Dataflow Programming with RVC-CAL

Endri Bezati

July 17, 2019

Chapter 1

Dataflow Programming

1.1 Model of Computations

The emergence of parallel processing elements such as many-cores/multi-cores, FPGAs, GPGPUs demands to rethink the way of programming them. It is widely recognized that programming parallel platforms is difficult and tedious. In addition, heterogeneous platforms consisting on parallel processing elements is becoming a standard on personal computers, a combination of multi-core processors and massively parallel GPUs, and also the introduction of MPSoCs with programmable logic in the industry demands higher level of abstraction. A key to the heterogeneous system level design is the notion of models of computation (MoC) [13]. A MoC is the semantics of the interactions between modules. Moreover, it is the model or the specification principles of a design. Furthermore, MoCs relate strongly to the design style but is not necessary to refer to the implementation technology. Classes of MoCs include: Imperative, Finite State Machine, Discrete Event, Synchronous Languages, and Dataflow.

The imperative MoC executes the modules sequentially to accomplish a task. In Finite State Machines MoC, an enumeration of set of states specifies the steps to achieve a task. In the Discrete Event MoC, modules react to event that occurs at a given time instant and produces other events at the same time instant or at some future time instant. In Synchronous Languages MoC, modules simultaneously react to a set of input events and instantaneously produce output events.

A Dataflow MoC, is conceptually represented as a direct graph where nodes, called actors, represent computational units, while edges describes communication channels on which tokens are flowing. A token is an atomic piece of data. Dataflow graphs are often used to represent data-dominated systems, like signal processing applications. Using Dataflow MoC in such application domains often leads to behavioral descriptions that are much closer to the original conception of the algorithms than if an imperative MoC was used. Dataflow models also date back to the early 1970s, starting with seminal work by Dennis [6] and Kahn [11].

Several execution models that define the behavior of a dataflow program have been introduced in literature [11, 13]. A Dataflow MoC may constrain the behavior of an actor, how actors are executed relatively to each other, and aspects of their interaction with one another. As a result, different MoCs offer different degrees of analyzability and compile-time schedulability of dataflow programs written in them, and permit different guarantees (such as absence of deadlocks or boundedness of buffers) to be inferred from them.

The first two MoC are fundamentally sequential and the last three are concurrent. In fact it is possible to use the first two on parallel processing elements and the last three on sequential machines. Thus, there is a distinction between MoCs and the way that they are implemented. As a consequence, the efficiency might take a hit. In heterogeneous platforms there should be a separation of tasks depending properties of a design. Modules that are sequential should preferably execute on sequential platforms and parallel ones should perform on concurrent machines. In effect, for system level design either should be a mix of different MoCs or the properties and semantics of a single MoC should be rich enough to support heterogeneous designs.

A potential candidate for heterogeneous system level design is RVC-CAL. RVC-CAL is dataflow programming language that is based on the Dataflow MoC and it has the property to express applications as network processes. In fact, it offers parallelism scalability, modularity, scheduling by finite state machines, portability, and adaptivity properties that are necessary to unify the system level design for heterogeneous platforms. The MoC underlying the dataflow networks that are expressed using the CAL formal language is based on the *dataflow process networks* (DPN) model [13]. In addition to the properties of dataflow mentioned above, each DPN actor executes a sequence of discrete computational steps, called *firings*. In each step, an actor may (a) consume a finite number of input tokens, (b) produce a finite number of output tokens, and (c) modify its internal state, if it has any. In an actor language such as CAL [7] and its subset RVC-CAL this behavior is specified as one or more *actions*. Each action describes the conditions under which it may be fired (which may include the availability and values of input tokens, and the actor's state), and also what happens when it fires, i.e. how many tokens are consumed and produced at each port, the values of the output tokens, and how the actor state is modified. The execution of such an actor consists of two alternating phases: the determination of an action who firing conditions are fulfilled (including a choice if there is more than at some point), and the execution of that action itself.

Table 1.1 compares traditional languages against a set of language requirements. Partial values are retrieved from [9]. RVC-CAL supports the following behavioral hierarchies: sequential execution inside actions, FSM by the finite state machine of the actor, concurrent and pipelined execution by the MoC. In addition, RVC-CAL supports structural hierarchy by actor composition. An actor composition may contain another actor composition. Furthermore, Synchronization is provided by the FIFO queues that the actors are interconnected. Exception handling is not currently supported. RVC-CAL is high-level language that makes a total abstraction of time. Moreover, RVC-CAL support list types

Table 1.1: System-Level Requirements and Coverage. With ●supported, ◗partially supported, and ❍not supported.

	C	C++	OpenCL	Java	VHDL	Verilog	HardwareC	SpecC	RVC-CAL
Behavioral hierarchy	❍	❍	◗	❍	❍	❍	❍	●	●
Structural hierarchy	❍	❍	❍	❍	●	●	●	●	●
Concurrency	❍	◗	●	◗	●	●	●	●	●
Synchronization	❍	❍	●	◗	●	●	●	●	●
Exception handling	◗	●	●	●	●	●	❍	●	❍
Timing	❍	❍	❍	❍	●	●	◗	●	❍
State transitions	❍	❍	◗	❍	❍	❍	❍	●	●
Composition data types	●	●	●	●	◗	◗	❍	●	◗
Heterogeneous & CoDesign	❍	❍	●	❍	❍	❍	❍	❍	●
Fine-Grain Profiling	❍	❍	◗	❍	❍	❍	❍	❍	●

and future version will also support composite types. Finally, this thesis demonstrates that RVC-CAL is a potential candidate heterogeneous system level design and that it supports fine-grain profiling for hardware and software processing elements.

Before describing the CAL programming language and its features, a brief introduction to the Process Networks and the Model of Computations that CAL uses is given in the next section.

1.2 Process Networks

In this section, the Kahn Process Network and Dataflow Process Network MoCs are described. CAL inherits the properties of both models and extends them with the Actor Transition System. A Model of Computation or MoC is a formal representation of the operational semantics of a network of functional blocks describing a computation [12]. Moreover, it allows to specify the algorithm and the cost (i.e. time) of the operations.

1.2.1 KPN

The Kahn Process Networks [11] is a formal model of concurrent computation first introduced by French computer scientist Gilles Kahn in 1974. He introduced a language with simple semantics with the goal of applying mathematical approaches to programming languages and system design. Kahn's model can naturally describe signal processing systems in which infinite streams of data samples are incrementally transformed by a collection of processes executing either in sequence or parallel.

In Kahn's model, a network of processes communicates with each other via unbounded FIFO queues. Kahn describes its model as a set of Turing machines [18] connected via one-way tapes. Each process shares data with each other only through the input and output queues. Each queue may contain a possible infinite sequence of tokens. By using the same notation as in [13], each sequence or a stream is denoted with $X = [x_1, x_2, x_3, \ldots]$ where each x_i is a token drawn from a set. A token is an atomic data object that is *written* (produced) exactly once, and *read* (consumed) exactly once. *Writes* to the queues are *non-blocking*, in the sense that they always succeed immediately, but *reads* from queues are *blocking*, in the sense that if a process attempts to read a token from a queue and no data is available, then it stalls (i.e. wait) until the queue has sufficient tokens to satisfy the read. In other words, it is not possible to test the presence of input tokens.

Let S^p denotes the set of p-tuples of sequences as in $X = \{X_1, X_2, \ldots, X_p\} \in S^p$. A Kahn process is then defined as a mapping from a set of input sequences to a set of output sequences such as:

$$F : S^p \rightarrow S^q \tag{1.1}$$

The KPN process F has *event semantics* instead of state semantics such as continuous time which is used in some other domains. Furthermore, the only technical restriction is the need of F a continuous mapping function.

Considering a prefix ordering of sequences, the sequence X precedes the sequence Y (written $X \sqsubseteq Y$) if X is a prefix of (or is equal to) Y. For example, if $X = [x_1, x_2]$ and $Y = [x_1, x_2, x_3]$ then $X \sqsubseteq Y$. It is common to say that X approximates Y, since it provides partial information about Y. The empty sequence, denoted with $\perp$ is a prefix of any other sequence.

The increasing chain (possibly infinite) of sequences is defined as $\chi = \{X_0, X_1, \ldots\}$ where $X_1 \sqsubseteq X_2 \sqsubseteq \ldots$. Such an increasing chain of sequences has one or more upper bounds Y, where $X_i \sqsubseteq Y$ for all $X_i \in \chi$. The least upper bound (LUB) $\sqcup\chi$ is an upper bound such that for any other upper bound Y, $\sqcup\chi \sqsubseteq Y$. The LUB may be an infinite sequence.

Let consider a functional process F and an increasing chain of sets of sequences χ. As defined in the Equation (1.1), F will map χ into another set of sequences Ψ that may or may not be an increasing chain. Let $\sqcup\chi$ denote the LUB of the increasing chain χ. Then F is said to be **continuous** if for all such chains χ, $\sqcup F(\chi)$ exists and:

$$F(\sqcup\chi) = \sqcup F(\chi) \tag{1.2}$$

Networks of continuous processes have a more intuitive property called monotonicity. A process F is said to be **monotonic** if:

$$X \sqsubseteq Y \Rightarrow F(X) \sqsubseteq F(Y) \tag{1.3}$$

A continuous process is monotonic. However, a monotonic process might be not continue. A key consequence of these properties is that a process can be computed iteratively [16]. This means that given a prefix of the final input

sequences, it may be possible to compute part of the output sequences. In other words, a monotonic process is non-strict (its inputs need not be complete before it can begin computation). In addition, a continuous process will not wait forever before producing an output (i.e. it will not wait for completion of an infinite input sequence). Networks of monotonic processes are *determinate*. The KPN monotonicity prove is given in [13].

1.2.2 Dataflow Process Network

Dataflow Process Networks (DPN) [13] formally establish a particular case of KPN, where the computational blocks are called *actors*. As for the KPN process, actors can communicate only through unidirectional and unbounded queues that can carry possible infinite sequences of tokens. As for KPN, writes to queues are *non-blocking*. Contrarily, reading from queues is *blocking* because an actor can test the presence of input tokens. If there are not enough input tokens, then the read returns immediately and the actor needs not be suspended as it cannot read. This could introduce *non-determinism*, without requiring the actor to be nondeterminate.

DPN networks naturally extend the KPN embracing the notion of actor firing [6]. Actor firing can be defined as an indivisible (atomic) quantum of computation. The firings themselves can be described as functions, and the invocation of them is controlled by firing rules. Sequences of firings define a continuous Kahn process as the least fixed point of an appropriately constructed function and are therefore formally establishing DPN as a particular case of KPN [15].

An actor with m inputs and n output is defined as a pair $\{f, R\}$, where:

- $f : S^m \to S^n$ is a function called the *firing function*
- $R \subseteq S^m$ is a set of finite sequences called the *firing rules*
- $f(r)$ is finite for all $r \in R$
- no two distinct $r, r' \in R$ are joinable, in the sense that they do not have a LUB

The Kahn process F defined in Equation (1.1) based on the actor $\{f, R\}$ has to be interpreted as the least-fixed-point function of the functional $\phi : (S^m \to S^n) \to (S^n \to S^m)$ defined such as:

$$(\phi(F))(s) = \begin{cases} f(r) \bullet F(s') & \text{if there exist } s \in R \text{ such that } s = r \bullet s' \text{ and } s \sqsubseteq s' \\ \Lambda & \text{otherwise.} \end{cases} \tag{1.4}$$

where $\bullet$ represents the concatenation operator, Λ the tuple of empty sequences and $(S^m \to S^n)$ the set of function mapping S^m to S^n. It is possible to demonstrate that ϕ is both a continuous and monotonic function. The firing function f does not need to be continuous. In fact, it might not be even monotonic. It merely needs to be a function, and its value must be finite for each of the firing rules [15].

1.2.3 Actor Transition System and Composition

Actor transition systems (ATS)[10] describe actors in terms of labeled transition systems. The ATS extends the notion of actor with firings by introducing the concepts of internal state, atomic step and priority. In an ATS, a step makes a transition from one state to another. An actor maintains and updates its internal variables: those are not sequences of tokens, but simple internal values that can not be shared among actors. Moreover, the notion of priority allows actors to ascertain and react to the *absence* of tokens. On the other hand, however, it can also make them harder to analyze, and it may introduce unwanted non-determinism into a dataflow application.

Let Σ denote the non-empty actor state space, $\mathcal{U}$ the universe of tokens that can be exchanged between actors and S^n a finite and partially ordered sequence of n tokens over $\mathcal{U}$. A n-to-m **actor** is a labeled transition system $\langle \sigma_0, \tau, \succ \rangle$ where:

- $\sigma_0 \in \Sigma$ is the actor initial state
- $\tau \subset \Sigma \times S^n \times S^m \times \Sigma$ defines the transition relation
- $\succ \subset \tau \times \tau$ defines a strict partial order over τ

Any $(\sigma, s, s', \sigma') \in \tau$ is called a **transition**, where $\sigma \in \Sigma$ is its source state, $s \in S^n$ its input tuple, $\sigma' \in \Sigma$ its destination state and $s' \in S^m$ its output tuple. It must be noted that $\succ$ is a non-reflexive, anti-symmetric and transitive partial order relation on τ, also called **priority** relation.

An equivalent and more compact notation for the transition (σ, s, s', σ') is $\sigma \xrightarrow{s \to s'} \sigma'$.

Enabled transition and step of an actor

Intuitively, the priority relation determines that a transition cannot occur if some other transition is possible. This can be seen as the definition of a valid step of an actor, which is a transition such as two conditions are satisfied:

- the required input tokens must be present
- there must not be another transition that has priority

Given a n-to-m actor $\langle \sigma_0, \tau, \succ \rangle$, a state $\sigma \in \Sigma$ and an input tuple $v \in S^n$, a transition $\sigma \xrightarrow{s \to s'} \sigma'$ is **enabled** if and only if:

$$\begin{cases} v \sqsubseteq s \\ \nexists \sigma \xrightarrow{r \to r'} \sigma'' \in \tau \ : \ r \sqsubseteq v \wedge \sigma \xrightarrow{s \to s'} \sigma' \succ \sigma \xrightarrow{r \to r'} \sigma'' \end{cases} \tag{1.5}$$

Hence, a **step** from state σ with input v is then defined as any enabled transition $\sigma \xrightarrow{s \to s'} \sigma'$.

Actors composition

For any transition relation τ its set of *input ports* P_τ^I and its set of *output ports* P_τ^O are defined as the ports where at least one transition consumes input from or produce output to:

$$\begin{cases} P_\tau^I = \{p \in P \mid \exists \sigma \xrightarrow{s \to s'} \sigma' \in \tau \ : \ \sigma(p) \neq \perp\} \\ P_\tau^O = \{p \in P \mid \exists \sigma \xrightarrow{s \to s'} \sigma' \in \tau \ : \ \sigma'(p) \neq \perp\} \end{cases} \tag{1.6}$$

where P is the set of input and output ports names. It is assumed that an input port with name p and an output port of the same name are in no way related. In order to express complex functionality, actors are composed into a **dataflow network** as the one depicted in Figure 1.1. The structure of a network can be represented by a partial function from (input) ports to (output) ports, mapping each input port in its domain to the output port that connects to it. Note that this implies the absence of fan-in (as every input port is connected to at most one output port), and it permits unconnected (open) input (and output) ports.

1.3 CAL Actor Language

The Cal Actor Language (CAL) [8] is a domain-specific language that provides useful abstractions for dataflow programming with actors. The language directly captures the features of DPN [14] MoC by adding the notion of atomic action firings, also called *steps*.

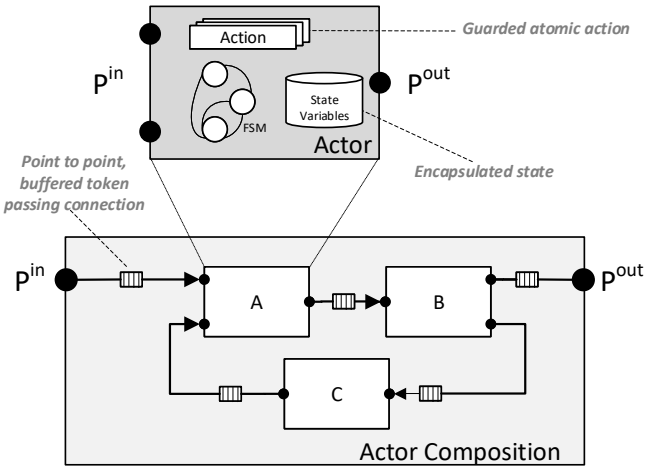

Figure 1.1: Actor Composition and Actor Structure.

Figure 1.1 illustrates the basic concepts of a CAL program. It represents a dataflow network composed by a set of **actors** and a set of first-in first-out (FIFO) **queues**. Each CAL actor is defined by a set of **input ports**, a set of

output ports, a set of **actions**, and a set of **internal variables**. CAL also includes the possibility of defining an explicit **finite state machine** (FSM). This FSM captures the actor state's behavior and drives the **action selection** according to its particular state to the presence of input tokens and to the value of the tokens evaluated by other language operators called **guard functions**. Each action may capture only a part of the firing rule of the actor together with the part of the firing function that pertains to the input/state combinations enabled by that partial rule defined by the FSM. An action is **enabled** according to its *input patterns* and *guards expressions*. While patterns are determined by the amount of data that is required for the input sequences, guards are boolean expressions on the current state and/or on input sequences that need to be satisfied for enabling the execution of an action.

In the following, a basic overview of the main concepts concerning the syntax and semantics of CAL language is presented.

1.3.1 CAL Program

A CAL program **network** (actor composition) N is defined as a tuple (K, A, B) where:

- $K = \{\kappa_1, \kappa_2, \ldots \kappa_{n_\kappa}\}$ is a finite set of actor classes
- $A = \{a_1, a_2, \ldots, a_{n_A}\}$ is a finite set of actors
- $B = \{b_1, b_2, \ldots, b_{n_B}\}$ is a finite set of queues

A CAL **actor class** κ defines the program-code-template and the implementation behaviors of the actor (i.e. the CAL source code). Different actors can instantiate the same class. However, each actor corresponds to a different *object* with its internal states that can not be shared.

A CAL **actor** a is defined as a tuple $(\kappa, P^{in}, P^{out}, \Lambda, \mathcal{V}, \text{FSM})$ where:

- κ is the actor class
- $P^{in} = \{p_1^{in}, p_2^{in}, \ldots, p_{n_I}^{in}\}$ is the finite set of input ports
- $P^{out} = \{p_1^{out}, p_2^{out}, \ldots, p_{n_O}^{out}\}$ is the finite set of output ports
- $\Lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_{n_\Lambda}\}$ is the finite set of actions
- $\mathcal{V} = \{v_1, v_2, \ldots, v_{n_V}\}$ is the finite set of internal variables
- FSM is the internal finite state machine

A CAL **queue** b is defined as a tuple (a_s, p_s, a_t, p_t) where:

- $a_s \in A$ is the source actor (i.e. the one that produces the tokens)
- $p_s \in P_{a_s}^{out}$ is the output port of the source actor
- $a_t \in A$ is the target actor (i.e. the one that consumes the tokens from the queue)
- $p_t \in P_{a_t}^{in}$ is the input port of the target actor

1.3.2 Execution Model

Here we assume that the firing of an action is performed following the serial execution of the four stages illustrated in Figure 1.2. Those stages are respectively:

- **Action selection** where the internal actor scheduler selects the next schedulable action (i.e. that satisfies all firing conditions). It should be noted that the action selection must wait until all the input tokens are available on the respective input queues (i.e. block-reading).
- **Read from input queues** where all the input tokens required by the algorithmic part of the action are read from the respective input queues.
- **Action execution** when the algorithmic part of the action is executed.
- **Write to output queues** when all the output tokens produced during the action execution are written to the respective output queues. It is to mention that when an actor is implemented either in software or hardware processing elements, the action can only start writing on the output queues when enough space for accommodating all output tokens is available.

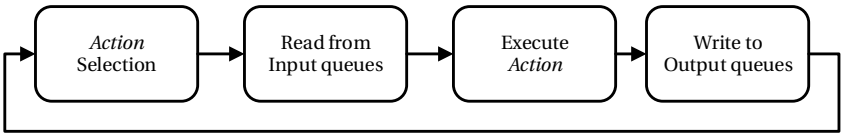

Figure 1.2: Actor Execution Model.

1.3.3 CAL Syntax and Semantics

Lexical tokens

Lexical tokens help the user to understand the functionality provided by any language. They are a string of indivisible characters known as lexemes. The CAL lexical tokens, also summarized in Table 1.2, are described in the following:

- **Keywords** Keywords are a special type of identifiers. They are already reserved by default in the programming language. These keywords can never be used as identifiers in the code. Some of them are `action`, `actor`, `priority`, `fsm`, `procedure`, `function`, `begin`, `if else`, `end`, `foreach`, `while`, `do`, `procedure`,`in`, `List`, `int`, `uint`, `float`, `bool`, `true` and `false`.
- **Operators** Operators usually represent mathematical, logical or algebraic operations. Operators are written as any string of characters `!`, `%`, `^`, `&`, `*`, `/`, `+`, `-`, `=`, `<`, `>`, `?`, `~` and `|`.

- **Delimiters** Delimiters are used to indicate the start or the end of this syntactical element in the CAL. Following elements are used as delimiters: `(`, `)`, `[`, `]`, { and }.
- **Comments** Comments in CAL language are the same as in Java, C and C++. Single line comments start with `//` and multiple line comments start with `/*` and end with `*/`.

Table 1.2: CAL lexical tokens

Kind	**Symbols**
Keywords	`action`, `actor`, `procedure`, `priority`, `fsm`, `function`, `begin`, `if`, `else`, `end`, `foreach`, `while`, `do`, `procedure`, `in`, `List`, `int`, `uint`, `float`, `bool`, `true`, `false`
Operators	`!`, `%`, `^`, `&`, `*`, `/`, `+`, `-`, `=`, `<`, `>`, `?`, `~`, `|`
Delimiters	`(`, `)`, `[`, `]`, {, }, `==>`, `-->`, `:=`
Comments	`//`, `/* ... */`

Actions

The simplest actor that can be described using CAL is the `Passthrough` actor defined in Listing 1.1. This actor copies a token from its input port and places it into its output port. The actor header is defined in the first line, which contains the actor name, followed by a list of parameters provided inside the `()` construct (empty, in this case), and the declaration of the input and output ports. The input ports are those in front of the ==> construct and the output ports are those after it. In this case the input and output ports set are defined as $P^{in}_{\texttt{Passthrough}} = \{\texttt{I}\}$ and $P^{out}_{\texttt{Passthrough}} = \{\texttt{O}\}$ respectively. For each parameter and port, the data type is specified before the name (all defined with an `int` data type, in this case). This actor contains only one *action*, labeled as `pass` as defined in second line. In this case, the actions set is defined as $\lambda_{\texttt{Passthrough}} = \{\texttt{pass}\}$. Action `pass` demonstrates how to specify token consumption and production. The part in front of the ==>, which defines the *input patterns*, it specifies how many tokens to consume from which ports and what to call those tokens in the rest of the action. In this case, there is one input pattern: `I:[v]`. This pattern indicates that one token is to be read (i.e. consumed) from the input port `I`, and that the token is to be called `v` in the rest of the action. Such an input pattern also defines a condition that must be satisfied for this action to fire: if the required token is not present, this action will not be executed. Therefore, input patterns do the following:

- They define the number of tokens (for each port) that will be consumed when the action is executed (fired).

- They declare the variable symbols by which tokens consumed by an action firing will be referred to within the action.
- They define a firing condition for the action, i.e. a condition that must be met for the action to be able to fire.

The *output patterns* of an action are those defined after the ==> construct. They simply define the number and values of the output tokens that will be produced on each output port by each firing of the action. In this case, the output pattern `O:[v]` says that exactly one token will be generated at output port `O`, and its value is `v`.

Listing 1.1: Passthrought.cal

```
package examples;
actor Passthrought() int I ==> int O :
    pass: action I:[v] ==> O:[v]
    end
end
```

Action *Guard*

So far, the only firing condition for actions was that there be enough tokens to consume, as specified in their input patterns. However, in many cases it is possible to specify additional criteria that need to be satisfied for an action to fire. Conditions, for instance, that depend on the values of the tokens, or the state of the actor, or both. These conditions can be specified using *guards*, as for example in the `Split` actor, defined in Listing 1.2. This actor defines one input port `I`, two output ports `O1` and `O2`, and two actions `A` and `B`. Those actions require the availability of one token in `I`. However their selection is guarded by the value of the input token `val` read from `I`. In this example, if `val >= 0` then the action `A` is selected, otherwise action `B`.

Listing 1.2: Split.cal

```
package examples;
actor Split() int I ==> int O1, int O2 :

    A: action I:[val] ==> O1:[val]
    guard
        val >= 0
    end
    B: action I:[val] ==> O2:[val]
    guard
        val <  0
    end

end
```

In the `PingPongMerge` actor, reported in Listing 1.3, a finite state machine schedule is used to sequence the two actions `A` and `B`. The schedule statement

introduces two states `s1` and `s2`. Contrarily, in the `BiasedMerge` actor, reported in Listing 1.4, the selection of which action to fire is not only determined by the availability of tokens, but also depends on the priority statement.

Listing 1.3: PingPongMerge.cal

```
package examples;
actor PingPongMerge() T In1, T In2 ==> T O :

    A: action In1:[val] ==> O:[val] end

    B: action In2:[val] ==> O:[val] end

    schedule fsm s1:
        s1(A) --> B;
        s2(B) --> A;
    end

end
```

Listing 1.4: BiasedMerge.cal

```
package examples;
actor BiasedMerge() T In1, T In2 ==> T O :

    A: action In1:[val] ==> O:[val] end

    B: action In2:[val] ==> O:[val] end

    priority
        A > B
    end

end
```

1.4 Standardization

A subset of the CAL programming language has been standardized by the ISO MPEG comity and is called RVC-CAL. The MPEG Reconfigurable Video Coding ISO/IEC 23001-4 has as a purpose to offer more flexible use and faster path to innovation of MPEG standards in a way that is competitive in the current dynamic and evolving environment. Thus, MPEG standards give an edge over its competitors by substantially reducing the time for which technology is developed and the time the standard is available for market applications. The RVC (Reconfigurable Video Coding) initiative is based on the concept of reusing commonalities among different MPEG standards and provide possible extensions by using appropriate higher level specification formalisms. Thus, the objective of the RVC standard is to describe current and future codecs in a way that makes such commonalities explicit, reducing the implementation burden by providing a specification that its starting point is closer to the final

implementation. To achieve this objective, RVC provides the specification of new codecs by composing existing components and possibly new coding tools described in modular form.

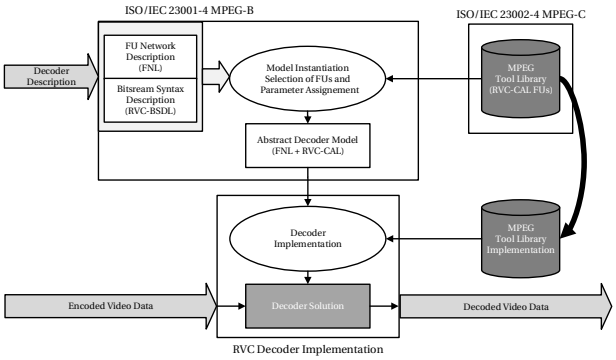

Figure 1.3: Reconfigurable Video Coding.

The MPEG-B standard defines the language that is used to build the MPEG RVC framework. The RVC-CAL dataflow programming language is the core of the system; it is used to describe the behavior description of each module called Functional Unit (FU). With the specification of the FU network topology, the functional behavior of a video decoder is specified. An FU network topology is also called abstract decoder module. The term abstract refers to the fact that FUs are only characterized by the I/O behavior and the firing rules embedded in the RVC-CAL language. Thus, the interaction of each FU other FUs is fully specified by abstracting the time and by only defining dependencies of data production and consumption. The MPEG-C standard defines the Video Tool Library, a library of video coding tools. Figure 1.3 illustrates the concept that any abstract decoder model, which is constituted by an FU network description, can be implemented either in hardware or software.

Chapter 2

RVC-CAL Design Flow

The architecture of the high-level synthesis design flow and associated design space exploration that offers a complete design flow methodology for programming heterogeneous platforms is illustrated in Figure 2.1. A design contains a set of stages by which, from an abstract representation of the application (i.e. the dataflow program), it is possible to accomplish the synthesis of an integrated circuit or the implementation onto a SW processing unit, or any combination of both elements. Each stage is composed by a single tool or the integration of a composition of tools.

The RVC-CAL design flow is composed by eight stages. The stages are respectively:

1. **Behavioral Description, Architecture, and Constraints**: The design is expressed by the RVC-CAL dataflow programming language based on process network principles. The architecture defines on which kind of platform the design is implemented. The architecture contains operators, media, and links. An operator defines the type of the processing element; the media defines the way that this platform is communicating, and the links are the connection between operators and media. In addition, the constraints are applied to the architecture and defines the clocking for each operator and the input and output data consumption and production of the design.

2. **Compiler Infrastructure**: The abstract design specification is verified for algorithmic correctness. The design can also be statically profiled for complexity analysis and for identifying the longest computational path occurring when a set of input vectors are processed.

3. **Code Generation**: According to the architecture defined at the abstract level, the code generation stage generated the source code for execution on the SW architecture and HDL code for configuring the HW architectures, FPGAs and/or ASICs.

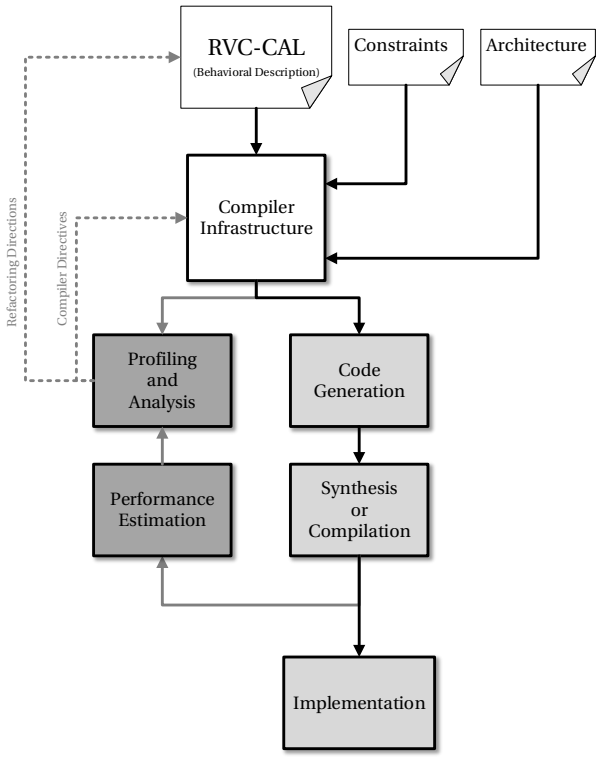

Figure 2.1: RVC-CAL Design Flow. Two directional flows, in *black* top to down implementation and in *grey* the iterative feedback.

4. **Synthesis or Compilation**: Generates code which is then synthesized or compiled using standard tools to obtain software executables and/or hardware binary files/netlists for physical implementations.

5. **Performance Estimation**: At this stage platform specific software profilers and/or HDL testbenches are used to measure the performance of individual dataflow processing components.

6. **Analysis & Profiling**: At this stage, the design bottlenecks are iteratively identified and analyzed. Initially, in the early phase of the design process, the buffer size for the different dataflow elements, the memory defined in the architecture given by the constraints are estimated and allocated. In a second phase, a more in-depth design space exploration including all dataflow components is applied to the design and profiling information is extracted. For a software architecture, the design can be partitioned into different processing units according to an optimization objective functions for the given set of an input vector and design constraints.

For hardware architectures, different multi-clock domain partitioning are identified with the goal for reducing power dissipation and respecting throughputs constraints. Finally, the performance of the composition of hardware and software architectures can be analyzed with the purpose of verifying the satisfaction of the overall system design constraints.

7. **Code Refactoring Directions and Compiler Directives**: After the Profiling & Analysis stage, feedback is provided on how the dataflow program components, at a high abstraction layer, should be modified to satisfy the design constraints.

8. **Implementation**: The structure of the design flow is composed by two main paths. The first is a direct path from the top to bottom linking the high-level dataflow program abstraction to the synthesized executable implementation and the second is the iterative system-level design exploration and optimization cycle.

The structure of the design flow is composed by two main paths. The first is a direct path from the top to bottom linking the high level dataflow program abstraction to the synthesized executable implementation which is described in this report, and the second is the iterative system-level design exploration and optimization cycle which the interested reader can have a look at [5].

2.1 RVC-CAL Compiler Infrastructure

A compiler supporting compilation of programs written using the standard RVC-CAL language is called Orcc [2]. Orcc stands for Open RVC-CAL compiler, and it is collaboration work between INSA of Rennes and EPFL. Recalling the RVC-CAL Design Flow in Figure 2.1. Orcc provides the necessary tools for designing, simulating and generating source code for different targets.

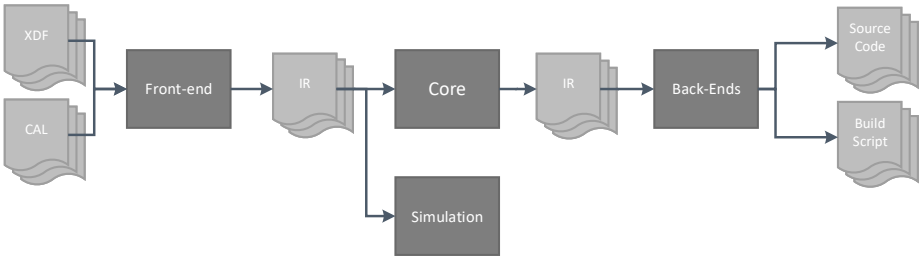

Figure 2.2: Open RVC-CAL Compiler Infrastructure.

Figure 2.2 represents the Orcc building blocks of the Orcc compiler Infrastructure. The compilation flow primarily translated the RVC-CAL into source code or into intermediate representation (e.g. such as the LLVM (Low Level Virtual Machine)), instead of generating machine code like traditional compilers (e.g. GCC, Clang) do.

Orcc uses extensively Model-Driven Engineering (MDE) by representing the IR with meta-models. Also, it uses the same MDE technologies that are employed in Eclipse IDE. Those MDEs are the Eclipse Modeling Framework (EMF), Xtext, and Xtend. The use of meta-models and MDE speeds up the development by automating time-consuming and most important error-prone tasks. Meta-modeling offers maintainability of the source code by having a global homogeneity of a unique model for different meta-tools (in Orcc's case Xtext and Xtend uses the same EMF meta-model for the Orcc's IR). Also, a meta-model is a source of documenting the code equivalent to UML.

The Orcc's compiler infrastructure is the following:

- **Front-end**: RVC-CAL is parsed and translated into an Abstract Syntax Tree. The parsing is implemented using Xtext [4], a framework dedicated to the development of DSL(Domain-Specific Language) that automatically generates a parser, a linker and an editor from the behavioral description grammar. The AST (Abstract Syntax Tree) then is transformed into an Intermediate Representation. During this step the front-end, by extending classes in Xtext framework, performs semantic validation, type inference, and expression evaluation.

- **Core**: Defines the IR and the visitors for optimizing it. The IR is modeled using the Eclipse Modeling Framework [1, 17]. This framework offers many methods for manipulating the data structure, one of them is the containment relationship between objects. Furthermore, it provides the automatic serialization of the meta-model (IR), allowing incremental compilation.

- **Simulation**: Orcc offers a simulation of an RVC-CAL program by interpreting its IR. The simulation is type accurate, and it permits verifying the correct functionality of the RVC-CAL program before implementation.

- **Back-end**: Is the final block in the compiler flow. It applies target particular optimization (IR to IR transformations) before the code is generated. Orcc's back-ends translate the RVC-CAL program into a general purpose programming language to benefit from the optimizations that those compilers offer. Whenever these optimizations are not enough, additional IR optimization passes are performed by Orcc's back-ends to meet the demands. To generate the code for a particular target, the Xtend [3] framework is used. This framework provides a template based code generation that is flexible and easy to use. It is meta-language based on Java and is fully integrated into Eclipse IDE.

2.2 Intermediate representation

Orcc dataflow IR is the implementation of the DPN MoC into a graph model and its representation in Classes is given in Figure 2.3. According to this model actors and a networks of actors are vertices that contains Input/Output (IO) ports.

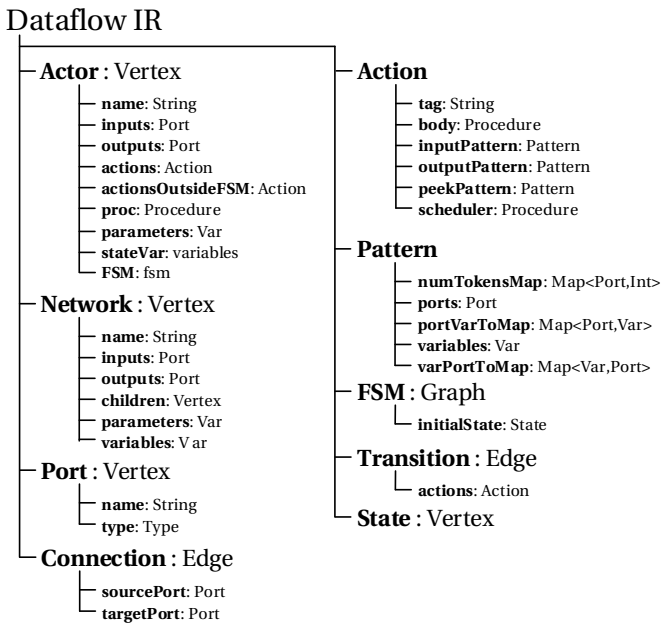

Figure 2.3: The Classes of the Dataflow IR Model

Each vertex is connected through edges that represents the queues. Each actor contains IO port, state variables, actions and, if any, the finite state machine (FSM) of the action execution which is also represented as a graph with nodes representing states and the transition as the actions. Actions, are procedures and guards of all actions and their firing rules are integrated into a procedure called *action selection*, which is fully described below. The semantic information of procedures is represented with a low-level instruction based language which includes: functional calls, conditionals, assignments statement for local variables and load/stores for state variables.

2.3 General execution model followed by backends

In Section 1.3.2, it was described how actions are being selected for execution. Here we are going to discuss how backends are handling the action execution inside actors and the execution of actors inside a dataflow network.

Figure 2.4 represents an internal state machine of actor with two actions. Those actions, have a label X and Y. Depending on the guard conditions and the finite state machine of an actor, the backends will generate a state machine for representing its execution. In general, the action selection or the actor scheduler

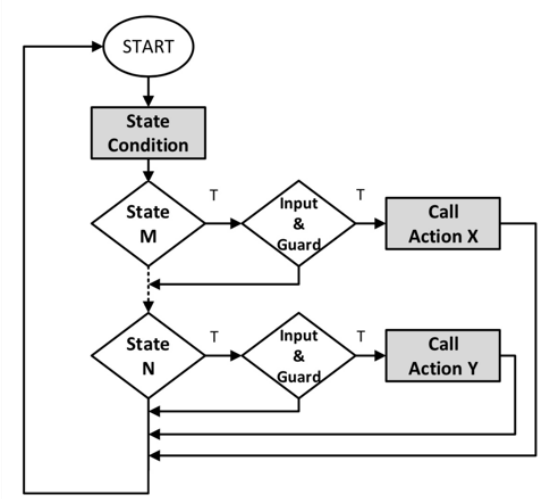

Figure 2.4: Flowgraph of action selection inside actors.

(as named by the ORCC intermediate representation) is the top function of an actor and is responsible for selecting an action depending on the state of the actor, the availability of input tokens for an action and its guard expression. Most of the ORCC backends represents this state machine as a *switch-case* statement. Moreover, Figure 2.4 depicts also that only one action can be executed at a time as defined by the MoC of the DPN.

The majority of dataflow MoC expects that every actor runs in parallel. Due to the fact that most of the dataflow applications might contain more actors than available processing cores, an actor scheduling should be implemented for executing correctly the dataflow on a given machine with few processing cores. In ORCC backend this achieved by the round-robin scheduling. As an example, the pseudo-code of the round-robin scheduling of the dataflow network represented in Figure 1.1 is depicted in Listing 2.1.

Listing 2.1: C Pseudo-Code for round robin-scheduling of the dataflow network represented in Figure 1.1

```
Actor A, B, C;

void main(){
    while(1){
        if(A.enable()) A.schedule();
        if(B.enable()) B.schedule();
        if(C.enable()) C.schedule();
    }
}
```

Moreover, the round-robin scheduling in ORCC is also non-premptive, which means that an actor is going to be executed as long as there are available input tokens and the guard expression of a selected action is true. For the Listing 2.1,

actor A is going to be executed up until all tokens from the two inputs are consumed and its guard conditions are true. After that, actor B and then C is executed in the same way as Actor A. Once all actors can not execute anymore, the network execution will terminate.

Chapter 3

Coding recommendations

3.1 Buffer Size

CAL inherits its dataflow model of execution from DPN. One of features of DPN and also Kahn Process Networks is that buffers should have an infinite size. For implementation reasons the buffer in Orcc are bounded. These have two implications on the generated code. First the actor scheduler should control if it has space in its output before execution, and second if the bounded buffer size is not large enough it may lead the dataflow program to a deadlock.

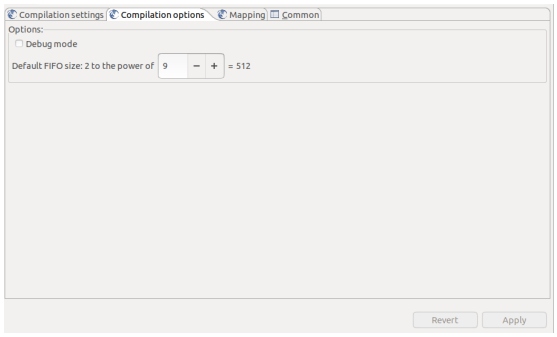

Figure 3.1: Buffer size selection in Orcc compilation run configuration.

By default, in Orcc all buffers has a fixed size of 512 tokens. This can be easily changed before or after generating the source code by backend in the run configuration tab in Eclipse (see Figure 3.1). The buffer size in Orcc are always values of power of two. Having power of two buffer size permits to replace a modulo operator by a shift operator each time a value in the FIFO is read or write. For the Java and C backends non power of two FIFO buffers will result in unexpected execution of the dataflow network and it may crash the program, thus Orcc does not permit you to put non power of two buffer sizes.

If the user wants to have a specific buffer size for a given FIFO in the dataflow network it can be easily achieved in the network editor using the Properties view of Eclipse. By default, the Properties view is not activated, the user can activated it by selecting Window→Show View→Other...→General→Properties.

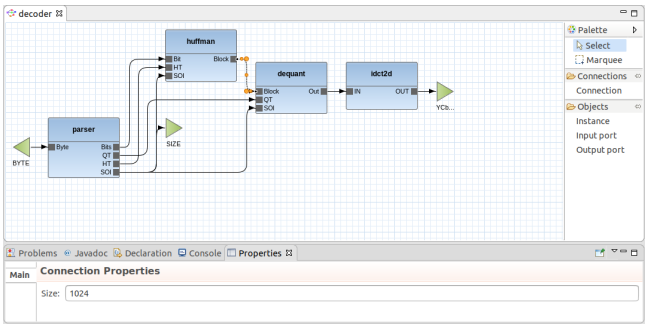

Figure 3.2: Changing the buffer size for a given FIFO.

Figure 3.2 depicts how to change the buffer size for a given FIFO of dataflow network. By selecting a FIFO (highlighted in Orange in the graph) the properties window permits the user to change the buffer size to the desired value. If a value is not a power of two the Size text-box will become red, notifying the user that this value is not permitted.

3.2 Most common deadlock issue

As we described in the previous section its possible to have a deadlock on the dataflow network execution if the buffers size is not set appropriately. Nevertheless, with the default buffer size most dataflow design should not deadlock. For actors that contains *repeats* on input and output patterns, Orcc will show a warning if the buffers size is less than the repeat expression.

The most common error in writing actors is when actors contains actions that reads from the same port. The designer should always cover all cases for consuming tokens with guard expressions in multiple actions. An example of buffer explosion is given in the next source code snippet.

Listing 3.1: An actor that might cause a deadlock

```
package examples;
actor PossibleDeadlock() int IN==> int OUT:

    action IN:[token] ==> OUT:[token]
    guard
        token < 5
    end

end
```

The CAL actor in Listing 3.1 reads integer tokens that are inferior of the value **5** If a source actor is connected to the PossibleDeadlock actor and produces token values that are inferior of 5 the network will not block. But if the $token \in Z$, the PossibleDeadlock actor will block its execution for values greater or equal to 5. Thus the buffer connected to port IN, at one moment will become full and the overall execution of the dataflow program will deadlock.

To solve the deadlock problem we need to consume the tokens that are equal or greater than 5. To do that, we need to add another action that consumes those tokens as depicted in Listing 3.2.

Listing 3.2: An actor that might cause a deadlock

```
package examples;
actor NoDeadlock() int IN==> int OUT:

    action IN:[token] ==> OUT:[token]
    guard
        token < 5
    end

    action IN:[token] ==>
    guard
        token >= 5
    end

end
```

3.3 Almost non-determinism in Orcc

The CAL programming language permits to describe non-determinate actors. The most common example is the Merge actor as depicted in Listing 3.3.

Listing 3.3: A merge actor

```
package examples;
actor Merge() int A, int B ==> int O:

    Act_A : action A:[token] ==> O:[token]
    end

    Act_B : action B:[token] ==> O:[token]
    end

end
```

The CAL code in Listing 3.3 has two inputs A and B and an output O. Depending on the token availability in a given time on port A or B, action Act_A or action Act_B is going to be executed. This actor comportment is called a non-determinant actor.

In Orcc, the implementation of such an actor is not *entirely* non-determinated. In the actors internal scheduler a choice to test an action before another one

is already predefined. In such a case, Orcc first will check if Act_A can be executed and then it will test Act_B. The choice in this case is not by the order of appearance of actions but on the string label of the actions, which in this case Act_A > Act_B. Here the developer should pay a lot of attention with such kind of actors, because the output of such an actor might depend also on the arrival of tokens on port A and B. Given the execution of Act_A and Act_B this actor might also add time dependency problems on the rest of the dataflow network.

It is recommended to not write such actors if we need determinate actors, non-determinate actors should always be the last choice for a developer. On the next two sections we will discuss how to implement determinate actors by using priorities and finite state machines.

3.4 Priorities in actors

CAL provides a construct for giving different priorities to actions. It might happen that we need to a give a higher priority to action so that it can be checked for scheduling before another action. For example, given the previous example of Listing 3.3, we want to give a higher priority to Act_B to be checked before Act_A for execution we do the following modifications as depicted in Listing 3.4.

Listing 3.4: A merge actor with priority

```
package examples;
actor MergeWithPriority() int A, int B ==> int O:

    Act_A : action A:[token] ==> O:[token]
    end

    Act_B : action B:[token] ==> O:[token]
    end

    priority
        Act_B > Act_A;
    end

end
```

As we can see in Listing 3.4, the *priority* construct has been included in the code. These constructs accepts only greater than expressions with labels of actions. The priority order is given by the order of appearance from left to right. If the MergeWithPriority actor had another action Act_C, and we want it to be the action with the lesser priority, we can write the following $Act\_B > Act\_A > Act\_C$. Also, we can add lines with other priorities if we have more actions in out actor. Finally, the priority construct can be used together with the finite state machine that we are going to describe in the next section.

3.5 Finite state machine in actors

In CAL we can order the execution of different actions by a using a finite state machine (FSM). The FSM in actors permit the developer to add multi functionality on an actor and also sequential behavior. The CAL keyword is *schedule fsm* and it is generally positioned at the end of actors source file.

Listing 3.5: An actor with an FSM

```
package examples;
actor ActorFSM() int A, int B ==> int O:

    Read_5 : action A:[token] repeat 5 ==> O:[token] repeat 5
    end

    Read_10 : action B:[token] repeat 10 ==> O:[token] repeat 10
    end

    schedule fsm s0:
        s0  (Read_5) --> s1;
        s1  (Read_10) --> s0;
    end

end
```

An example of actor containing a FSM is depicted in Listing 3.5. The actor contains two actions that read from port A five tokens and ten tokens from port B accordingly, and outputs it to the output port O. The FSM of the actor has two states *s0* and *s1*. The initial state of the FSM is s0. In addition the FSM has two transitions Read_5 and Read_10 which are the labels of the actions of the actor. The lines inside the *schedule fsm* block, can be interpreted as following. From a source state S_{source} execute transition T_{label} and go to a state S_{target}. The execution pattern of the actor is the following Read_5 then Read_10 and then back Read_5 as long as we have data on port A and B.

A FSM can be combined with a priority if we have a state with many transitions. The Listing 3.6 shows how to combine those two principles. Assuming that we need a read tokens one by one by from the input port and then again we need to write one by one tokens to the output port we can do the following in CAL. We need three action: read, write and done. The read action will consume a token every time it executes and store it to an array, the write action will produce a token to the output port. Both actions are going to be executed up until the Counter is equal to 64. Finally, the done action will stop either the reading or writing once the counter is equal to 64. As it can be observed on the listing state s0 and s1 has two transitions. The priority conditions forces the actor scheduler to check for execution the done action before checking the read action, thus we ensure that the read action will not be executed more than 64 times. The same happens on the state s1. It is possible to modify the code and not have priorities conditions by adding a guard expression on both read and write actions as follows $Counter < 64$. But this will imply that we are going to add a check in the generated actor scheduler that is not necessary. Because

Listing 3.6: An actor with an FSM and priority

```
package examples;
actor ActorFsmWithPriority() int A ==> int O:

    int Counter := 0;

    int array[64];

    read: action A:[token] ==>
    do
        array[Counter] := token;
        Counter := Counter + 1;
    end

    done: action ==>
    guard
        Counter = 64
    do
        Counter := 0;
    end

    write: action ==> O:[token]
    var
        int token = array[Counter]
    do
        Counter := Counter + 1;
    end

    schedule fsm s0:
        // -- Read 64 tokens
        s0 (read)  --> s0;
        s0 (done) --> s1;

        // -- Write 64 tokens
        s1 (write) --> s1;
        s1 (done) --> s0;
    end

    priority
        done > read;
        done > write;
    end

end
```

guard expression are expressed as functions in the generated code.

One particular aspect of the FSM supported by Orcc is that all states defined in the *schedule fsm* should exist, it is not possible to have a dangling state. This limitation in Orcc, does not permit the developer to have a dead state (a state that will stop the execution of an actor), thus it induces the developer to create an action with a guard that will block the actors execution.

3.6 Initialize action

An *initialize action* is an action that will run when the actor is being initialize and it runs only once. It is like a normal action so that it can read and write to I/O ports, modify the state of the actor and have guard conditions (generally in actor parameters, otherwise it does not make any sense to have a guard).

For the purpose of this project an initialize action should be used to initialize external system libraries. Another particular case of *initialize action* is to add initial tokens on a buffer as we are going to see on the following example.

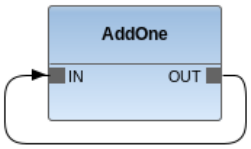

Figure 3.3: A close loop network with one actor.

Listing 3.7: An actor with an initialize action

```
package examples;
actor AddOne()
    int IN ==> int OUT:


    initialize ==> OUT:[0]
    end

    action IN:[token] ==> OUT:[token + 1]
    end

end
```

Considering that we want to create a close loop system that will run indeterminately. Here we have the AddOne actor which reads a token from its input and adds the value 1 on it and writes it to its output. Here the initialize action will write the value 0 on its output buffer, and as you can see in the Figure 3.3 the output of the actor is connected to its input port. If the initialize action was not presented in the actor, the action could not execute, as there is no data in its input.

3.7 Free running actions

It is possible to mix labeled actions that are used as transitions inside an FSM and actions that they are not controlled by the FSM. Here I am calling those actions free running actions. Sometimes in complicated actors there is a need to have an action that reads from input "asynchronously" from the rest of the actors algorithm, an example of such an actor can be a syntactical parser. A free running action, is an action that it will be checked for execution before an action defined in a state transition, thus free running actions have the highest priority if an actor has an FSM.

Listing 3.8: Example of free running action

```
package examples;
actor FreeRunningAction() int A, int B ==> int O:

    int MAX_TOKENS = 256;

    int Counter := 0;

    int mem[MAX_TOKENS];

    @native hash(int values[MAX_TOKENS]) --> int;

    bool done_reading := false;

    action A:[token] ==>
    guard
        Counter < MAX_TOKENS
    do
        mem[Counter] := token;
        Counter := Counter + 1;
    end

    action ==>
    guard
        Counter = MAX_TOKENS
    do
        done_reading := true;
    end

    process_A: action ==> O:[ hash(mem) ]
    do
        done_reading := false;
    end

    process_B: action B:[token] ==> O:[token]
    end

    send_zero: action ==> O:[0]
    do
        Counter := 0;
    end

    schedule fsm s0:
        s0 (process_A) --> s1;
        s1 (process_B)  --> s2;
        s2 (send_zero) --> s0;
    end

end
```

The Listing 3.8 shows an example for understanding the concept of free running actions, what the actor does is not important here. What is important is the order of execution of the actions. The two unlabeled actions (Orcc will give the following labels : untagged_0 and untagged_1) will always be checked before any other action called by the FSM transitions. If we assume that there are always tokens available in port A, then the execution of action is the following one:

1. untagged_0 up until its guard condition is false
2. untagged_1
3. process_A
4. untagged_0 up until its guard condition is false
5. untagged_1
6. process_B
7. send_zero
8. cycle is complete back to 1

Finally, as it can be observed the action selection for execution is normal because the actions outside the FSM are always checked for execution before the transitions in states.

3.8 Functions & Procedures

A CAL function can have N arguments and only one return value. It is possible to define variables, but those should be constants. The body of a function is an expression, and the result of the expression is going to be given in its output. A procedure in CAL does not return a value but it can modify the state of an actor.

As a recommendation you should use procedures in actions when you need to call many times the same source code. As for functions, you need to see them as lambda expressions, thus it is not allowed to have statements. But you can have constant statements defined in the var of each function. An example of constant statement is given in the Listing 3.9.

Listing 3.9: Constant statment in functions

```
function ConstStatementFunct(int a, int b) --> int :
var
    int c = if a > b then a else b
do
    c
end
```

3.9 Sharing constants and functions between actors, Unit cal files

In RVC-CAL its possible to have dedicated source files that contains constants, functions, native functions, native procedures that can imported by many actors, those source files are called units. A *unit* CAL file should be store on its own file with the same name as the unit. An example of unit is given in the Listing 3.10.

Listing 3.10: A CAL unit file

```
package examples;
unit UnitExample :

    int MAX_VALUE = 10;

    function PlusOne(int x) --> int :
        x + 1
    end

end
```

An example of using the unit in Listing 3.10 is given by the Listing 3.11.

Listing 3.11: A CAL actor that uses a unit

```
package examples;

import UnitExample.MAX_VALUE;
import UnitExample.PlusOne;

// -- If we want to import everything, the syntax is the same as Java
// import.UnitExample.*;

actor UsingAUnit()
    int A ==> int O:

    int Counter := 0;

    action IN:[token] ==> O:[PlusOne(token)]
    guard
        Counter < MAX_VALUE
    do
        Counter := Counter + 1;
    end

end
```

3.10 Handling communication issues

The datflow with firing MoC does not permit the lost of tokens between actors. But in some situations is possible to permit the loss of tokens programmatically. Both source and sink actors, which in the current context might be running on

IoT devices, might support the loss of information by using the guards expression in actions.

The solution is to have composite tokens that contains a timestamp information. Although, ORCC does not support composite types, it handles correctly integers with variable bitwidth. The Listing 3.12 and Listing 3.13 gives two implementations of a source and a sink actor. The source actor generates tokens with a timestamp, and the sink actor accepts tokens with an acceptable timestamp.

Listing 3.12: Source with timestamp

```
package examples;
actor SrcTimestamp() ==> int OUT:

    @native timestamp(uint(size=32) value) ==> uint(size=64) T;

    uint(size=32) Counter := 0;

    action ==> OUT:[token]
    guard
        Counter < 64
    var
        int token = timestamp(Counter)
    do
        Counter := Counter + 1;
    end

end
```

Listing 3.12 implements a simple *Source* actor that will produce 64 tokens with a timestamp and will write them to its output port. The timestamp function can be implemented using the ORCC native function support. Here, the timestamp function take a 32 bit unsigned integer as an argument and returns an unsigned integer of 64 bit which represents the composition of the token value and a timestamp. Then, this value is written to the output port OUT. The actor is going to execute only 64 times given the guard expression condition.

Listing 3.13 implements a simple *Sink* actor that consumes values with timestamps and prints its values in the standard output. The actor contains two native functions. The acceptable_timestamp native function takes as an argument a composite token with a timestamp and return true or false if the timestamp in the token is acceptable. The decode_from_timestamp native function accepts a composite token with a timestamp argument and returns the token value without the timestamp. To make it possible to remove tokens without a correct timestamp we need to implement two actions, one that will accept them and one that will decline them. In addition, we need to give a priority on which action should be checked first for execution. It is better to be pessimistic and always check first if a token might have a wrong timestamp, and thus we need to consume this token from the FIFO. Finally, we can also have many actions in our actor and thus we need an FSM for handling the computation. It is suggested that on the state that the reading is occurred (s0) we also have the priority conditions.

Listing 3.13: Sink with timestamp

```
package examples;
actor SinkTimestamp() uint(size=32) IN ==> :

    @native acceptable_timestamp(uint(size=64) value) ==> bool;

    @native decode_from_timestamp(uint(size=32) value) ==> uint(size=32);

    accept: action IN:[token] ==>
    guard
        acceptable_timestamp(token)
    do
        println("Value : " + decode_from_timestamp(token));
    end

    decline: action IN:[token] ==>
    guard
        not acceptable_timestamp(token)
    do
        // -- Just consume the token
    end

    priority
        decline > accept;
    end

    // -- Not necessary but just an example, if we want to schedule many
       actions
    schedule fsm s0 :
        s0 (accept) --> s0;
        s0 (decline) --> s0;
    end

end
```

Conclusion

This report has introduced key aspects on dataflow programming, its compiler infrastructure, and coding recommendations. Chapter 1 explains what is a model of computation and what makes the dataflow programing different to other classical approaches. Three models of computation KPN, DPN and ATS have been elaborated by using mathematical formulation, which is important if the interested developer needs to understand the theory behind the CAL programming language, and RVC-CAL a subset of CAL that is supported by the Orcc framework. Furthermore, a sufficient introduction to CAL programming language is included in this report, for more information on the language itself the interested developer should read the document entitled "CAL Language Report" by Johan Eker and Jörn Janneck for in depth analysis of this programming language.

Chapter 2 introduced an "ideal" RVC-CAL design flow. This design flow is composed of two directions: *top down implementation* and *iterative feedback*. The top down implementation of Orcc during this project has been extended for creating a framework for IoT devices. In this chapter, the intereseted developer can understand the structure of the Orcc compiler and how dataflow programs are being executed on a given platform. As an extension of this work, I suggest for the interested developers to include the *iterative feedback* direction of the design flow. A lot of research has been done in this domain and there are already available tools that are supporting Orcc. A framework called TURNUS, available on github, permits to find the minimal and optimal buffer size of dataflow network, multicore partitioning, and deep analysis on the CAL source code for finding hotspots in parallel programs.

Finally, Chapter 3 gives a lot of coding recommendation to ta new CAL developer. Different particularities when coding in CAL are described and reasons why a deadlock might appear is explained. The provided examples are simple so that the developer can easily understand the different concepts of coding in RVC-CAL. Concepts such as the actors finite state machine and priorities which is a must for a developer to code efficiently in CAL are explained to the detail.

Bibliography

[1] Eclipse modeling framework. http://eclipse.org/modeling/emf/. Accessed: 12-2014.

[2] Open RVC-CAL compiler. http://orcc.sourceforge.net/. Accessed: 12-2014.

[3] Xtend: Modernized java. http://www.altera.com/products/software/opencl/opencl-index.html. Accessed: 12-2014.

[4] Xtext: Language development made easy! https://eclipse.org/Xtext/. Accessed: 12-2014.

[5] S. Casale-Brunet. Analysis and optimization of dynamic dataflow programs. 2015.

[6] J. B. Dennis. First version of a data flow procedure language. In *Symposium on Programming*, pages 362–376, 1974.

[7] J. Eker and J. Janneck. CAL Language Report. Technical Report ERL Technical Memo UCB/ERL M03/48, University of California at Berkeley, Dec. 2003.

[8] J. Eker and J. Janneck. CAL Language Report. Technical Report ERL Technical Memo UCB/ERL M03/48, University of California at Berkeley, Dec. 2003.

[9] D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Normwell, Ma:Springer, 1st edition, 2000.

[10] J. Janneck. A machine model for dataflow actors and its applications. In *Signals, Systems and Computers (ASILOMAR), 2011 Conference Record of the Forty Fifth Asilomar Conference on*, pages 756–760, Nov 2011.

[11] G. Kahn. The Semantics of Simple Language for Parallel Programming. In *IFIP Congress*, pages 471–475, 1974.

[12] K. Keutzer, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: Orthogonalization of concerns and platform-based design. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 19(12):1523–1543, nov 2006.

[13] E. Lee and T. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773 –801, may 1995.

[14] E. Lee and T. Parks. Dataflow process networks. In *Proceedings of the IEEE*, pages 773–799, 1995.

[15] E. Lee and A. Sangiovanni-Vincentelli. Comparing models of computation. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 234–241. IEEE Computer Society, 1997.

[16] D. McAllester, P. Panangaden, and V. Shanbhogue. Nonexpressibility of fairness and signaling. *J. Comput. Syst. Sci.*, 47(2):287–321, oct. 1993.

[17] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

[18] A. Turing. On computable numbers with an application to the "Entscheidungsproblem". *Proceeding of the London Mathematical Society*, 1936.