for example ReactiveML [11] and SugarCubes [10], translate their parallel instructions into sequential instructions, which are run on a single processor core. In this implementation, true parallelism is achieved through the use of the *Thread* class offered by the Java standard library. This class allows the parallel execution of program instructions on multiple processor cores. This has the benefit of speeding up the execution of instructions that do not depend on one another.

As an example, the program shown in figures 6 and 7 performs a bounded sum on a range of values. A bounded sum is a summation in which each addition is accompanied by a modulo operation in order to keep the result bounded to a range of values. Since this operation is associative, the program makes use of a group of worker threads that run in parallel and compute the sum over separate regions of the initial range. The threads will store their results in the shared array *results*, and the last thread to do so will also emit the signal *DONE* to notify the main thread that the worker threads have finished. The main thread will then compute the bounded sum over the elements of *results* to obtain the final answer.

To show the benefit of using true parallelism, this program is run with 1, 2, 3, and 4 worker threads on a machine with 4 processor cores, and the execution is timed. Figure 8 shows the output and the execution time in the increasing order of the number of worker thread. Figure 9 shows the decrease of the execution time as the number of worker threads increases.

4.3 Determinism

A deterministic program is a program that, given a certain input, will always produce the same output and the underlying machine will always pass through the same states.

Definition 5. Given a program P: P is deterministic $\Leftrightarrow \forall$ I input of P :
$(P \xrightarrow{I/O_1} P_1 \wedge P \xrightarrow{I/O_2} P_2) \Rightarrow (P_1 = P_2 \wedge O_1 = O_2 \wedge P_1$ is deterministic),
where $P \xrightarrow{I/O} P'$ represents the transition from the program state P with input I to the program state P' with output O.

A deterministic programming language is a language (set) that contains only deterministic programs.

Definition 6. Given a programming language L:
L is deterministic $\Leftrightarrow \forall\ P \in L :\ P$ is deterministic

The main benefit of using deterministic languages in software development is the guaranteed avoidance of harmful non-determinism, also known as undefined behaviour, in production code. The assurance that a program is deterministic comes with the following benefits:

1. A deterministic program allows formal reasoning about its output and behaviour.
2. If a deterministic program passes a test, it will always pass that test.
3. Bugs are easy to reproduce and fix.

Both the original model and the presented implementation fail to fully adhere to this requirement. While the transition function for a single thread T in a signal environment E is clearly deterministic, the non-deterministic ordering of parallel operations causes the implemented language to be non-deterministic.

In the case of synchronous languages, one can discuss about a weaker form of determinism, at the level of instants. A language that has a deterministic transition function from the program state at the end of an instant to the program state at the end of the next instant will also offer the benefits of deterministic programming languages.

In the original model, although there can be multiple states to which a program can transition within an instant, they all eventually transition to a single state that marks the ending of that instant. The final state of an instant, as well as its output, can be deterministically inferred from the final state of the previous instant by applying the rewriting rules on the program threads in any arbitrary order, until no further progress can be made. The reason why this order is not relevant for either the final state or the output is because of the following characteristics of the model:

1. Signals cannot be made absent. Once emitted, a signal will be present in the environment until the end of the instant.

```
// A thread-pool demo.

import: "stdlib.srl", "array_list.srl".

signal_domain: START, DONE.

shared_data: numWorkers, results.

// replace 1 with he number of worker threads:
parallelSum(1, 100000000, 100000000);

let parallelSum(poolSize, workSize, sumBound) = (
    lock(numWorkers, results,
        numWorkers = poolSize;
        array_list(results);
    );

    threadPool(poolSize, workSize, sumBound);

    await(DONE);

    lock(results,
        here.i = 0;
        here.sum = 0;
        loop(here.i != results.length,
            here.sum =
                (here.sum + results[here.i]) % sumBound;
            here.i = here.i + 1;
        );
    );

    print("Sum from 0 to "
            + (workSize - 1)
            + ", bounded by "
            + sumBound + ": "
            + here.sum + "\n");
);
```

Figure 6: parallel_sum.srl - part 1

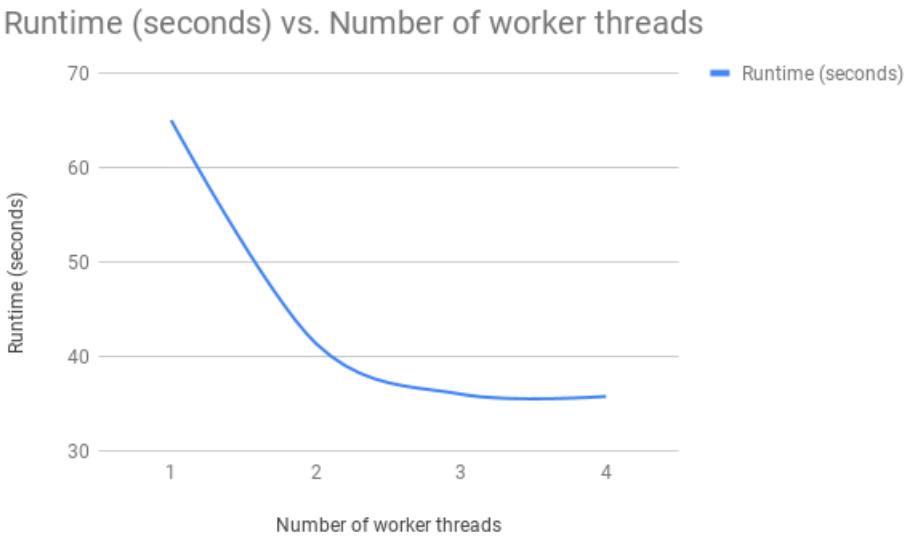
```
let threadPool(numWorkers, workSize, sumBound) = (
    here.chunkSize = workSize / numWorkers;
    here.start = 0;
    here.end = workSize % numWorkers + here.chunkSize;
    loop(here.start != workSize,
        spawnWorkerThread(here.start, here.end, sumBound);
        here.start = here.end;
        here.end = here.end + here.chunkSize;
    );
    emit(START);
);

let spawnWorkerThread(startVal, endVal, maxSumVal) =
    shared(start, end, sumBound,
        lock(start, end, sumBound,
            start = startVal;
            end = endVal;
            sumBound = maxSumVal;
        );
        thread(
            work(start, end, sumBound);
        );
);

let work(start, end, sumBound) = (
    await(START);
    lock(start, end, sumBound,
        here.i = start;
        here.sum = 0;
        loop(here.i != end,
            here.sum = (here.sum + here.i) % sumBound;
            here.i = here.i + 1;
        );
    );
    lock(numWorkers, results,
        array_list_push(results, here.sum);
        if(results.length == numWorkers,
            emit(DONE); // all workers finished
        ,
            ()
        );
    );
);
```

Figure 7: parallel_sum.srl - part 2

```
$ time echo "" | java -jar InterpreterSRL.jar
parallel_sum.srl
Sum from 0 to 99999999, bounded by 100000000: 50000000
Instant 0 ended. Output: DONE START

real    1m5.117s
user    1m3.544s
sys     0m0.324s

$ time echo "" | java -jar InterpreterSRL.jar
parallel_sum.srl
Sum from 0 to 99999999, bounded by 100000000: 50000000
Instant 0 ended. Output: DONE START

real    0m41.352s
user    1m20.364s
sys     0m0.316s

$ time echo "" | java -jar InterpreterSRL.jar
parallel_sum.srl
Sum from 0 to 99999999, bounded by 100000000: 50000000
Instant 0 ended. Output: DONE START

real    0m35.997s
user    1m43.276s
sys     0m0.396s

$ time echo "" | java -jar InterpreterSRL.jar
parallel_sum.srl
Sum from 0 to 99999999, bounded by 100000000: 50000000
Instant 0 ended. Output: DONE START

real    0m35.770s
user    2m14.068s
sys     0m0.784s
```

Figure 8: Output of parallel_sum.srl

Figure 9: Parallel efficiency example



2. A thread cannot react to the absence of a signal within the instant in which it is absent.

3. Signal emission cannot interrupt the execution of a thread within the current instant.

The combination of these properties implies that the more signals are emitted, the more progress can be made within an instant. Because of this, both the set of signals that will be emitted, as well as the state in which the threads will be at the end of an instant, can be deterministically evaluated, since one advances the other until no further progress is possible:

1. The signal environment is the union between the set of input signals and the set of signals emitted by activated *emit* instructions.

2. Each thread will stop in either the terminated state (), or at the first *when* instruction that contains a signal that is never emitted (or inputted).

Since the output of an instant is a set of signals, the signals have no value and they cannot be made absent, neither the order in which signals are emitted nor the number of times an already present signal is emitted will influence the output.

These properties also stand for the implemented language, since it follows the behaviour described by the original model. Therefore, as long as the additional constructs added to the implemented language are deterministic at the

level of instants, then so is the implemented language. These additional constructs are the data variables and the operations defined on them. Since each operation is deterministic in and of itself, the only cause for non-determinism would be the order in which they occur. The order of serialized operations is inherently deterministic, and the order in which parallel operations occur is made deterministic through the use of the synchronized locking mechanism described in section 3.1.1. This means that the state of the memory values at the end of an instant can be deterministically determined from the program state, memory state and input at the beginning of the instant.

As an example, the program shown in figures 10 and 11 simulates a simple soda machine. The program is composed of 6 threads that loop indefinitely. One thread will react to the signal `MENU` by displaying the list of available beverages and their respective prices, another thread will react to the signal `COIN` by incrementing the *credit* variable, and 4 threads that will react to purchases, one for each product. If the order in which concurrent purchases are handled would be arbitrary, the program would not be deterministic. This is not the case with the implemented language, because the order in which threads access the shared variable *credit* is deterministic, therefore the order in which concurrent purchases are handled is also deterministic. According to the priority function defined by Def. 4 in section 3.1.1, the order in which concurrent product orders are handled is `TEA`, `COFFEE`, `COLA`, `WATER`. This deterministic behaviour is illustrated in Figure 12, which shows the output of the program when all products are ordered within the same instant, emphasizing the order in which the *credit* variable is accessed and modified.

```
// A demo for a vending machine program.
import: "stdlib.srl".
signal_domain : MENU, COIN, TEA, COFFEE, COLA, WATER.
shared_data: credit.
lock(credit,
    credit = 0;
    print("Vending machine demo. Commands:\n
        MENU - display product list with prices\n
        COIN - increment credit\n\n");
);
thread(
    loop(true,
        when(MENU,
            print("\nMenu:\nTEA       2$\nCOFFEE   3$\nCOLA
                2$\nWATER    1$\n\n");
        );
        pause();
```

Figure 10: vending_machine.srl - part 1

```
    );
);
thread(
    loop(true,
        when(COIN,
            lock(credit,
                credit = credit + 1;
                print("\nCurrent credit: " + credit
                    + "\n\n");
            );
        );
        pause();
    );
);
let orderListener(signal, name, price) = thread(
    loop(true,
        when(signal,
            lock(credit,
                if(price <= credit,
                    credit = credit - price;
                    print("\nYour order: " + name
                        + "\nRemaining credit: "
                        + credit + "\n\n");
                , //else:
                    print("\nInsufficient credit: "
                        + credit + "\nPrice of "
                        + name + ": " + price + "\n\n");
                );
            );
        );
        pause();
    );
);
orderListener(TEA, "Tea", 2);
orderListener(COFFEE, "Coffee", 3);
orderListener(COLA, "Cola", 2);
orderListener(WATER, "Water", 1);
```

Figure 11: vending_machine.srl - part 2

```
Vending machine demo. Commands:
MENU - display product list with prices
COIN - increment credit

Instant 0 ended. Output:
COIN

Current credit: 1

Instant 1 ended. Output: COIN
COIN

Current credit: 2

Instant 2 ended. Output: COIN
COIN

Current credit: 3

Instant 3 ended. Output: COIN
MENU WATER COLA COFFEE TEA

Menu:
TEA      2$
COFFEE   3$
COLA     2$
WATER    1$

Your order: Tea
Remaining credit: 1

Insufficient credit: 1
Price of Coffee: 3

Insufficient credit: 1
Price of Cola: 2

Your order: Water
Remaining credit: 0

Instant 4 ended. Output: COFFEE TEA MENU COLA WATER
```

Figure 12: Output of vending_machine.srl

4.4 Limitations

Since the interpreter described in this paper is only a proof of of concept, it has a few limitations that can be resolved with further development.

One such limitation is the lack of static semantics checks to verify the validity of a program. The current implementation will check only at runtime if the interpreted program conforms to the language semantics. This will cause invalid programs run without being rejected, in cases where execution does not reach the invalid instructions. Performing these checks dynamically also lowers the computational performance of the interpreter, since each instruction has to be validated before execution. The language semantics make it possible for programs to be fully checked at compile-time or, in the case of an interpreter, while building the abstract syntax tree. Performing these checks statically would greatly increase performance and ensure the rejection of invalid programs.

Another limitation of the interpreter is the current intermediate representation it translates the code into. While the tree-like representation was chosen for convenience and efficiency, it makes formal analysis of the interpreter itself rather difficult. A more functional representation, with a list of statements and a rewriting function, would have made the behaviour of the interpreter much more intuitive and easy to formalise.

The current limitations on both parallelism and determinism are a result of the current design, and will not be solved with further development. A compromise has to be made between the two, either by using weaker versions of both properties or by sacrificing one in favour of the other. Most implementations of SRP languages available will translate parallel composition of instructions into sequential instructions it order to preserve determinism.

The current implementation only allows the use of *pure* signals. This design decision was made in order to avoid the problems of non-determinism and causality cycles introduced by having to merge multiple occurrences of the same valued signal within the same instant. To extend the interpreted language in order to allow valued signals, while also preserving the current property of determinism at the level of instants, the interpreter should either only accept associative and commutative functions or apply the merging operation in a deterministic order. Following the example of ReactiveML[11], in order to solve the problem of causality cycles, the merging function should only be applied after the instant has ended, when all the emitted values are known.

5 Comparison with Existing SRP Languages

This sections offers a general comparison between the implementation described in this paper and other existing implementations of synchronous reactive languages, namely Esterel [9], ReactiveML [11], and SugarCubes [10].

5.1 Esterel

Esterel is a system-design language specialized for reactive systems. Since the implemented model is inspired by the Esterel model, the two languages behave similarly in terms of their approach to the synchronous reactive paradigm. Both languages allow for the composition of parallel instruction blocks that use signals to coordinate their execution in a deterministic fashion. They also have the same input-output behaviour and definition of what an instant is. The key difference between the Esterel model and the implemented language is the ability to instantaneously react to the absence of a signal. This introduces the problem of causality cycles, which have to be detected and disallowed by the Esterel compiler. The implemented model solves this problem by postponing the evaluation of a signal as absent within an instant until after the instant has ended. Another difference between the two languages is the way they handle the compromise between determinism and parallelism. All Esterel statements and constructs are guaranteed to be deterministic. The compiler will check the given programs and ensure they are deterministic in terms of their input responses. The determinism of instantaneous state transitions will depend on whether the compiler implementation translates parallel composition to serialized operations or allows for "true" parallelism (the use of multiple processor cores). Esterel solves the problem of non-determinism caused by threads sharing data by making all variables local to their respective threads. Instead of sharing variables, the threads can send and receive data through the use of valued signals. This differs from the language described in this paper, which uses only *pure* signals in order to avoid the problem of non-deterministic merging functions, and allows threads to share variables, but imposes a deterministic synchronization mechanism for accessing them.

5.2 ReactiveML

ReactiveML is a functional synchronous reactive language designed for the implementation of interactive systems. It is based on the Synchronous Language described in [4], and therefore it also postpones the reaction to the absence of a signal until after the end of the instant. This makes it even more similar to the implemented language than Esterel. In terms of differences, ReactiveML allows the use of valued signals and runs its threads sequentially.

5.3 SugarCubes

SugarCubes is a set of Java classes used to implement reactive, event based, parallel systems. It follows the same approach to synchronous reactive programming as the implemented language and ReactiveML, and postpones the reaction to the absence of a signal until after the end of the instant. The constructs offered by SugarCubes follow similar design decisions as the implementation described in this paper - the signals are *pure* and threads can share variables. The main difference between the two implementations lies with the way they solve the problem of non-determinism as a consequence of parallelism. SugarCubes does not use multi-threading, so parallel operations are run sequentially on a single processor core in order to preserve determinism.

6 Conclusions

The design and implementation described in this paper offer insight into the relation between reactive programming constructs and synchronous reactive constructs, as well as how they relate to properties like expressiveness, determinism or parallelism. In conclusion, the implemented programming language meets all the functionality required to adhere to the Synchronous Reactive Paradigm. Since the interpreter was built in accordance with reactive programming practices, it serves as proof that reactive programming is sufficiently powerful to implement synchronous reactive programming. Understanding the relation between these two approaches, as well as the advantages and disadvantages offered by each one, can help with the design of reactive and interactive systems. This can be useful when deciding which technique is best suited for which task, or when attempting to combine the two within the same system.

7 References

[1] Nicolas Halbwachs: "Synchronous programming of reactive systems". Kluwer Academic Publishers, 1993.

[2] Robert de Simone, Jean-Pierre Talpin and Dumitru Potop-Butucaru. "The Synchronous Hypothesis and Synchronous Languages". Embedded Systems Handbook (2005).

[3] Roberto M. Amadio, Gérard Boudol, Ilaria Castellani, Frédéric Boussinot. Reactive concurrent programming revisited. Express, Sep 2006, France. Elsevier, 162, pp.49-60, 2006, Electronic Notes in Theoretical Computer Science, 162.

[4] F. Boussinot and R. De Simone: "The SL Synchronous Language". IEEE Trans. on Software Engineering, 22(4):256–266, 1996.

[5] E. Bainomugisha, A. L. Carreton, T. Van Cutsem, S. Mostinckx, and W. De Meuter: "A survey on reactive programming. ACM Computing Surveys", 45(4):52, 2013.

[6] Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson: "The Reactive Manifesto", 16 September 2014.

[7] Nicolas Halbwachs and Louis Mandel: "Simulation and Verification of Asynchronous Systems by means of a Synchronous Model". Proceedings of the Sixth International Conference on Application of Concurrency to System Design, 2006 (ACSD'06). Publisher: IEEE

[8] Paolo Baldan, Filippo Bonchi, Fabio Gadducci, and Giacoma Valentina Monreale: "Concurrency cannot be observed, asynchronously". Mathematical Structures in Computer Science, vol. 25, no. 4, pp. 978–1004, 2015.

[9] G. Berry and G. Gonthier: "The Esterel synchronous programming language". Science of computer programming, 19(2):87–152, 1992.

[10] F. Boussinot and J-F. Susini: "The SugarCubes tool box - a reactive Java framework". Software Practice and Experience, 28(14):1531–1550, 1998.

[11] Louis Mandel and Marc Pouzet. ReactiveML: "A Reactive Extension to ML". In ACM International conference on Principles and Practice of Declarative Programming(PPDP'05), Lisbon, Portugal, July 2005.

[12] JoelJones: "Abstract Syntax Tree Implementation Idioms". The 10th Conference on Pattern Languages of Programs 2003.

[13] http://reactivex.io/

[14] https://github.com/ReactiveX/RxJava

[15] https://javacc.org/

[16] https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html