Reactive Programming on the Bare Metal: A Formal Model for a Low-Level Reactive Virtual Machine

Bjarno Oeyen
bjarno.oeyen@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

Joeri De Koster
joeri.de.koster@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

Wolfgang De Meuter
wolfgang.de.meuter@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

Abstract

Reactive programming has many applications for embedded devices in IoT and other application domains. For these constrained devices it is crucial to bound the overhead of the execution of non-functional concerns such as glitch freedom. Reactive programming languages with static dependency graphs can implement glitch freedom by topologically sorting the dependency graph ahead of time. However, for more traditional reactive programming languages that support a dependency graph that can dynamically change, glitch freedom is typically implemented using a priority queue in which signals are enqueued according to their location in the graph. This is undesirable for embedded devices, as dynamically updating this priority queue has an undesirable, and sometimes unbounded, overhead. In this paper, we present a small-step operational semantics of a virtual machine specialised for running reactive programs that combines static ahead-of-time compilation of reactive programs into topologically sorted dependancy graphs while maintaining support for dynamic modifications to those dependency graphs.

CCS Concepts: • **Software and its engineering → Data flow languages**; • **Theory of computation → Operational semantics**.

Keywords: Reactive Programming, Internet-of-Things, Operational Semantics, Memory Consumption

ACM Reference Format:
Bjarno Oeyen, Joeri De Koster, and Wolfgang De Meuter. 2022. Reactive Programming on the Bare Metal: A Formal Model for a Low-Level Reactive Virtual Machine. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Reactive programs are made up from time-varying signals. Whenever a data source on which a time-varying signal depends updates its value, that change is propagated through a *dependency graph* (which keeps track of the dependencies between the time-varying signals) to update that signal's value. The goal of reactive programming (RP) languages is to propagate these changes in a glitch-free manner [7]. A glitch occurs when a signal's value is updated before the signals that it depends on are updated. The standard explanation of a glitch uses the following example program: `(< time (+ time 1))` (where `time` is a time-varying signal that is automatically incremented over time). This program contains three time-varying signals: the `time` signal (σ_1), the signal corresponding with the `(+ …)` expression (σ_2) and the signal corresponding with the `(< …)` expression (σ_3). Whenever σ_1 (the `time` signal) is updated (e.g., by incrementing it by one), both σ_2 and σ_3 must also be updated as they depend on σ_1. However, if during this update phase (which we refer to as a *turn*), σ_3 is updated before σ_2, then the < computation might use an old value of σ_3 and (temporarily, depending on the semantics of the reactive programming language) produce false.

There are various kinds of solutions to prevent glitches (some of which are intended to be used in multi-core computing environments [12], or for distributed applications [20, 21]). The standard approach, proposed by Cooper and Krishnamurthi in [7], is to use a priority queue in which signals are ordered according to their *height* in the reactive program's dependency graph: signals with a smaller height are updated before those with a larger height. Signals that do not depend on any other signal are given height 0, and all others signals have a height that is strictly larger than the heights of its dependencies (usually, a signal's height is computed by incrementing the largest height assigned to its predecessors by one).

However, this approach can be quite inefficient. While individual operations in a priority queue data structure are usually quite efficient (both enqueueing and dequeuing in a heap data structure, often used to implement priority queues, have a worst-case time complexity of $O(\log n)$, where n is the size of the priority queue [8]), most reactive programming languages that employ a priority queue to schedule

signal updates have to constantly make use of these operations to correctly schedule the updates of signals, causing an overhead that (as this paper will show) is easy to avoid. Especially in reactive programs with a dynamic dependency graph (e.g., those with the presence of `if` forms), keeping the priority queue up-to-date with the changing heights can incur a hefty overhead.

While this overhead is mostly negligible on personal computers (and other, more powerful, hardware), it can potentially become significant on low-powered devices with limited processing power. For example, microcontrollers, especially those targeted at embedded devices in IoT applications, have only limited processing power (usually expressed in MHz, not in GHz) and limited memory capacity (KiB instead of GiB). If one wants to use an RP language for writing IoT applications, the RP language must make effective use of the limited resources of the embedded hardware which means that the priority queue solution is therefore not an option.

One approach, in which no priority queue (or another more complex data structure) is needed is to perform a topological sort of the reactive program's dependency graph [28]. However, this solution is inadequate for reactive programs with a dynamic dependency graph (i.e. one that changes while the program is running). Reactive programming languages usually provide the means to alter dependencies at-runtime. For example by allowing lifted functions to create new signals which can then be connected via built-in higher-order operators, the dependencies between the signals can change while the program is running. If special care is not taken, these operators can invalidate a pre-determined topological ordering of the older graph.

In this paper, we present Remus, a formal specification of a virtual machine that has been carefully designed to be usable for running reactive programs in low-powered computing environments, despite having support for reactive programs with dynamic dependency graphs. This approach, of using a virtual machine, enabled us to examine the individual computational steps that must bet taken during the evaluation of a reactive program in a much more fine-grained manner. This allowed us to design a general solution for allowing reactive programs to have a dynamic dependency graph, without needing to use a priority queue or another more complex data structure, independent of the original language. As such, we conjecture that the presented formalisation can be applied to multiple reactive languages, despite the design of Remus being considerably influenced by Haai [24]. The main abstraction of Haai, that is consequently also used by Remus, is that of a reactor. Reactors in Haai are mere descriptions of (uninstantiated) dependency graphs which can be (for example, by a compiler) topologically ordered ahead of time. Whenever there is a dynamic branching point in a reactor, the compiler has to generate

```
1  (define t (- seconds (value-now seconds)))
2  (define s (if (even? t) (/ t 2) (+ (* t 3) 1)))
```

Listing 1. FrTime program that defines different types of signals.

code that forces Remus can jump to another reactor's description before resuming, similar to function application. This approach is possible for the Haai language, as Haai programs consist of only reactors. Haai programs are not constructed by applying procedures that, as a side effect of being evaluated, affect the dependency graph (which is the case in e.g., FrTime [6], as we will explain in Section 2.1). As such there is no need to run a so-called *loader program* that is implemented in a non-reactive (e.g., functional, procedural…) language.

The structure of the rest of this paper is as follows. In Section 2 we present the evaluation model that influenced the design of our virtual machine by means of example code in Haai. In Section 3 we introduce Remus and present its syntax, semantic entities and the small-step operational semantics that describe Remus' run-time behaviour. In Section 4 we reflect on the design of Remus, discuss its compiler and ongoing research activities that we intent to solve in future extensions. In Section 5 we present a brief overview of related work, before concluding the paper in Section 6.

2 Approach

2.1 Problem Statement

There are different approach to reactive programming. In our approach, we think of reactive programs as nothing more than graphs whose nodes represent signals (which are to be updated in a certain way) and edges as the dependencies between signals. In a lot of reactive programming languages, these *dependency graphs* are constructed incrementally by evaluating *non-reactive* (e.g., imperative, functional…) code. We will explain this using the code example in Listing 1, which shows a reactive program implemented in FrTime [6]. FrTime's evaluation model is an adaptation of Scheme's [15]. Similar to Scheme, FrTime code is evaluated from top-to-bottom. First, a new variable named `t` is defined which is bound to the result of evaluating the expression `(- seconds (value-now seconds))`. During the evaluation of that expression the subexpression `(value-now seconds)` is evaluated first. So far, these are ordinary Scheme semantics. The result of evaluating this expression is the *at evaluation-time* value of the `seconds` signal (which is defined as a signal that carries the number of seconds since the Unix epoch). The value returned by `(value-now seconds)` is thus an ordinary number (n), and not a time-varying signal and is thus not be updated when `seconds` changes. Then, FrTime evaluates `(- seconds` n`)`, this returns a signal whose value is equal to the current value of `seconds`, minus n. Or in other

words, the - function is *lifted* to the reactive level such that it can be applied to the time-varying signals. This effectively creates a signal whose value is initially equal to 0, and is automatically incremented by one every second.

After `t` has been defined, the expression that defines the variable `s` is evaluated. This signal is defined using `if` which has the same semantics as Scheme's `if` if only the conditional expression would not evaluate to a conditional signal (which it does in this example, as `(even? t)` produces a signal). In that case, FrTime's lifts the `if`: creating a conditional signal which alternates between two *expressions*. Assuming that both definitions in Listing 1 are processed in quick succession, the signal bound to `t` will probably still be equal to 0 and thus FrTime will evaluate the conditional's consequent expression. The conditional signal's value is, from that point on, equal to the value of the signal returned by evaluating that expression. One second later (when the `t` signal becomes 1) the conditional signal changes from `#t` (true) to `#f` (false). FrTime will then evaluate the conditional's alternate expression and let the conditional signal's value be equal to the value of that signal, from that point on. Another second later, the opposite happens again. The consequent expression is re-evaluated (for the second time) and the conditional signal is connected to the result of this evaluation.

In summary, FrTime's evaluation model is similar to those of Scheme except that evaluating certain expressions will make incremental changes to a *globally-maintained dependency graph*. I.e. a reactive program in FrTime contains a *loader program* that constructs the dependency graph (which is the actual reactive progam). This loader program is not only executed once when the program is started, but also during the evaluation of lifted functions.

This tight embedding of Scheme in FrTime makes it challenging to predict, in general, the structure of a reactive program's dependency graph as it constantly relies on the evaluation of unrestricted Scheme expressions. To statically construct a dependency graph of a FrTime program (e.g., to perform a topological sort of the dependency graph), one needs to analyse the FrTime program in its entirety, incorporating in the analysis not only Scheme's original semantics but also FrTime's non-trivial extensions.

We have observed that the mixing of reactive and non-reactive code is not only limited to FrTime, but can also be found in other reactive languages, such as REScala [26] and Flapjax [18]. However, other languages and frameworks in which a dependency graph is constructed in a similar way include RxJS [17] and Akka [32]. The former constructs dependency graph-like data structures when running a JavaScript program, while the latter does so using Scala (or Java).

2.2 Evaluation Model

The evaluation model of reactive programs that we will employ is unlike FrTime's. Instead of evaluating expressions

```
1  (defr (main)
2    (def t (- seconds (deploy-time-value seconds)))
3    (def s (if (even? t) (/ t 2) (+ (* t 3) 1)))
4    (out s))
```

Listing 2. Translation of the FrTime program from Listing 1 into a Haai reactor.

that, as a side effect of being executed, alter a dependency graph, we make use of reactors [24]. Reactors are textual descriptions of dependency graphs which can be constructed directly, without having to evaluate non-reactive code. Reactors themselves are only descriptions of reactive programs. Instances of reactors, which we call deployments, are actually connected to time-varying signals and will react to changes.

The rest of this section, will show code examples in Haai, a reactive programming language that we have designed as part of our research endeavours into reactive programming. Haai's only computational unit is that of a reactor. A translation of Listing 1 into a Haai reactor is shown in Listing 2. Haai's syntax has been inspired by Scheme's syntax. And besides a few minor changes, the code is identical[1]. However, Haai's semantics are unlike those of Scheme or FrTime. Expressions like `(/ t 2)` and `(+ (* t 3) 1)` are *not* application expressions that, when evaluated, call (or apply) a procedure on a sequence of argument values. These expressions represent the deployment of another reactor: the instantiation (deployment) of a reactor on a sequence of (possibly) time-varying signals. We call these expressions *deployment expressions*. Programmer-defined reactors in Haai are always composed out of smaller reactors (e.g., `main` makes use of `-`, `even?`...). These reactors can be other programmer-defined reactors, or primitive reactors. This is different from (e.g.,) FrTime where such an abstraction is absent and where composition can only be supported by composing functions (procedures) which can, when applied, alter the globally-maintained dependency graph.

Reactor definitions in Haai correspond with a so-called *reactor graph*: a graph that shows the internal wirings that represent the reactor with various sources that have to be filled in when the reactor is being deployed. According to this definition, a reactor can have more than one source and sink nodes, which is shown in Listing 3. Sources can be added by providing additional symbols (names) after the name of the reactor (within the parentheses). Multiple sinks

[1]The only noteworthy difference is a different name used in place of `value-now`. The intent of `deploy-time-value`, however, is the same as that of `value-now` in FrTime: to get a signal's current value at a certain moment in time. In FrTime this is called `value-now` as it returns the value that a signal has at the moment that that expression is evaluated. In Haai, we call this `deploy-time-value` to emphasise the fact that reactors in Haai have to deployed (instantiated). As such, `deploy-time-value` remembers the value that a signal had at-deployment time.

```
(defr (sum-and-product x y)
  (def s (+ x y))
  (def p (* x y))
  (out s p))
```

Listing 3. Reactor definition of `sum-and-product`.

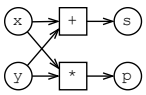

Figure 1. Reactor graph of the program in Listing 3. The boxes around + and * represent the at-runtime deployment of the + and * reactors, respectively.

are supported by adding more operand expressions to the `out` form, which by convention is the last expression placed in the body of a reactor definition. Figure 1 shows the corresponding reactor graph.

2.3 Design of the Virtual Machine

The design of Remus is thus inspired on Haai's evaluation model of reactors and deployments. Remus programs, however, are not simply a collection of reactor graphs from Haai. Remus does not make use of a dependency graph where signals are assigned heights and a priority queue to properly schedule signal updates. Remus executes instructions called *commands* that, when executed, allocate deployments, move data around and perform (primitive) computations. These commands are executed from top-to-bottom. As such, reactor definitions in Remus are *linear* descriptions of a possible ordering to update the values of the signals that make up a reactive program.

These linear descriptions are generated by pre-scheduling a reactor. A reactor is pre-scheduled by ordering the nodes (i.e. signals) of the reactor graph in topological order. Haai's reactors are easy to pre-schedule since their textual description corresponds directly with the dependency graph: there is no need to evaluate non-reactive code to construct the dependency graph. Furthermore, since a reactor graph does not contain nodes for the signals created by any inner deployments, each reactor can be pre-scheduled independent of one another. This will be the key idea that allows Remus to have support for reactive programs with dynamic dependency graphs.

2.4 Dynamic Deployments

The program in Listing 4 exemplifies how the dependency graph of a Haai program can be dynamic, by treating reactors as first-class values.

A deployment of `foo-or-bar` will, during a turn, make use of a deployment of either `foo` or `bar`, depending on the parity of `n` (a source signal of the `foo-or-bar` reactor). As `n` can change over time, the deployment used by `foo-or-bar` can change between turns.

```
(defr (foo) …)
(defr (bar) …)
(defr (foo-or-bar n)
  (def r (if (even? n) foo bar))
  (out (r n 10)))
```

Listing 4. Example of a dynamic deployment expression in Haai. The definitions of `foo` and `bar` are elided since they are not important for the example.

When talking about dynamic deployments, we make the distinction between *static* and *dynamic deployment expressions*, and *monomorphic* and *polymorphic branching points*[2] that they bring about. The former represents a *syntactic* distinction, while the latter a *behavioural* distinction. A static deployment expression is one where the operator expression refers directly (by name) to a defined reactor. E.g., the expression `(+ 10 temperature)` is a static deployment expression (assuming that + is bound to the primitive + reactor, and is not shadowed by another variable definition). If the operator expression refers to something else, it will be a dynamic deployment expression. E.g., the expression `(r n 10)` (in Listing 4). Dynamic deployment expressions can result in branching points that are either monomorphic or polymorphic, depending on the program itself and which values the operator signal can carry at-runtime. A polymorphic deployment, however, is always the result of a dynamic deployment expression. Finally, `if` expressions themselves are similar to dynamic deployment expressions, as the active expression depends on a run-time value.

Reactors with *polymorphic branching points* are supported by Remus. A special set of commands exists that will enable Remus Remus to not only swap to the correct *active deployment*, but also to create a new one if a deployment of the reactor to use does not exist yet. In other words, there is no need to analyse the program in its entirety to determine a topological sort of all the signals that make up a (running) program. This avoids the problem of having to analyse a reactive program in its entirety before a topological ordering can be generated that remains valid, even when the structure of the dependency graph changes.

3 Remus: A Virtual Machine for RP

In this section, we present Remus, a formal specification of a virtual machine for reactive programming[3].

$$
\begin{array}{rcl}
p \in \textbf{Program} & ::= & R \\
r \in R \subseteq \textbf{Reactor} & ::= & \mathcal{R}\langle x, \overline{c}_d, \overline{c}_r \rangle \\
c \in C \subseteq \textbf{Command} & ::= & C_{AllocMono}\langle x \rangle \\
 & | & C_{Supply}\langle o, l, i \rangle \\
 & | & C_{React}\langle l \rangle \\
 & | & C_{Consume}\langle l, i \rangle \\
 & | & C_{Global}\langle x \rangle \\
 & | & C_{Sink}\langle o, i \rangle \\
 & | & C_{MakePoly} \\
 & | & C_{AllocPoly}\langle o, l \rangle \\
 & | & C_{Primitive}\langle x \rangle \\
o \in \textbf{Operand} & ::= & v \mid l \\
l \in L \subseteq \textbf{Location} & ::= & \odot_D\langle i \rangle \mid \odot_R\langle i \rangle \\
 & | & \odot_I\langle i \rangle \mid \odot_O\langle i \rangle \\
v \in V \subseteq \textbf{Value} & ::= & i \mid b \mid x \\
\multicolumn{3}{c}{x \in X \subseteq \textbf{Name}} \\
\multicolumn{3}{c}{i \in \textbf{Number} = \mathbb{N},\ b \in \textbf{Boolean} = \{\text{true}, \text{false}\}}
\end{array}
$$

Figure 2. Syntax of Remus.

3.1 Syntax

The syntax of programs that Remus uses is shown in Figure 2. A reactive program in Remus consists of a set of reactor definitions R. A reactor definition is a name given to two distinct command sequences $\overline{c}_d$ and $\overline{c}_r$ that describe the behaviour of an individual reactor. The first sequence contains the *deployment commands*, which are only executed once per deployment of a given reactor (and initialise the deployment). The second sequence contains the *reaction commands*, which are executed whenever a deployment needs to update its sinks. An overview of the commands is presented in Section 3.1.1. Most commands have one or more *operands* (o) of which there are two types: values (v) and locations (l). Valid values in Remus are natural numbers (i), booleans (b) and names (x). Names are used to refer to reactors (which are uniquely identified by their name) and do not represent variables. Locations represent the address (relative to the current deployment) in which the current value of a signal can be found. There are four types of locations (represented by $\odot_D$, $\odot_R$, $\odot_I$ and $\odot_O$).

3.1.1 Commands. Table 1 presents a brief overview of all the commands present in Figure 2. Commands are used for the various kinds of operations that Remus needs to perform: moving values around, allocating and managing deployments and applying primitive computations. Each command is executed using a specific deployment as its context. The result of executing a command is, usually, stored within a deployma specificent. Commands that store values will almost always store a value in a deployment-specific memory region relative to the location of the command itself (one can see a deployment as a vector where each slot corresponds with a command in either command sequence that is used to store that command's result). For example the $C_{AllocMono}\langle x \rangle$ command will create a new deployment of a reactor named x, and store a reference to it in the deployment-specific memory region, relative to the location of the command itself. Commands can then refer to the output of other commands as if these would be signals from an RP language.

Command	Description	D	R
$C_{AllocMono}\langle x \rangle$	Allocates (creates) a new deployment of a reactor named x	✓	–
$C_{Supply}\langle o, l, i \rangle$	Supplies a value o (which can either be a location referencing a time-varying value, or a constant value) to the deployment stored in l (as the source value on position i).	–	✓
$C_{React}\langle l \rangle$	Will update the deployment stored in l and afterwards return and resume the evaluation of the current command sequence.	–	✓
$C_{Consume}\langle l, i \rangle$	Consumes a sink value (of position i) of the deployment stored in l.	–	✓
$C_{Global}\langle x \rangle$	Reads the current value of a global (primitive) time-varying signals named x.	–	✓
$C_{Sink}\langle o, i \rangle$	Stores the value of location o as one of the deployment's own sink values (position i).	–	✓
$C_{MakePoly}$	Creates a new branching point for dynamic deployment expression.	✓	–
$C_{AllocPoly}\langle o, l \rangle$	Creates a new deployment of the reactor identifier that can be found in o given the branching point identifier found in l, if it does not already exist. If it already exists, it reuses the old deployment.	–	✓
$C_{Primitive}\langle x \rangle$	Performs a primitive computation, as identified by x. E.g., $C_{Primitive}\langle + \rangle$ performs the addition of two numbers.	–	✓

Table 1. Summary of the commands of Remus. The columns **D** and **R** show a ✓ if a command is suitable to be placed in either the deployment command sequence or reaction command sequence of a reactor.

While this approach might not actually be optimal for an actual virtual machine (especially one intended to be used on small-scale embedded devices with limited memory as those mentioned in Section 1) as not all commands store data in a deployment's memory, we found this approach convenient to think about a reactive program's memory consumption (which will be discussed in much greater detail in Section 4). Still, if in the future a different memory layout is chosen, or another (more direct) approach is chosen to compile reactive programs for small-scale (embedded) devices, one can still use Remus' representation as an intermediate representation.

Table 1 also includes an overview of which commands are present in which the two types of command sequences. A ✓ under **D** indicates that a command will probably occur in the deployment command sequence, and a ✓ under **R** indicates those commands that can occur in the reaction commands sequence. This is not strictly enforced but indicates the locations where our current compiler (which is discussed in Section 4.3) places certain commands, and we deem it likely that future changes may change the location of certain commands. Therefore, we do not make a syntactical distinction between the two types of commands in Figure 2.

[2]This vocabulary has been inspired by Deutsch and Schiffman in [11].

[3]The operational semantics described in this section have been implemented using PLT Redex [16]. The source code of this implementation can be found online: https://gitlab.soft.vub.ac.be/boeyen/remus/.

r_{sap} =$\mathcal{R}\langle$ sum-and-product,
$\mathcal{D}$[⟨1⟩ $C_{AllocMono}\langle + \rangle$,
⟨2⟩ $C_{AllocMono}\langle * \rangle$],
$\mathcal{R}$[⟨1⟩ $C_{Supply}\langle \odot_I\langle 1\rangle, \odot_D\langle 1\rangle, 1\rangle$,
⟨2⟩ $C_{Supply}\langle \odot_I\langle 2\rangle, \odot_D\langle 1\rangle, 2\rangle$,
⟨3⟩ $C_{React}\langle \odot_D\langle 1\rangle\rangle$,
⟨4⟩ $C_{Consume}\langle \odot_D\langle 1\rangle, 1\rangle$,
⟨5⟩ $C_{Supply}\langle \odot_I\langle 1\rangle, \odot_D\langle 2\rangle, 1\rangle$,
⟨6⟩ $C_{Supply}\langle \odot_I\langle 2\rangle, \odot_D\langle 2\rangle, 2\rangle$,
⟨7⟩ $C_{React}\langle \odot_D\langle 2\rangle\rangle$,
⟨8⟩ $C_{Consume}\langle \odot_D\langle 2\rangle, 1\rangle$,
⟨9⟩ $C_{Sink}\langle \odot_R\langle 4\rangle, 1\rangle$,
⟨10⟩ $C_{Sink}\langle \odot_R\langle 8\rangle, 2\rangle$]⟩

Figure 3. Encoding of the `sum-and-product` reactor from Listing 3.

r_{fob} =$\mathcal{R}\langle$ foo-or-bar,
$\mathcal{D}$[⟨1⟩ $C_{AllocMono}\langle$even?$\rangle$,
⟨2⟩ $C_{AllocMono}\langle$if*$\rangle$,
⟨3⟩ $C_{MakePoly}$],
$\mathcal{R}$[⟨1⟩ $C_{Supply}\langle \odot_I\langle 1\rangle, \odot_D\langle 1\rangle, 1\rangle$,
⟨2⟩ $C_{React}\langle \odot_D\langle 1\rangle\rangle$,
⟨3⟩ $C_{Consume}\langle \odot_D\langle 1\rangle, 1\rangle$,
⟨4⟩ $C_{Supply}\langle \odot_R\langle 3\rangle, \odot_D\langle 2\rangle, 1\rangle$,
⟨5⟩ $C_{Supply}\langle$foo, $\odot_D\langle 2\rangle, 2\rangle$,
⟨6⟩ $C_{Supply}\langle$bar, $\odot_D\langle 2\rangle, 3\rangle$,
⟨7⟩ $C_{React}\langle \odot_D\langle 2\rangle\rangle$,
⟨8⟩ $C_{Consume}\langle \odot_D\langle 2\rangle, 1\rangle$,
⟨9⟩ $C_{AllocPoly}\langle \odot_R\langle 8\rangle, \odot_D\langle 3\rangle\rangle$,
⟨10⟩ $C_{Supply}\langle \odot_I\langle 1\rangle, \odot_R\langle 9\rangle, 1\rangle$,
⟨11⟩ $C_{Supply}\langle 10, \odot_R\langle 9\rangle, 2\rangle$,
⟨12⟩ $C_{React}\langle \odot_R\langle 9\rangle\rangle$,
⟨13⟩ $C_{Consume}\langle \odot_R\langle 9\rangle, 1\rangle$,
⟨14⟩ $C_{Sink}\langle \odot_R\langle 13\rangle, 1\rangle$]⟩

Figure 4. Encoding of the `foo-or-bar` reactor from Listing 4.

3.1.2 Basic Example.

Before moving unto the semantic entities and the reduction relation that we have defined for Remus, we first present a few examples of reactors encoded using the syntax of Remus. Figure 3 shows a possible encoding of the `sum-and-product` reactor from Listing 3.

To aid in the readability of this encoding, we have included a few annotations in grey. A $\mathcal{D}$ and $\mathcal{R}$ are used to tag both command sequences and numbers in front of the individual commands (⟨...⟩) denote the location of each individual command in its sequence. These annotations are not part of the syntax themselves and are simply there to aid the reader in understanding the $\odot_D$ and $\odot_R$ operands in the two command sequences.

The deployment command sequence of `sum-and-product` contains two commands: one to allocate a deployment of the `+` reactor, and one to allocate a deployment of the `*` reactor. When the `sum-and-product` reactor is deployed, these two commands will create their respective deployments, and store a reference to that deployment that can be referenced by the location of the command. The first reaction command will immediately make use of the `+` deployment. The C_{Supply} command will supply the first source value of the local deployment ($\odot_I\langle 1\rangle$, first operand) to the first source (1, third operand) of the deployment that is stored in memory in the location that belongs to the first deployment command ($\odot_D\langle 1\rangle$, second operand)[4]. The second reaction command does the same for the second source. The third reaction command (C_{React}) then signals the virtual machine that the deployment stored in $\odot_D\langle 1\rangle$ must react. This updates the sinks of the `+` deployment. The mechanism that drives primitive reactors will be explained later in Section 3.3.1. Once that deployment has updated itself, the computation of the `sum-and-product` deployment continues by getting the sink value of that deployment and storing it locally using the $C_{Consume}$ command (storing it in $\odot_R\langle 4\rangle$). The same thing that happened to the deployment of `+` now happens to the deployment of `*` in commands 5–8, storing the arithmetic product in $\odot_R\langle 8\rangle$. Finally, the sinks of the `sum-and-product` deployment are updated using the C_{Sink} commands which move the values in $\odot_R\langle 4\rangle$ and $\odot_R\langle 8\rangle$. Once these commands have been executed by the virtual machine, other deployments can look up the sink values by referencing $\odot_O\langle 1\rangle$ and $\odot_O\langle 2\rangle$.

Most (simple) reactor definitions will always follow this same pattern: allocate the nested deployments in the deployment command sequence, and in the reaction command sequence supply the values to the nested deployment, let it react, and then consume them locally. We call this the *Supply–React–Consume Pattern*.

3.1.3 Example with Dynamic Deployments.

Figure 4 shows how Listing 4 can be represented for Remus, using the same kind of annotations that were used in Figure 3. This example shows how dynamic deployments are supported. The third deployment command (a $C_{MakePoly}$ command, referred to by $\odot_D\langle 3\rangle$) creates a new branching point identifier and stores it (like almost every command) somewhere relative to that command's location. It is the $C_{AllocPoly}$ command (on position $\odot_R\langle 9\rangle$) that actually creates the deployment, depending on the run-time value of the first source signal (reaction commands 1–8 determine the reactor to use).

The $C_{AllocPoly}$ command in the deployment command sequence will allocate a new deployment of the reactor referred to by its first operand, unless such a reactor has already been deployed (for the same branching point) in a previous turn. The $C_{AllocPoly}$ command is placed in the reaction command sequence and will thus be able to create (allocate) new deployments while the program is running.

[4]Note that offsets in Remus use 1-based indexing. As Remus is a formal specification of a virtual machine, we did not see any reason to start counting from 0.

$$
\begin{array}{rcl}
k \in \textbf{Configuration} & ::= & \mathcal{K}\langle u, E, D, T\rangle \\
d \in D \subseteq \textbf{Deployment} & ::= & \mathcal{D}\langle \iota_d, x, b, \Sigma\rangle \\
u \in \textbf{Update Stack} & ::= & \text{halt} \mid \langle \iota_d, l, u\rangle \\
v & ::= & \ldots \mid \iota_b \mid \iota_d \\
\\
E \textbf{ (Value Environment)} & : & (X \to V) \\
\Sigma \textbf{ (Deployment Memory)} & : & (L \to V) \\
T \textbf{ (Toggle Environment)} & : & (I_b \times X \to I_d) \\
\multicolumn{3}{c}{\iota_b \in I_b \subseteq \textbf{BranchId}, \iota_d \in I_d \subseteq \textbf{DeploymentId}}
\end{array}
$$

Figure 5. Semantic entities of Remus.

We will discuss in Section 4.1.1 a possible solution which can avoid these run-time allocations.

Figure 4 highlights the use of a reactor named `if*`. The `if*` reactor implements an *eager* `if`: both the consequent signal and alternate signal are created and updated before the `if` selects one of them as its output. This is different from Haai's default *lazy* `if` which only keeps the signals activated that are needed for either the consequent or alternate signal. In the case of Figure 4, both are constant signals, so this distinction makes no difference here. However, Remus can support programs that need a lazy `if`. As long as a compiler compiles the consequent and alternate expressions into separate reactors, and the eager `if*` is used to first produce either that consequent reactor, or that alternate reactor which can then be deployed accordingly. If the consequent and alternate expressions contain free variables that are not free variables in the original reactor, the values of these variables (i.e. signals) can be provided as sources to the generated consequent and alternate reactors.

Note here that in function-based programming, once a function has finished being evaluated, its call frame can usually be removed from memory. In Remus, a deployment has to remain in memory as it may contain stateful information needed in consequent turns. Even if it is no longer active. Furthermore, in incremental update strategies for reactive programming, this is the intended behaviour as signals not affected by a change are usually not updated (while signals that are updated can refer to the current values of those that are not).

3.2 Operational Semantics

We now present Remus' semantics. We first describe the semantic entities that are used in the reduction relation, then show the construction of the initial configuration (which is the domain and codomain of the reduction relation), before showing the definition of the reduction relation itself.

3.2.1 Semantic Entities. The semantic entities are shown in Figure 5. A configuration entity that models the state of the virtual machine is typed k, and is a 4-tuple containing an update stack u, a value environment E, a set of deployments D and a toggle environment T. The update stack u is used to keep track of the locations in the command sequences that still have to be executed whenever one deployment is finished updating its own sink signals. The value environment E contains the values of all the global (top-level) signals which the deployments have access to read the value of these signals within a turn. The set of deployments D contains all the deployments that have already been created by the program, which contains the information about the reactive program's dependency graph. We explain its structure shortly. And finally, the T contains the toggle environment: a mapping from branching points (ι_b, *branching point identifier*) and reactor identifiers (names) to the corresponding deployment identifier. The toggle environment remembers all run-time deployments and is used to support dynamic deployment expressions (see Section 2.4).

Deployments themselves are also represented as a 4-tuple. Its structure, unsurprisingly, is different. The first element is the identifier of the deployment (ι_d) which is used to keep track of the distinct deployments that make up the reactive program. The second element is the name of the reactor (x) of which the deployment is an instance of, this is needed to look up the commands that have to be executed when updating a deployment's state. The third element is a boolean which represents whether or not the deployment has already been initialised (b). Or, in other words, whether its deployment commands (the commands in the deployment command sequence) have already been executed. The fourth element is the memory of the deployment (Σ) which is, essentially, a mapping from locations (l) to values that are stored on that location. In the formalisation, Σ is represented as an environment that can grow infinitely. However, since the number of locations of a single deployment is finite (as there are only a finite number of commands per reactor) these could be implemented as a vector of a fixed size (per deployment), as informally described in Section 3.1.1.

3.3 Initial Configuration

The reduction relation does not work on programs p, it operates on configurations ($\mathcal{K}$) as defined in Figure 5. The initial configuration k_{init} is defined as follows:

$$k_{init} = \mathcal{K}\langle \text{halt}, E_{init}, \{\mathcal{D}\langle \iota_{d,main}, \text{main}, \text{false}, \emptyset\rangle\}, \emptyset\rangle$$

Where $E_{init} = \{\text{time} \mapsto 0\} \cup \{x \mapsto \text{false} \mid \forall x \in X\}$ and X is the set of names of the global time-varying signals (such as `temperature`). These global signals represent the external signals that cause the signals in the reactive program to be updated. Their value will be updated at the transition from one turn to the next.

A few notes about k_{init}:

- The initial update stack of k_{init} is halt: this indicates to the reduction relation that the current turn has ended and that a new turn can begin.

- The initial value environment contains a mapping for all the global time-varying signals. When the program is initially started, these signals have no actual value and are thus, temporarily, set to `false`, until they are assigned an value (alternatively, there could be a unique value noting the absence of a value to distinguish scenarios where a signal's value is actually equal to `false`. The only signal with an actual value here is the signal named `time` which is automatically incremented at the start of each turn by the reduction relation.
- The set of deployments contains an initial deployment of a reactor called `main`, we assume in the rest of the paper that each program run by Remus has a `main` reactor (e.g., like shown in Listing 2) whose outputs are the outputs produced by the reactive program.
- Finally, the toggle environment (T) is initialised to $\emptyset$: which signifies an empty toggle environment.

3.3.1 Reduction Relation. The reduction relation $\longrightarrow$ describes the small-step operational semantics of Remus and is shown in Figure 6. The top of the figure shows the reduction rules that make up the reduction relation, and a textual summary of the helper functions, whose definition we deem trivial, as used by the reduction rules is shown in the bottom. By convention, these are typeset with $[\![\ldots]\!]$.

The NEXT-TURN rule detects if the current stack u is equal to halt, which signals that the current turn is over. It extracts the outputs of the deployment of `main` ($\overline{v}_o$) and gives these value to `sample_in` which determines the inputs of the time-varying signals which are to be processed in the next turn in fucntion of the current time (i_t) and the output values of the current turn ($\overline{v}_o$). The values produced by `sample_in` are then stored into E in which the time is also incremented by one, before (re)starting the computation to update all deployments. The way how NEXT-TURN interacts with an external environment via `sample_in` is related to how the `reactimate` function (e.g., from Yampa [9]) lets a reactive program interact with an external environment by continuously feeding the reactive program new inputs and acting on the output values that it produces.

The rules that are responsible for executing commands (ALLOCMONO, GLOBAL, SUPPLY, REACT, PRIMITIVE, CONSUME, SINK, MAKEPOLY, ALLOCPOLY-NEW and ALLOCPOLY-EXISTING) are very similar to each other and will not be explained separately. These rules all use the `get_command`$[\![\ldots]\!]$ to fetch the current command to ensure that a rule only can only be applied if the virtual machine is in the right state. Commands (and their corresponding rules) that modify a deployment's memory (e.g., GLOBAL) usually do so by storing a value (v) in the signal's memory (Σ) on the location of the command (l), as introduced in Section 3.1.1. Most of these rules should be easy to comprehend given the explanation in Table 1. Though, a few of these command-evaluating rules do require some additional explanation.

- The REACT rule does not modify any Σ (nor the Σ of the deployment which performs the C_{React} command, nor the deployment that has to react). It "pushes" a new frame on top of the update stack (u) that causes the specified deployment to be updated before continuing the update of the current deployment. The initial location (l_o) is different depending on whether the specified deployment has already been deployed (b_o). If it has already been deployed (i.e. it's deployment commands have already been executed earlier) then $l_o = \odot_R\langle 0\rangle$, if not $l_o = \odot_D\langle 0\rangle$. The D-TRANSITION rule (see below) will make sure that b_o will be updated accordingly.
- The PRIMITIVE rule is used to support primitive computations[5]. For example, the + reactor will make use of this command. It is represented as $r_+ = \mathcal{R}\langle +, [], [C_{Primitive}\langle + \rangle]\rangle$ and thus when a deployment of the + reactor needs to react, it only needs to execute the $C_{Primitive}$ command, which in turn triggers the PRIMITIVE rule. This rule takes all the source values supplied to the deployment (i.e. those stored in $\odot_I\langle i\rangle$), computes the result by applying δ_+ which serves as the escape mechanism for applying basic computations. Similar to reactors, primitive reactors can also produce multiple values, thus each δ function is able to return multiple values. These values are then stored in $\odot_O$ locations. Other primitive reactors follow the same principles as r_+.
- The ALLOCPOLY-NEW and ALLOCPOLY-EXISTING rules both evaluate $C_{AllocPoly}$ commands. The former can only be applied if for a given branching point (ι_b), no reactor named x_r has been deployed. The latter can only be applied if such a reactor has already been deployed (in a previous turn). The ALLOCPOLY-NEW rule is the only rule that modifies T, while the ALLOCPOLY-EXISTING rule is the only one that makes use of it (without modifying it).
- Finally, there are two rules that detect when there are no longer any commands in a command sequence to execute: D-TRANSITION and R-TRANSITION. The former detects when the deployment commands have all been executed, the latter does the same for the reaction commands. The D-TRANSITION rule updates the boolean stored in d to `true` (to remember that the deployment has been initialised by executing the deployment commands). The R-TRANSITION rule only needs to "pop" the current stack frame from the update stack (u). Both rules detect the end of their respective command sequences by seeing if (x, l) is in the domain of the helper function `get_command`$[\![\ldots]\!]$.

An important remark with regards to the presented reduction relation is that it has no support for handling erronous configurations. I.e. configurations which due to some kind

[5]Note that primitive computations are not the same as (lifted) functions. The virtual machine provides, as part of its standard library, a set of simple primitive computations that are assumed to be total and not dependent on any signal themselves as a means to perform simple computations on the values in V.

(NEXT-TURN)

$$\frac{i_t = E(\text{time}) \qquad \mathcal{D}\langle \iota_{d,main}, \text{main}, b, \Sigma_{main}\rangle \in D \qquad \overline{v} = \text{extract\_out}[\![\Sigma_{main}]\!] \qquad E_{sample} = \text{sample\_in}[\![i_t, \overline{v}]\!] \qquad E' = E[\text{time} \mapsto i_t + 1][x \mapsto E_{sample}(x) \mid \forall x \in \text{dom}(E_{sample})] \qquad l_{restart} = \begin{cases} \odot_D\langle 0\rangle & (b = \text{false}) \\ \odot_R\langle 0\rangle & (otherwise) \end{cases}}{\mathcal{K}\langle \text{halt}, E, D, T\rangle \longrightarrow \mathcal{K}\langle\langle \iota_{d,main}, l_{restart}, \text{halt}\rangle, E', D, T\rangle}$$

(ALLOCMONO)

$$\frac{C_{AllocMono}\langle x_{new}\rangle = \text{get\_command}[\![x, l]\!] \qquad \iota_{d,new} \text{ fresh} \qquad d_{new} = \mathcal{D}\langle \iota_{d,new}, x_{new}, \text{false}, \emptyset\rangle \qquad \Sigma' = \Sigma[l \mapsto \iota_{d,new}]}{\mathcal{K}\langle\langle \iota_d, l, u\rangle, E, \{\mathcal{D}\langle \iota_d, x, b, \Sigma\rangle\} \uplus D, T\rangle \longrightarrow \mathcal{K}\langle\langle \iota_d, \text{next}[\![l]\!], u\rangle, E, \{d_{new}, \mathcal{D}\langle \iota_d, x, b, \Sigma'\rangle\} \cup D, T\rangle}$$

(GLOBAL)

$$\frac{C_{Global}\langle x_{global}\rangle = \text{get\_command}[\![x, l]\!] \qquad v = E(x_{global}) \qquad \Sigma' = \Sigma[l \mapsto v]}{\mathcal{K}\langle\langle \iota_d, l, u\rangle, E, \{\mathcal{D}\langle \iota_d, x, b, \Sigma\rangle\} \uplus D, T\rangle \longrightarrow \mathcal{K}\langle\langle \iota_d, \text{next}[\![l]\!], u\rangle, E, \{\mathcal{D}\langle \iota_d, x, b, \Sigma'\rangle\} \cup D, T\rangle}$$

(SUPPLY)

$$\frac{\mathcal{D}\langle \iota_d, x, b, \Sigma\rangle \in D \qquad C_{Supply}\langle o, l_{local}, i\rangle = \text{get\_command}[\![x, l]\!] \qquad v = \text{fetch}[\![\Sigma, o]\!] \qquad \iota_{d,o} = \text{fetch}[\![\Sigma, l_{local}]\!] \qquad \Sigma'_o = \Sigma_o[\odot_I\langle i\rangle \mapsto v]}{\mathcal{K}\langle\langle \iota_d, l, u\rangle, E, \{\mathcal{D}\langle \iota_{d,o}, x_o, b_o, \Sigma_o\rangle\} \uplus D, T\rangle \longrightarrow \mathcal{K}\langle\langle \iota_d, \text{next}[\![l]\!], u\rangle, E, \{\mathcal{D}\langle \iota_{d,o}, x_o, b_o, \Sigma'_o\rangle\} \cup D, T\rangle}$$

(REACT)

$$\frac{\mathcal{D}\langle \iota_d, x, b, \Sigma\rangle \in D \qquad C_{React}\langle l_{local}\rangle = \text{get\_command}[\![x, l]\!] \qquad \iota_{d,o} = \text{fetch}[\![\Sigma, l_{local}]\!] \qquad \mathcal{D}\langle \iota_{d,o}, x_o, b_o, \Sigma_o\rangle \in D \qquad l_o = \begin{cases} \odot_D\langle 0\rangle & (b_o = \text{false}) \\ \odot_R\langle 0\rangle & (otherwise) \end{cases}}{\mathcal{K}\langle\langle \iota_d, l, u\rangle, E, D, T\rangle \longrightarrow \mathcal{K}\langle\langle \iota_{d,o}, l_o, \langle \iota_d, \text{next}[\![l]\!], u\rangle\rangle, E, D, T\rangle}$$

(PRIMITIVE)

$$\frac{C_{Primitive}\langle x_p\rangle = \text{get\_command}[\![x, l]\!] \qquad \overline{v}_i = \text{extract\_in}[\![\Sigma]\!] \qquad \overline{v}_o = \delta_{x_p}[\![\overline{v}_i]\!] \qquad \Sigma' = \text{insert\_out}[\![\Sigma, \overline{v}_o]\!]}{\mathcal{K}\langle\langle \iota_d, l, u\rangle, E, \{\mathcal{D}\langle \iota_d, x, b, \Sigma\rangle\} \uplus D, T\rangle \longrightarrow \mathcal{K}\langle\langle \iota_d, \text{next}[\![l]\!], u\rangle, E, \mathcal{D}\langle \iota_d, x, b, \Sigma'\rangle\} \cup D, T\rangle}$$

(CONSUME)

$$\frac{C_{Consume}\langle l_o, i\rangle = \text{get\_command}[\![x, l]\!] \qquad \iota_{d,o} = \text{fetch}[\![\Sigma, l_o]\!] \qquad \mathcal{D}\langle \iota_{d,o}, x_o, b_o, \Sigma_o\rangle \in D \qquad v = \Sigma_o(\odot_O\langle i\rangle) \qquad \Sigma' = \Sigma[l \mapsto v]}{\mathcal{K}\langle\langle \iota_d, l, u\rangle, E, \{\mathcal{D}\langle \iota_d, x, b, \Sigma\rangle\} \uplus D, T\rangle \longrightarrow \mathcal{K}\langle\langle \iota_d, \text{next}[\![l]\!], u\rangle, E, \mathcal{D}\langle \iota_d, x, b, \Sigma'\rangle\} \cup D, T\rangle}$$

(SINK)

$$\frac{C_{Sink}\langle o, i\rangle = \text{get\_command}[\![x, l]\!] \qquad v = \text{fetch}[\![\Sigma, o]\!] \qquad \Sigma' = \Sigma[\odot_O\langle i\rangle \mapsto v]}{\mathcal{K}\langle\langle \iota_d, l, u\rangle, E, \{\mathcal{D}\langle \iota_d, x, b, \Sigma\rangle\} \uplus D, T\rangle \longrightarrow \mathcal{K}\langle\langle \iota_d, \text{next}[\![l]\!], u\rangle, E, \mathcal{D}\langle \iota_d, x, b, \Sigma'\rangle\} \cup D, T\rangle}$$

(MAKEPOLY)

$$\frac{C_{MakePoly} = \text{get\_command}[\![x, l]\!] \qquad \iota_b \text{ fresh} \qquad \Sigma' = \Sigma[l \mapsto \iota_b]}{\mathcal{K}\langle\langle \iota_d, l, u\rangle, E, \{\mathcal{D}\langle \iota_d, x, b, \Sigma\rangle\} \uplus D, T\rangle \longrightarrow \mathcal{K}\langle\langle \iota_d, \text{next}[\![l]\!], u\rangle, E, \mathcal{D}\langle \iota_d, x, b, \Sigma'\rangle\} \cup D, T\rangle}$$

(ALLOCPOLY-NEW)

$$\frac{C_{AllocPoly}\langle o_r, l_b\rangle = \text{get\_command}[\![x, l]\!] \qquad x_r = \text{fetch}[\![\Sigma, o_r]\!] \qquad \iota_b = \text{fetch}[\![\Sigma, l_b]\!] \qquad (\iota_b, x_r) \notin \text{dom}(T) \qquad \iota_{d,o} \text{ fresh} \qquad d_{new} = \mathcal{D}\langle \iota_{d,o}, x_r, \text{false}, \emptyset\rangle \qquad \Sigma' = \Sigma[l \mapsto \iota_{d,o}]}{\mathcal{K}\langle\langle \iota_d, l, u\rangle, E, \{\mathcal{D}\langle \iota_d, x, b, \Sigma\rangle\} \uplus D, T\rangle \longrightarrow \mathcal{K}\langle\langle \iota_d, \text{next}[\![l]\!], u\rangle, E, \{d_{new}, \mathcal{D}\langle \iota_d, x, b, \Sigma'\rangle\} \cup D, T \cup \{(\iota_b, x_r) \mapsto \iota_{d,o}\}\rangle}$$

(ALLOCPOLY-EXISTING)

$$\frac{C_{AllocPoly}\langle o_r, l_b\rangle = \text{get\_command}[\![x, l]\!] \qquad x_r = \text{fetch}[\![\Sigma, o_r]\!] \qquad \iota_b = \text{fetch}[\![\Sigma, l_b]\!] \qquad \iota_{d,o} = T(\iota_b, x_r) \qquad \Sigma' = \Sigma[l \mapsto \iota_{d,o}]}{\mathcal{K}\langle\langle \iota_d, l, u\rangle, E, \{\mathcal{D}\langle \iota_d, x, b, \Sigma\rangle\} \uplus D, T\rangle \longrightarrow \mathcal{K}\langle\langle \iota_d, \text{next}[\![l]\!], u\rangle, E, \mathcal{D}\langle \iota_d, x, b, \Sigma'\rangle\} \cup D, T\rangle}$$

(D-TRANSITION)

$$\frac{(x, \odot_D\langle i\rangle) \notin \text{dom}(\text{get\_command})}{\mathcal{K}\langle\langle \iota_d, \odot_D\langle i\rangle, u\rangle, E, \{\mathcal{D}\langle \iota_d, x, b, \Sigma\rangle \in D\} \uplus D, T\rangle \longrightarrow \mathcal{K}\langle\langle \iota_d, \odot_R\langle 0\rangle, u\rangle, E, \{\mathcal{D}\langle \iota_d, x, \text{true}, \Sigma\rangle \in D\} \cup D, T\rangle}$$

(R-TRANSITION)

$$\frac{\mathcal{D}\langle \iota_d, x, b, \Sigma\rangle \in D \qquad (x, \odot_R\langle i\rangle) \notin \text{dom}(\text{get\_command})}{\mathcal{K}\langle\langle \iota_d, \odot_R\langle i\rangle, u\rangle, E, D, T\rangle \longrightarrow \mathcal{K}\langle u, E, D, T\rangle}$$

Helper functions:

get_command$[\![x, l]\!]$ retrieves the command stored for reactor x on location l. fetch$[\![\Sigma, o]\!]$ fetches o. If $o \in V$ then fetch produces o directly. Otherwise, it looks up o in Σ. This helper function allows commands like C_{Supply} and C_{Sink} to work with both values and locations as operands. next$[\![l]\!]$ increments the given l by one (which is used to go to the next command). extract_in$[\![\Sigma]\!]$ extracts the $\odot_I$s stored in Σ (in order). extract_out$[\![\Sigma]\!]$ extracts the $\odot_O$s stored in Σ (in order). insert_out$[\![\Sigma, \overline{v}]\!]$ inserts the values $\overline{v}$ into Σ as outputs ($\odot_O$s). sample_in$[\![i, \overline{v}]\!]$ samples given the current time i, and the current output values $\overline{v}$ of the reactive, the external environment, producing a E. And finally, $\delta_{x_p}[\![\overline{v}]\!]$ represents a function that performs a primitive computation. E.g., $\delta_+[\![i_1, i_2]\!]$ performs the addition of i_1 and i_2. δ_{x_p} functions can produce multiple values.

Figure 6. Small-step operational semantics of Remus.

of programmer error, no longer to reduce. The reduction relation will get stuck in cases where a command refers to a memory address (e.g., a source) for which no value has been assigned. This can happen if a deployment is not supplied the right number of values. As a consequence, proving progress (an important property of an RP language, that is related to liveness [1, 14]) given the reduction relation defined in Figure 6 is not possible given the presented reduction relation.

4 Discussion

4.1 Memory Consumption of Remus Programs

In Section 3.1.1, we mentioned that our current virtual machine design makes it easier (with respect to standard approaches) to reason about a reactive program's memory consumption and allocation behaviour.

As can be easily seen from our reduction relation, the only commands that allocate memory are the commands that, according to Table 1, can be placed in a deployment command sequence and $C_{AllocPoly}$. All commands that write to Σ always do so in the static location: either as defined by one of the operands themselves, or relative to the location of the command, not a run-time value. Thus, the size of each deployment's Σ can easily be derived statically.

Ignoring $C_{AllocPoly}$ for now, all commands in the deployment command sequence are only executed once for every deployment of a reactor. In a reactive program without dynamic deployment expressions, all these deployment allocations (thus memory allocations) happen only in the very first turn. Without dynamic deployment expressions, the number of deployments is static (as there are no polymorphic branching points) and the whole dependency graph can be pre-determined. This makes it easy to determine an entire reactive program's memory footprint if the deployments are known before-hand. The only entity in a configuration that can grow dynamically (between state transitions) is the update stack (u), of which an upper-bound on its required size can easily be determined in the absence of dynamicity in the reactive program's dependency graph.

4.1.1 Pre-determining the Toggle Environment.

However, if a reactive program uses dynamic deployment expressions, the above does no longer apply. These programs compile to programs that use the $C_{MakePoly}$ and $C_{AllocPoly}$ commands, where the latter is able to create (allocate) new deployments while reacting. These allocations can thus happen in turns after the first one. One possible approach could be to statically determine which reactors can flow over the time-varying operator signals of the dynamic deployment expressions. By doing so, one could (for programs that make principled use of recursion) pre-determine a T_{final} such that at run-time no new deployments need to be allocated (e.g., by the AllocPoly-new rule). Furthermore, in such an analysis, one can determine which dynamic deployment expressions are actually monomorphic and can be compiled into a $C_{AllocMono}$.

Such an analysis is essential in environments where one has to be absolutely certain that a reactive program can be run given a well-known memory limitation (e.g., in safety-critical applications). If such a guarantee is not needed, one can still use the memory model of Remus to reason about the memory consumption of individual reactor deployments, ignoring the memory requirements of any dynamic deployments that may be created while the program is running.

While such an analysis does require one to analyse the reactive program's dependency graph in full, we still deem our approach as an improvement over standard approaches which requires one to analyse both reactive and non-reactive code.

4.2 What Has Not (Yet) Been Formalised

The presented formalisation only covers the core semantics of Haai. In this section, we briefly discuss the most significant features of Haai that are missing in Remus. When possible, we briefly discuss a possible extension that would make this available to the presented formal specification.

Incremental Updates. First, the presented formal specification of Remus does not support any form of incrementality. In most reactive programming languages, only signals that depend on a signal that is updated are also updated by the RP runtime. Dormant signals, which are signals that are not affected by a change, should not be updated. In Remus, signals that should be dormant are also updated. In every turn, the whole dependency graph (represented by deployments with their corresponding command sequences) is updated. One possible modification that can easily be made to the formalisation is to give deployments a dirty bit which is initially set to `true` (each newly-instantiated deployment must always execute its commands in full). During the evaluation of C_{Supply} commands, the dirty bit is set to `true` when the value that is being supplied is different from the one that is currently stored in the deployment. The C_{React} command can then decide to skip the update of the nested deployment if the dirty bit is still set to `false`: if none of the sources changed their value, updating the inner signals will be pointless. This change, however, would not actually alter the semantics of the VM. However, as needlessly updating signals has a adverse impact on the performance, such a modification is probably well-advised for an actual implementation of Remus. Although, by doing so the time-per-turn could greatly vary over time (causing jitter), which may be undesirable for certain embedded systems.

Stateful Reactors. Second, Haai has support for stateful reactors using special variables called trampoline variables [23]. Trampoline variables (or stateful operators in general) can be modelled on top of the presented formalisation by providing new commands that identify, initialise, fetch, and update trampoline variables. Stateful operators from other RP languages (like `foldp` in Elm [10] and `pre` in RT-FRP [33]) can then be implemented using these same commands.

Run-Time Composition of Reactors. Remus has no support for the run-time *weaving* of reactors. Haai has a set of composition operators called weaving operators that combine reactors in a point-free style[22]. For example, the `parallel` weaving operators combines two reactors such that the inputs are divided (according to the input arities of the original reactors) between the two reactors. A possible solution that can happen at compile-time could be to compile expressions like `(parallel + *)` into a reactor like `sum-and-parallel` (see Listing 3 and Figure 1). However, weaving operators can be applied on time-varying signals in Haai, so this syntactic transformation is not always possible. However, the analysis proposed in Section 4.1.1 can help in solving this problem as the necessary reactors can then be generated at-compile time.

Anonymous Reactors with Lexical Scope. Finally, Remus has no support for captures, anonymous reactors with

lexical scope [24]. Reactors with lexical scope have, besides access to the globally-defined signals (which can currently be accessed using C_{Global} commands) also access to signals from another deployment (the deployment that created the capture). In the future, we aim to further research the characteristics of anonymous reactors with lexical scope, and consider this as part of our future work.

4.3 Compiler

The main focus of this paper has been on the design of the virtual machine: not on the compiler from Haai to Remus. Still, we wish to discuss two interesting areas with regards to the compilation of reactors.

First, the examples of reactor defintions in Remus presented in this paper (Figures 3 and 4) only show one correct compiled representation each. For example, the `sum-and-product` reactor's commands can be arbitrarily permuted as long as commands that refer to another command's output are placed after that command (i.e. in a glitch-free way). For example, the first C_{Sink} command can be placed directly after the $C_{Consume}$ command that fetches the sum. This of course is a consequence of the fact that a topological ordering of directed acyclic graphs (which reactor graphs are) is often not unique. A compiler is thus free to choose an arbitrary ordering as long as it is a valid topological sort. An important insight here is that a Haai interpreter (using a regular priority queue to schedule signal updates) is free to change the order in which signals are updated at *every* computational step (as long as it choses a signal to update in a glitch-free manner), while the order used by Remus programs is determined at-compile time for all turns. However, as long as all orders (both the one determined statically by the compiler and the ones decided at-runtime during every turn) ensure that signal updates happen in a glitch-free manner, both run-time strategies should be equivalent (i.e. produce the same values on the same signals).

Second, our compiler currently places commands according the validity shown in Table 1. In the future, we deem it likely that these validity restrictions might change if additional features (e.g., those mentioned in Section 4.2) are added to Remus, or if additional compile-time optimisations are introduced.

5 Related Work

Before we conclude this paper, we present an overview of other research projects that intent to make reactive programming a suitable approach for building software targeted at small-scale embedded (e.g., IoT) devices.

Reactive Programing Languages. ReactiFi [30] is a reactive programming language implemented as an embedded domain-specific language in Scala that compiles reactive programs directly into C code which can then compiled into a native binary for (e.g.,) WiFi hardware platforms. It does so by compiling both the lifted functions and the dependency graph. The latter is compiled similar to programs in Remus by topologically sorting the constructed dependency graph. However, unlike Remus, ReactiFi has only limited support for dynamic dependency graphs as it does not feature higher-order reactives.

Emfrp [28] is another RP languages that compiles reactive programs into C code. Emfrp lacks an abstraction for reactive computational units (i.e. reactors): all signals are defined directly in the top-level (which does not encourage reuse) and also lacks the ability to dynamically change the dependency graph.

CFRP [31] and Hae [35] are two reactive languages that are both similar to Emfrp, but are built on top Haskell (Emfrp uses its own syntax), inheriting its powerful type and macro system, and generate C++ code from a reactive program definition. Unlike Emfrp, CFRP and Hae do support dynamic changes in the dependency graph. It does so by using a queue data structure on which signals are enqueued when they can be updated. This is similar to the FrTime example from Section 2, except that the queue is used only for signals that are ready to be updated: each signal keeps track of the number of signals that it depends on and is only enqueued when it has received updates from all its dependencies. In the case of Remus, such a queue is not needed, even for programs with dynamic dependency graphs.

E-FRP [34] programs are reactive programs that are compiled into a very simple imperative language (called "SimpleC") that serves as an intermediate representation of C. E-FRP's design is similar to Emfrp's as signals in Emfrp are also always defined in the top-level (and has no support for dynamic modifications to the signal dependencies). However, E-FRP's programming style is completely different: signals in E-FRP are constructed, in essence, out of functions that simply return the new value of a signal given the old values of all signals and the event that caused the signals to update.

Synchronous Programing Languages. We cannot conclude this section without mentioning synchronous programming languages. Synchronous languages are closely related to reactive languages as they are also used for building applications that continuously react to external events. Many synchronous programming language support the ability to compile to imperative code (or another in-between representation) similar to the languages mentioned above: for example, Esterel [3] and Lucid Synchrone [5, 25], The main difference between the RP languages and synchronous languages is that the latter are usually designed with responsiveness and reliability in mind (i.e. synchronous languages are commonly used for building real-time applications where safety and reliability is vital [2]), while reactive programming has originally been invented for preventing callback hell [18] in event-driven applications and, usually, do not

need the same kind of constraints and guarantees compared that synchronous languages provide [14].

One noteworthy mention is the language Céu-T, which is derived from the synchronous language Céu [27], for which a specialised virtual machine exist called VM-T [4]. VM-T, however, is much more low-level compared to Remus. Programs for VM-T are not simply an ordering of how various components (i.e. signals) are updated like in Remus, but are imperative programs that has to assign values to registers, push values to a stack (to store argument values when calling a function, or during the evaluation of complex arithmatic expressions), and use instructions to copy or clear certain data structures. VM-T's main focus is on the coordination of events that cause the evaluation of the various components that make up a syncronous program, which is different from Remus where we simply assume the existence of certain (globally-defined) signals that, from the perspective of the RP programmer, always have a value.

6 Conclusion

In this paper, we presented Remus: a formal specification which outlines the design of a virtual machine, specialised in running reactive programs via pre-scheduled reactors. Remus design makes a few simplifications with respect to other virtual machine designs. First, the memory layout of a Remus program consist, mostly, of deployments. The size of a deployment depends solely on the size of the reactor (i.e. the number of sources, sinks and commands) of which it is an instance of. Commands in a Remus program will, with some few exceptions, always store the value that they produce in a memory region belonging to a deployment, offset with the location of the command itself.

In this paper, we have outlined the functionality of Remus that supports higher-order reactive programs (i.e. reactive programs where the dependency graph is dynamic). More specifically, we explained that even in the presence of such functionality, Remus programs can still be pre-scheduled. This is a consequence of Haai's design (Haai is the reactive programming language that Remus is based on). Haai programs do not require one to write a non-reactive loader program that constructs the reactive program's dependency graph. By treating the whole reactive program as the composition of reactors, one can use the (simple) evaluation rules of reactors, without having to reason about all the complex means in which the reactive semantics are integrated with the non-reactive semantics.

In the future, we aim to use the design of Remus to implement an actual (efficient) virtual machine that can serve as a run-time for embedded reactive applications, similar to how (e.g.,) Warduino [29], MicroPython [19] and Duktape [13] are targeting microcontrollers for (respectively) Webassembly, Python and Javascript.

Acknowledgments

Bjarno Oeyen is funded by the Research Foundation - Flanders (FWO) under grant number 1S93822N.

References

[1] Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. 2021. Diamonds are not forever: liveness in reactive programming with guarded recursion. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434283

[2] Albert Benveniste. 1998. Synchronous Languages and Reactive System Design. *IFAC Proceedings Volumes* 31, 15 (1998), 35–46. https://doi.org/10.1016/S1474-6670(17)40526-X 9th IFAC Symposium on Information Control in Manufacturing 1998 (INCOM '98), Nancy, France, 24-26 June.

[3] Gérard Berry and Georges Gonthier. 1992. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Sci. Comput. Program.* 19, 2 (1992), 87–152. https://doi.org/10.1016/0167-6423(92)90005-V

[4] Adriano Branco, Noemi Rodriguez, and Silvana Rossetto. 2021. A Virtual Machine for Reactive Programming on IoT devices. (2021). https://web.archive.org/web/20220908133001/http://bib-di.inf.puc-rio.br/ftp/pub/docs/techreports/21_02_branco.pdf

[5] Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. 2006. Mixing signals and modes in synchronous data-flow systems. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software, EMSOFT 2006, October 22-25, 2006, Seoul, Korea*, Sang Lyul Min and Wang Yi (Eds.). ACM, 73–82. https://doi.org/10.1145/1176887.1176899

[6] Gregory H. Cooper. 2008. *Integrating Dataflow Evaluation into a Practical Higher-Order Call-by-Value Language.* Ph. D. Dissertation. Brown University, USA. https://cs.brown.edu/research/pubs/theses/phd/2008/cooper.pdf

[7] Gregory H. Cooper and Shriram Krishnamurthi. 2006. Embedding Dynamic Dataflow in a Call-by-Value Language. In *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3924)*, Peter Sestoft (Ed.). Springer, 294–308. https://doi.org/10.1007/11693024_20

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition.* MIT Press. http://mitpress.mit.edu/books/introduction-algorithms

[9] Antony Courtney, Henrik Nilsson, and John Peterson. 2003. The Yampa arcade. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2003, Uppsala, Sweden, August 28, 2003*, Johan Jeuring (Ed.). ACM, 7–18. https://doi.org/10.1145/871895.871897

[10] Evan Czaplicki and Stephen Chong. 2013. Asynchronous functional reactive programming for GUIs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 411–422. https://doi.org/10.1145/2491956.2462161

[11] L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient Implementation of the Smalltalk-80 System. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, Salt Lake City, Utah, USA, January 1984*, Ken Kennedy, Mary S. Van Deusen, and Larry Landweber (Eds.). ACM Press, 297–302. https://doi.org/10.1145/800017.800542

[12] Joscha Drechsler, Ragnar Mogk, Guido Salvaneschi, and Mira Mezini. 2018. Thread-safe reactive programming. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 107:1–107:30. https://doi.org/10.1145/3276477

[13] Duktape Contributors. 2022. Duktape. https://web.archive.org/web/20220822064245/https://duktape.org/.

[14] Alan Jeffrey. 2013. Functional reactive programming with liveness guarantees. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg

Morrisett and Tarmo Uustalu (Eds.). ACM, 233–244. https://doi.org/10.1145/2500365.2500584

[15] Richard Kelsey, William D. Clinger, and Jonathan Rees. 1998. Revised^5 Report on the Algorithmic Language Scheme. *ACM SIGPLAN Notices* 33, 9 (1998), 26–76. https://doi.org/10.1145/290229.290234

[16] Robert Bruce Findler Matthias Felleisen and Matthew Flatt. 2009. *Semantics engineering with PLT Redex*. MIT Press, Cambridge, Mass.

[17] Erik Meijer. 2010. Reactive Extensions (Rx): Curing Your Asynchronous Programming Blues. In *ACM SIGPLAN Commercial Users of Functional Programming* (Baltimore, Maryland) *(CUFP '10)*. Association for Computing Machinery, New York, NY, USA, Article 11, 1 pages. https://doi.org/10.1145/1900160.1900173

[18] Leo A. Meyerovich, Arjun Guha, Jacob P. Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. Flapjax: a programming language for Ajax applications. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, Shail Arora and Gary T. Leavens (Eds.). ACM, 1–20. https://doi.org/10.1145/1640089.1640091

[19] MicroPython Contributors. 2022. GitHub - micropython/micropython: MicroPython - a lean and efficient Python implementation for microcontrollers and constrained systems. https://web.archive.org/web/20220912080849/https://github.com/micropython/micropython/.

[20] Ragnar Mogk, Joscha Drechsler, Guido Salvaneschi, and Mira Mezini. 2019. A fault-tolerant programming model for distributed interactive applications. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 144:1–144:29. https://doi.org/10.1145/3360570

[21] Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. 2019. Distributed Reactive Programming for Reactive Distributed Systems. *Art Sci. Eng. Program.* 3, 3 (2019), 5. https://doi.org/10.22152/programming-journal.org/2019/3/5

[22] Bjarno Oeyen, Sam Van den Vonder, and Wolfgang De Meuter. 2020. Reactive sorting networks. In *REBLS 2020: Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, Virtual Event, USA, November 16, 2020*. ACM, 38–50. https://doi.org/10.1145/3427763.3428316

[23] Bjarno Oeyen, Sam Van den Vonder, and Wolfgang De Meuter. 2021. Trampoline variables: a general method for state accumulation in reactive programming. In *REBLS 2021: Proceedings of the 8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, Chicago, IL, USA, 18 October 2021*, Louis Mandel (Ed.). ACM, 27–40. https://doi.org/10.1145/3486605.3486787

[24] Bjarno Oeyen, Humberto Rodríguez-Avila, Sam Van den Vonder, and Wolfgang De Meuter. 2018. Composable higher-order reactors as the basis for a live reactive programming environment. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, REBLS@SPLASH 2018, Boston, MA, USA, November 4, 2018*, Guido Salvaneschi, Wolfgang De Meuter, Patrick Eugster, Lukasz Ziarek, and Francisco Sant'Anna (Eds.). ACM, 51–60. https://doi.org/10.1145/3281278.3281284

[25] Marc Pouzet. 2010. index. https://web.archive.org/web/20220909125429/https://www.di.ens.fr/~pouzet/lucid-synchrone/index.html.

[26] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. 2014. REScala: bridging between object-oriented and functional style in reactive applications. In *13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland, April 22-26, 2014*, Walter Binder, Erik Ernst, Achille Peternier, and Robert Hirschfeld (Eds.). ACM, 25–36. https://doi.org/10.1145/2577080.2577083

[27] Francisco Sant'Anna, Roberto Ierusalimschy, Noemi de La Rocque Rodriguez, Silvana Rossetto, and Adriano Branco. 2017. The Design and Implementation of the Synchronous Language CÉU. *ACM Trans. Embed. Comput. Syst.* 16, 4 (2017), 98:1–98:26. https://doi.org/10.1145/3035544

[28] Kensuke Sawada and Takuo Watanabe. 2016. Emfrp: a functional reactive programming language for small-scale embedded systems. In *Companion Proceedings of the 15th International Conference on Modularity, Málaga, Spain, March 14 - 18, 2016*, Lidia Fuentes, Don S. Batory, and Krzysztof Czarnecki (Eds.). ACM, 36–44. https://doi.org/10.1145/2892664.2892670

[29] Robbert Gurdeep Singh and Christophe Scholliers. 2019. WARDuino: a dynamic WebAssembly virtual machine for programming microcontrollers. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, MPLR 2019, Athens, Greece, October 21-22, 2019*, Antony L. Hosking and Irene Finocchi (Eds.). ACM, 27–36. https://doi.org/10.1145/3357390.3361029

[30] Artur Sterz, Matthias Eichholz, Ragnar Mogk, Lars Baumgärtner, Pablo Graubner, Matthias Hollick, Mira Mezini, and Bernd Freisleben. 2021. ReactiFi: Reactive Programming of Wi-Fi Firmware on Mobile Devices. *Art Sci. Eng. Program.* 5, 2 (2021), 4. https://doi.org/10.22152/programming-journal.org/2021/5/4

[31] Kohei Suzuki, Kanato Nagayama, Kensuke Sawada, and Takuo Watanabe. 2018. CFRP: A functional reactive programming language for small-scale embedded systems. In *Theory and Practice of Computation: Proceedings of Workshop on Computation: Theory and Practice WCTP2016*. World Scientific, 1–13.

[32] TypeSafe, Inc. 2022. Documentation | Akka. https://web.archive.org/web/20220908081951/https://akka.io/docs/.

[33] Zhanyong Wan, Walid Taha, and Paul Hudak. 2001. Real-Time FRP. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001*, Benjamin C. Pierce (Ed.). ACM, 146–156. https://doi.org/10.1145/507635.507654

[34] Zhanyong Wan, Walid Taha, and Paul Hudak. 2002. Event-Driven FRP. In *Practical Aspects of Declarative Languages, 4th International Symposium, PADL 2002, Portland, OR, USA, January 19-20, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2257)*, Shriram Krishnamurthi and C. R. Ramakrishnan (Eds.). Springer, 155–172. https://doi.org/10.1007/3-540-45587-6_11

[35] Sheng Wang and Takuo Watanabe. 2020. *Functional Reactive EDSL with Asynchronous Execution for Resource-Constrained Embedded Systems*. Springer International Publishing, Cham, 171–190. https://doi.org/10.1007/978-3-030-26428-4_12