Journal of Systems Architecture 101 (2019) 101657

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Verifying parallel dataflow transformations with model checking and its application to FPGAs

Robert Stewart[a,*], Bernard Berthomieu[d], Paulo Garcia[c], Idris Ibrahim[a], Greg Michaelson[a], Andrew Wallace[b]

[a] Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, UK
[b] Engineering and Physical Sciences, Heriot-Watt University, Edinburgh, UK
[c] Faculty of Engineering and Design, Carleton University, Ottawa, Canada
[d] LAAS-CNRS, Université de Toulouse, Toulouse, France

ARTICLE INFO

Keywords:
Dataflow
FPGAs
Model checking
Program transformation
Parallelism

ABSTRACT

Dataflow languages are widely used for programming real-time embedded systems. They offer high level abstraction above hardware, and are amenable to program analysis and optimisation. This paper addresses the challenge of verifying parallel program transformations in the context of dynamic dataflow models, where the scheduling behaviour and the amount of data each actor computes may depend on values only known at runtime.

We present a Linear Temporal Logic (LTL) model checking approach to verify a dataflow program transformation, using three LTL properties to identify cyclostatic actors in dynamic dataflow programs. The workflow abstracts dataflow actor code to Fiacre specifications to search for counterexamples of the LTL properties using the Tina model checker. We also present a new refactoring tool for the Orcc dataflow programming environment, which applies the parallelising transformation to cyclostatic actors. Parallel refactoring using verified transformations speedily improves FPGA performance, *e.g.*15.4× speedup with 16 actors.

1. Introduction

In the dataflow model of execution, the firing of actors depends only on data availability, allowing each actor in a program to execute asynchronously without a global control flow sequentialising their execution.

Dataflow languages are a natural abstraction for FPGAs, since a program's dataflow structure is distributed across the programmable hardware fabric. In other words, each actor in a dataflow program is compiled to an independent processing hardware block, subject to FPGA resource availability. Moreover, large kernel variables like intermediate arrays may use on-chip block RAM (BRAM). BRAMs are distributed across FPGA fabric, meaning there is no memory contention when multiple actors update their kernel variables, *e.g.* intermediate arrays. That is, both computation and memory access is inherently parallel. For FPGAs, dataflow programming environments represent a high level abstraction above Hardware Description Languages (HDLs) *i.e* Verilog or VHDL. The dataflow programming model can be exploited for high FPGA performance, *e.g.* [1], or as an Intermediate Representation in compilers for higher level FPGA languages *e.g.* [2]. The dataflow abstraction enables program analysis *e.g.* optimal static scheduling [3] and program transformation (this paper).

1.1. Parallel program transformation

The goal of parallelisation is to speed up performance by executing code across multiple processing elements *e.g.* the cores of a CPU. Parallel transformations must preserve a program's functional semantics, *i.e* a transformed program must have identical output for the same inputs as the original program.

Refactoring Software Code. Refactoring tools exploit language properties to parallelise code. For example, exploiting the referential transparency of pure code and equational laws to rewrite a *map*, which sequentially applies a function to every element in a collection, as a *parMap* to perform them in parallel. Software languages for which parallel transformation tools exist include Haskell [4], Erlang [5] and C++ [6].

Refactoring Embedded Systems Code. Static dataflow code is simple to parallelise because information about scheduling and data rates can be deduced at compile time. However, static dataflow language primitives are inexpressive, inhibiting the expression of algorithms with dynamic control flow and dynamic data rates. Introducing *dynamic* dataflow features to a language, *e.g.* value-dependent scheduling and dynamic data rates, complicates auto-parallelisation of dataflow code.

* Corresponding author.
E-mail address: r.stewart@hw.ac.uk (R. Stewart).

https://doi.org/10.1016/j.sysarc.2019.101657
Received 26 February 2019; Received in revised form 13 September 2019; Accepted 5 October 2019
Available online 23 October 2019

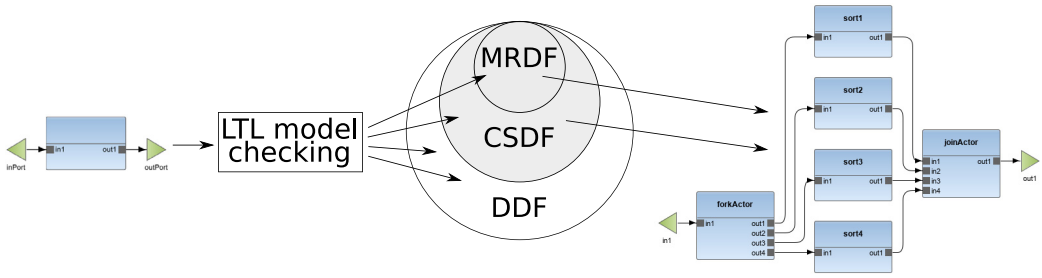


Fig. 1. Verifying and Transforming Cyclo-Static Dataflow.

1.2. Related work

This work is influenced by the Box Calculus [7], a program rewrite system for the Hume embedded systems language [8]. The approach specifies Hume programs with Lamport's temporal logic of actions (TLA) [9] and uses deductive verification in TLA to verify Box Calculus transformations [10]. This is exploited for parallelism by transforming single Hume boxes into multiple boxes using high level patterns like divide-and-conquer. The language for implementing Hume *boxes* (analogous to *actors*) is purely functional and stateless, mapping input patterns directly to output expressions and any state is carried via feedback wires. This contrasts with our work, where we aim to support stateful dataflow languages like CAL, where actors can have internal variables that persist values between firings. This complicates parallelisation, due to the possibility of changing the read/write order on mutable variables, or introducing race conditions. Moreover, Hume lacks tooling for programmers to apply program transformations. In contrast, our work uses model checking to verify transformations of stateful actors, and we have extended the Eclipse based Orcc IDE with a graphical tool to apply a parallel transformation when model checking enables it.

The most salient related work in the context of parallelising dataflow programs is the StreamIt compiler. That approach is restricted to parallelising stateless actors only [11], whereas our use of model checking enables the parallelisation of stateful actors too. Related work that combines dataflow models with model checking includes determining minimum dataflow buffer sizes [12], and enabling compile-time scheduling of multirate static actors [13]. On the hardware side, related work uses model checking to verify that two Verilog/VHDL modules satisfy the same global requirements [14] and hence enabling one to replace another with dynamic partial reconfiguration. None of these approaches consider the parallelisation of hardware designs or the verification of dataflow graph transformations.

1.3. Our approach and contributions

Approach. Our approach (Fig. 1) extends dataflow parallelisation beyond stateless actors, to actors that are stateful and contain firing rules with different data rates, provided they have cyclo-static properties. This is important because in practise cyclo-static actors very often hold state between action firings (Section 7.2). This is achieved by using model checking to identify parallelisable multi-rate static (MRDF) and cyclo-static (CSDF) actors that coexist with dynamic (DDF) actors in dataflow programs. These dataflow models are described in Section 2.2. The approach is general to any dynamic dataflow language, this paper demonstrates the approach with the CAL dataflow language [15] in the Orcc development environment [16]. It marks cyclostatic actors as parallelisable, to enable an interactive graphical program transformation tool.

Contributions This paper makes the following contributions:

- An abstraction of dataflow actors to Fiacre, a formalised language for representing behavioural and timing aspects of embedded and distributed systems (Section 3).
- Three Linear Temporal Logic (LTL) properties that model cyclo-static dataflow properties. Model checking Fiacre abstractions of actors identifies cyclostatic actors in dynamic dataflow programs (Section 4).
- A graphical interactive refactoring tool that automates the parallelisation of potentially stateful cyclostatic actors (Section 5).
- An evaluation showing counterexamples of the cyclostatic LTL properties, and the efficacy of parallel transformations of a cyclostatic actor on an FPGA with two case studies (Section 6).

2. Background

Task parallel languages provide either a high level programming model of spawning tasks and defining synchronisation points, or a lower level model with a fixed set of actors, explicit point-to-point FIFO connections and FIFO depths. *Software* oriented task parallel models are often more high level than parallel languages for *hardware*. Software languages often support spawning new tasks/actors at runtime across threads on a multicore CPU. This is unsuitable when compiling programs to programmable hardware (*i.e.* application specific circuits), where the fixed task graph is mapped into the hardware with place and route, and cannot be changed without re-synthesis.

There are numerous programming models to express task parallel programs, *e.g.* fork/join APIs and threads, where data flows are implicit between parallel tasks. Dataflow programming models offer a more explicit way to express independent parallel tasks, and which tasks communicate. They clearly separate computation and communication, and are popular for embedded systems programming, especially FPGAs, where the entire task graph and communication routing must be fixed in hardware prior to execution.

2.1. Properties of dataflow languages

Dataflow programs are directed graphs of connected actors. There are many dataflow languages for multicore processors and embedded systems, each exhibiting a trade-off between expressivity and reasoning power. The hardware architecture they are designed to target reflects their functionality, specifically the graph structure, scheduling and data rates of programs expressed with them, shown in Fig. 2.

Graph structure. Some languages support dynamic task graphs, *i.e.* where the number of actors change during runtime. An example is Cilk [17], a C++ fork/join model that creates parallel threads with `spawn` and synchronises their results with `sync`. This is similar to OpenMP's `task` thread creation and `taskwait` synchronisation pragmas [18]. With other languages, the number of tasks (or actors) and the task graph topology is static at compile time, and does not change at runtime.

Scheduling. This is the selection of executable code *within* an actor. Some dataflow languages support only fixed scheduling policies *e.g.* PREESM [19], whilst other models support code execution choice driven by data availability and data values using pattern matching *e.g.* Hume [20] and CAPH [21]. Some languages

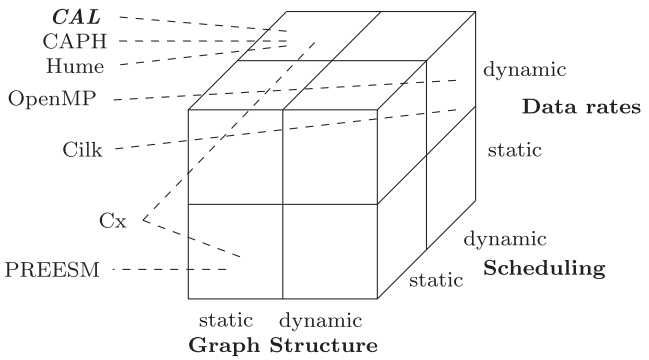


Fig. 2. Task Parallel Languages.

support both, *e.g.* Cx [22] has non-blocking reads/writes on *default* ports (static scheduling) and *synchronised* ports (dynamic scheduling), where data availability determines if an executable rule is enabled.

Data rates. An actor with a *static* data rate produces and consumes the same number of values every time it is executed. An actor with a *dynamic* data rate is free to consume/produce data sequences of arbitrary length with no periodic pattern.

2.2. Dataflow models

The models considered in our approach are:

1. *Multi-Rate Dataflow (MRDF)* [23], where actors consume and produce a fixed number of tokens every firing.
2. *Cyclo-Static Dataflow (CSDF)* [24], where actors contain multiple fireable actions. Actions may have different consumption/production rates, but the sequence of action executions must be periodic.
3. *Dynamic Dataflow (DDF)* [25], where, when a DDF actor is fired its scheduler picks an *enabled* execution rule and executes it. The enabled status of an execution rule can depend on both the *availability* and the *values* of tokens, and hence the data rate is unknown at compile time. If there are no fireable rules, execution is deferred and the actor's scheduler tries again to find an enabled execution rule.

Dataflow actors can have multiple actions but only one action can fire at a time, in contrast to *synchronous* languages *e.g.* Quartz [26], where *all* enabled actions execute in a single firing.

Some dataflow programming environments only support static dataflow models, *e.g.* direct feedthrough function blocks in Simulink [27]. The open source Orcc development environment [16] supports dynamic dataflow programming with CAL [15].

Real world application of dynamic dataflow properties include real-time Twitter data processing [28] and MPEG decoding, a widely used dataflow benchmark with practical relevance. [29] presents an MPEG implementation in CAL. The disadvantage of dynamic dataflow is that it inhibits program analysis, meaning a programmer has to revert to manual program refactoring to improve performance, and although this can improve FPGA performance [30], this approach is cumbersome and error-prone.

2.3. Constructing static dataflow graph structures with CAL

CAL is a dataflow language for programming real-time embedded systems, which was developed as part of the Ptolemy II project ([31]). It is a dataflow language for constructing dataflow graphs with (1) a **fixed set of actors and connections**, (2) support for **dynamic scheduling** and (3) support for **dynamic data rates**. These two dynamic properties make CAL suitable for implementing relatively complex algorithms on embedded systems, *e.g.* algorithms for which control flow and the input/output data size is determined by runtime values, at the expense of lost static analysis and program transformation opportunities due to these dynamic properties.

Each CAL actor encapsulates an algorithmic kernel. Actors are connected together with lossless order-preserving FIFOs, through which tokens flow. Fig. 3a shows a volume amplifier example. Actor ports are named ❶. The actor may have a store of private variables ❷. Actors contain discrete non-interruptible execution rules, called actions. Actions match on input patterns to consume tokens and output patterns produce tokens (❸ and ❺). Actions may also update store variables ❹. A Finite State Machine (FSM) determines enabled actions from each FSM state ❻, and the initial FSM state which in this example is `s0`. When multiple actions are fireable from a given state, a priority block can disambiguate multiple enabled actions ❼.

2.4. Implementing dynamic dataflow languages

Since action selection for dynamic actor firing is a runtime decision, implementations of dynamic dataflow models need a runtime scheduling mechanism to determine which action to execute next. Fig. 4 shows the architecture components necessary for supporting dynamic dataflow execution. Each actor consumes input streams and produces output streams via *ports*. *Connections* (edges) enable data to flow between ports. An internal store associates values with actor-local variables.

CAL includes a primitive called `guard` (Section 3.1), which supports the firing of an action depending on the *values* of incoming tokens and of variables in the actor's store. Satisfying both of the following conditions enables an action for firing:

1. there is a sufficient number of tokens to match its input patterns,
2. its guard evaluates to true (if a guard exists).

Fig. 5 shows the scheduling algorithm for action selection. If there are multiple enabled actions, and an absence of a priority statement to disambiguate action selection, the runtime scheduler chooses one at random.

2.5. Actor data rates

An FSM transition system drives actor execution, *e.g.* the FSM in Fig. 3b for the amplifier actor. Actions have data rates [*C*/*P*], meaning they consume *C* tokens and produce *P* tokens when transitioning between FSM states when their actions are fired.

Consider an actor *A* consuming tokens from a connection *v* and producing tokens on a connection *u*. Firing an action in this actor will consume c_A^v tokens from connection *v* and produce p_A^u tokens to connection *u*. Successive execution of actor *A* may select different actions internally, *e.g.* $c_A^v(1) = 3$ means the 1st firing of *A* resulted in a consumption of 3 tokens whilst $c_A^v(2) = 5$ means that on the 2nd firing 5 tokens were consumed. More generally, $c_A^v(n)$ and $p_A^u(n)$ are the number of tokens consumed and produced on the n^{th} firing of actor *A*.

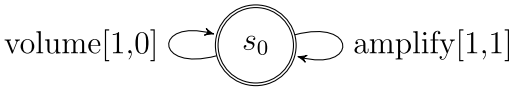
Definition 1. *Data rates.* The number of tokens produced on edge *u* after *n* firings for actor *A* is $P_A^u(n) = \sum_{i=1}^{n} p_A^u(i, a)$ where $a \in Actions_A$. The number of tokens consumed from edge *v* after *n* firings for actor *A* is $C_A^v(n) = \sum_{i=1}^{n} c_A^v(i, a)$.

Consider actor *A* with actions a_1 and a_2, both enabled after 5 actor firings. If $c_A^u(5, a_1) \neq c_A^u(5, a_2)$, then the actor's data rate may not be static, *i.e.* it would be a dynamic actor, and it may not be possible to answer:

1. can the program execute with bounded buffer size?
2. will parallelising an actor preserve the actor's functional semantics?

```
❶ actor Amplifier () In, Volume ==> Out :
❷    vol := 1;
❸    volume: action Volume:[newV] ==>
❹      do vol := newV; end
❺    amplify: action In:[d] ==>
                  Out:[d*vol] end
❻    schedule fsm s0 :
         s0 (volume)  --> s0;
         s0 (amplify) --> s0;
     end
❼    priority volume > amplify; end
end
```

(a) Actor Implementation



(b) Actor FSM Execution Model

Fig. 3. An Amplifier Dataflow Actor.

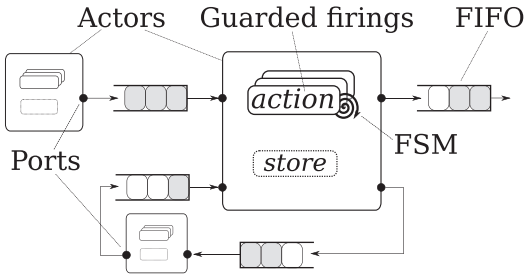


Fig. 4. An Architecture for Dynamic Actors.

2.6. Program preserving parallelisation

Parallelism can speed up execution. However, parallelisation (and any other program transformation) must preserve a program's functional semantics. Two properties of dynamic dataflow programs that must be preserved by parallel program transformation are:

1. **Functional data rates:** An actor may require multiple input tokens to compute its function and produce an output. Such an atomic operation is often implemented as multiple action firings (Section 7.2). Parallel instances of an actor should read and write enough tokens to preserve the atomic behaviour of the original sequential actor.
2. **Stores:** An actor may have a store of local variables. A program transformation should preserve the modification sequence of an actor's store.

Section 4 shows that, with model checking, one can parallelise actors expressed with a dynamic dataflow language to increase its throughput performance, whilst preserving the actor's functional semantics.

3. Abstracting actors to models for verification

Our approach uses model checking to identify cyclostatic actors for parallelisation. Since CAL is a dataflow programming language and not a model description language, this section shows how we abstract pertinent features of actor code into Fiacre.

3.1. Abstract dataflow transition system for CAL

An abstract transition machine model for describing dataflow actors [32] is now presented to formalise the actor model. It describes the data rates and scheduling rules for actors in terms of transitions on an abstract machine. The firing of actions is determined by an actor's FSM, a labeled transition system between control states. An actor is a sequential process, communicating values by transitioning between states in the actor's FSM. Executions of statements in the body of an action can compute output token values and can update the store. The state machine receives tokens and reacts to them, possibly entering another state, and possibly producing tokens. The dataflow transition:

$$\sigma\langle action_1 \leftarrow e\rangle \xrightarrow{s \mapsto s'} \sigma'\langle action_2 \leftarrow e'\rangle \qquad \text{(fire)}$$

describes a transition from FSM control state σ to σ' by firing $action_1$, and at σ' the enabled action is $action_2$. The store e with the input stream s is the input state, and the firing of $action_1$ produces a new store e' and outputs the s' stream. In the following example:

```
int accum := 0;
action sumStream in1:[x] ==> out1:[accum] :
do accum = accum+x; end
```

The store e is `{accum=0}` before the 1st firing. The action is `sumStream`, the input stream s is `[x]` consumed from input port `in1`. The new store e' is `{accum=0+x}` after the firing.

The following *(fire-guard)* rule supports dynamic dataflow. It adds a predicate function that guards the firing of the transition. This function *guard E* must evaluate to *true* for the action to fire. The E predicate can

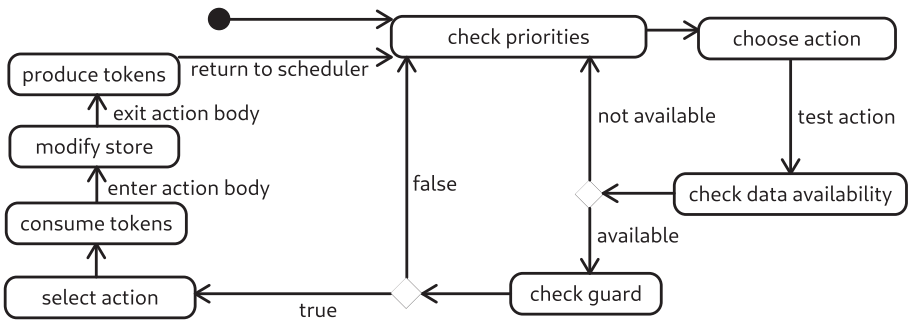


Fig. 5. Scheduling Algorithm to Execute Dynamic Actors.

use token values from s and store values from e.

$$\frac{e \vdash E \qquad guard\ E \rightsquigarrow true}{\sigma\langle action_1 \leftarrow e\rangle \xrightarrow[guard\ E]{s \mapsto s'} \sigma'\langle action_2 \leftarrow e'\rangle} \quad \text{(fire-guard)}$$

An example is:

```
int accum;
int count;
action sumTen in1:[x] ==> :
guard (count < 10)
do accum := accum + x;
   count := count + 1;
end
```

This `sumTen` action consumes token `x`, adding it each time to `accum` in the store with 10 successive firings. The *guard E* predicate checks that `count` is less than 10. Another action (omitted) might produce the value of `accum` on the 11th firing, resetting the count and accumulation variables.

If there are multiple fireable actions from σ' then we use • to indicate a runtime scheduler choice to disambiguate two or more actions that both can fire from the same control state. Hence when there are multiple enabled actions at σ' we write:

$$\frac{e \vdash E \qquad guard\ E \rightsquigarrow true}{\sigma\langle action_1 \leftarrow e\rangle \xrightarrow[guard\ E]{s \mapsto s'} \sigma'\langle \bullet \leftarrow e'\rangle} \quad \text{(fire-guard)}$$

An example of two *(fire-guard)* actions in an actor is:

```
action positivesRepeat    in1:[x] ==> out1:[x,x] :
guard (x > 0) end

action negativesIdentity in1:[x] ==> out1:[x] :
guard (x <= 0) end
```

The `positivesRepeat` action produces twice any positive integer consumed, whilst `negativesIdentity` simply propagates any negative or 0 values consumed. Here, • means that after firing either of these, the scheduler will not know which action to fire next until the value of `x` becomes known after consuming the token on the next firing. This becomes problematic for verifying cyclo-static data rates when two or more ambiguous actions have different production or consumption rates.

3.2. Actor abstractions

The CAL primitives we abstract for model checking are:

1. **The production and consumption rates of each action**. We need the model checker to verify if there exists a cyclic data rate sequence.
2. **Updates to variables in an actor's store**. These variables can be used to conditionally fire actions, *i.e.* the *(fire-guard)* rule.
3. **Statements inside action bodies**. Specifically, if/then/else, variable assignment and loop statements, because they can update variables in an actor's store.
4. **Actor FSMs**. They enforce action scheduling and we need to know (1) if exactly one periodic cycle of visited states exists, and (2) the cycle's data rate, *i.e.* the sum of data rates for each action representing a state transition in the cycle.

3.3. Fiacre

We abstract CAL code to Fiacre [33] to abstract actor to a model for checking cyclostatic properties. Fiacre is a formal intermediate model to represent both the behavioural and timing aspects of embedded and distributed systems for formal verification. It is a target language of program modelling tools for use with verification engines. From Fiacre specifications, the Fiacre compiler produces enriched Petri nets, in which transitions can test and modify a set of data variables. Therefore Fiacre provides a high level, compositional syntax for Time Petri Nets. TINA [34] is a model checking toolset to analyse these nets.

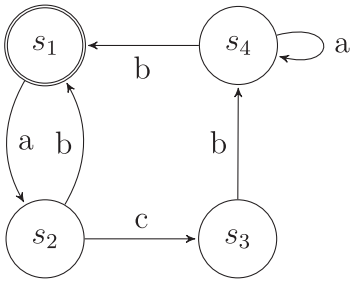


Fig. 6. Automota for Fiacre Process P.

3.3.1. Fiacre language

The primary Fiacre primitive is a *process*. Fiacre processes can contain deterministic statements including assignment and loops (Section 3.4). As with CAL actors, a Fiacre process describes sequential behaviour defined by a set of control states and a set of process transitions. Processes can contain non-deterministic choice, which we map FSM scheduling to in Section 3.5. CAL actor ports are mapped to Fiacre process ports, and dataflow connections are mapped to Fiacre channels.

The following `Product` Fiacre example[1] reads pairs of integers from port `b` then sends their product out from port `a`.

```
channel bytepair is byte # byte
process Product [a:nat, b:input bytepair] is
  states s0, s1
  var x,y: byte
  from s0 b?x,y; to s1
  from s1 a!x*y; to s0
```

The evaluation sequence in a Fiacre process is guided by a labelled transition system. The Fiacre process P below has four control states with labelled transitions between them. Ports denote here pure synchronization labels (without communications). The automaton for this process is in Fig. 6.

```
process P [a,b,c:sync] is
  states s1, s2, s3, s4
  from s1 a; to s2
  from s2 select b; to s1 [] c; to s3 end
  from s3 b; to s4
  from s4 select a; to s4 [] b; to s1 end
```

In the abstraction for model checking, actor FSMs map naturally to Fiacre process transitions. The transition labels are the actions to execute to get to new FSM control states. In our approach, action code bodies are abstracted (Section 3.4) and then inlined into Fiacre transitions between control states.

3.3.2. Fiacre evaluation rules

The Fiacre semantics presented here are taken from [36]. Labelled relations express the semantics of Fiacre statements.

The evaluation rules that follow have the shape:

$$\frac{P_1 \ldots P_n}{e \vdash E \rightsquigarrow v}$$

which states that under conditions P_1 to P_n, the value of expression E with store e is v, where $\rightsquigarrow$ is expression evaluation.

[1] from [35].

The semantics of statements is expressed operationally by a labelled relation. The big step relation holds triples $(S, e) \overset{l}{\Rightarrow} (S', e')$ in which S is a statement, e and e' are stores, $S' \in \{\texttt{done}\} \cup \{\texttt{self}\} \cup \{\texttt{target}\ s | s \in \Lambda\}$, where Λ is the declared set of states of the process. The $\Rightarrow$ transition updates the store from e to e', moving execution to statement S', and the label is either a communication action l or silent action ϵ.

3.4. Abstracting dataflow actions to small step fiacre evaluation rules

This section shows the abstraction of the pertinent CAL code primitives given in Section 3.2, *i.e.* procedures, data rates, and actor FSMs, to small step Fiacre rules. Each abstraction rule from actors to processes is written $[\![\ ]\!] = \ldots$, which translates CAL to Fiacre evaluation rules.

3.4.1. Action procedures

Action bodies may include statements that modify the actor's store. The translation maps these statements, *i.e.* for loops and if/then/else statements, to deterministic Fiacre process statements.

Conditional if/then/else CAL statements are translated to the following evaluation rules:

$$[\![\textbf{if}\ E\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end}]\!]$$

$$= \frac{e \vdash E \rightsquigarrow true \qquad (S_1, e) \overset{l}{\Rightarrow} (S, e')}{(\textbf{if}\ E\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end}, e) \overset{l}{\Rightarrow} (S, e')} \quad \text{(if-then-else)}$$

$$\frac{e \vdash E \rightsquigarrow false \qquad (S_2, e) \overset{l}{\Rightarrow} (S, e')}{(\textbf{if}\ E\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end}, e) \overset{l}{\Rightarrow} (S, e')} \quad \text{(if-then-else)}$$

Conditional while loops are translated as:

$$[\![\textbf{while}\ E\ \textbf{do}\ S\ \textbf{end}]\!]$$

$$= \frac{e \vdash E \rightsquigarrow true \qquad (S; \textbf{while}\ E\ \textbf{do}\ S\ \textbf{end}, e) \overset{l}{\Rightarrow} (S', e')}{(\textbf{while}\ E\ \textbf{do}\ S\ \textbf{end}, e) \overset{l}{\Rightarrow} (S', e')} \quad \text{(while)}$$

$$\frac{e \vdash E \rightsquigarrow false}{(\textbf{while}\ E\ \textbf{do}\ S\ \textbf{end}, e) \overset{\epsilon}{\Rightarrow} (\texttt{done}, e)} \quad \text{(while)}$$

CAL loops with a fixed iteration count are translated as:

$$[\![\textbf{foreach}\ v_1.v_2\ \textbf{do}\ S\ \textbf{end}]\!]$$

$$= \frac{(V := v_1\ ;\ S \ldots V := v_2\ ;\ S, e) \overset{l}{\Rightarrow} (S', e')}{(\textbf{foreach}\ V\ \textbf{do}\ S\ \textbf{end}, e) \overset{l}{\Rightarrow} (S', e')} \quad \text{(for each)}$$

For assignment, Fiacre offers a number of forms but we only need the simplest here. $e[X \mapsto u]$ is store e extended with the pair (X, u).

$$[\![v := E]\!] = \frac{e \vdash E \rightsquigarrow u}{(X := E, e) \overset{\epsilon}{\Rightarrow} (\texttt{done}, e[X \mapsto u])} \quad \text{(assignment)}$$

CAL action bodies can contain multiple procedures, *i.e.* a sequence of assignments, if/then/else blocks and loops. Fiacre statements can be compositions of other statements, and this is used to compile multiple CAL statements $(S_1; S_2)$, *e.g.* an if statement following a while loop, to the following Fiacre *(composition)* evaluation rules. In the second rule, the static semantics of Fiacre ensures that at most one of labels l_1 and l_2 is not empty, so $l_1.l_2$ resumes to either l_1 or l_2.

$$[\![S_1; S_2]\!]$$

$$= \frac{(S_1, e) \overset{l}{\Rightarrow} (\texttt{target}\ s, e')}{(S_1; S_2, e) \overset{l}{\Rightarrow} (\texttt{target}\ s, e')} \quad \text{(composition)}$$

$$\frac{(S_1, e) \overset{l_1}{\Rightarrow} (\texttt{done}, e') \qquad (S_2, e') \overset{l_2}{\Rightarrow} (S', e'')}{(S_1; S_2, e) \overset{l_1.l_2}{\Rightarrow} (S', e'')} \quad \text{(composition)}$$

3.4.2. Actions with guards

Actions with guards can only fire if the guard predicate function evaluates to true in the current context. Fiacre has an **on** statement, which blocks if its expression evaluates to false that prevents execution of subsequent statements. The mapping models the actor guard predicate *guard E* with **on**, s is the input queue and s' is the output queue for $action_1$ when executed. The execution of a sequential composition of Fiacre statements abstract from action bodies depends on the boolean value of the **on** condition.

With an environment context of an actor's store and input tokens s, the abstraction of action guards is:

$$[\![guard\ E]\!] = \frac{e \vdash E \rightsquigarrow true \qquad (S, e) \overset{l}{\Rightarrow} (S', e')}{(\textbf{on}\ E\ S, e) \overset{l}{\Rightarrow} (S', e')} \quad \text{(on)}$$

3.4.3. Actor communication

Actors share the results of their computations by explicitly passing tokens to each other via connections between their ports. The parallelising transformation of actors in Section 5 needs to know its cyclo-static data rate, so we capture token communication in the abstraction. That is, during execution of Fiacre statements we keep a count of the incoming and outgoing tokens into and out of the process.

The syntax for token communication in Fiacre is `!` and `?`, corresponding to the *(send)* and *(receive)* Fiacre rules respectively:

$$\frac{e \vdash E_1 \rightsquigarrow v_1 \qquad e \vdash E_n \rightsquigarrow v_n}{(p_\tau !E_1, \ldots, E_n, e) \overset{p_\tau\ v_1, \ldots v_n}{\Longrightarrow} (\texttt{done}, e)} \quad \text{(send)}$$

$$\frac{e' = e[X_1 \mapsto v_1, \ldots, X_n \mapsto v_n] \qquad [e' \vdash E \rightsquigarrow true]}{(p_\tau ?X_1 \ldots X_n [\textbf{where}\ E], e) \overset{p_\tau\ v_1, \ldots v_n}{\Longrightarrow} (\texttt{done}, e')} \quad \text{(receive)}$$

Communications are sequences $p_\tau\ v_1\ \ldots\ v_n$ where p_τ is a port identified by a label τ and the $v_1\ \ldots\ v_n$ are values.

Fiacre has a single-communication constraint whereby only one *(send)* or *(receive)* statement can occur in a *(process-action)* execution. In Fiacre, values are consumed from ports with `?`, *e.g.* `in1?x`. Whilst it is possible to consume multiple values from a Fiacre port, *e.g.* `in1?x,y,z`, this is constrained by the data type of the port, *e.g.* where the `bytepair` channel is a `byte # byte` tuple in the earlier example in Section 3.3.1, where exactly two values must be consumed every time, *e.g.* with `in1?x,y`. The same Fiacre constraint applies for producing values with `!`, *e.g.* `out1!x,y`. The single communication rule is enforced by Fiacre's type checker to ensure that only one label is ever carried by a $\Rightarrow$ transition update. In CAL, values are consumed/produced from ports by pattern matching, *e.g.* `in1:[x,y,z]` and `out1:[3,x+2,0]`.

The communication semantics for dynamic dataflow differ from Fiacre's single-communication semantics in two ways:

1. An action can have multiple communication events, because an action can have both an input pattern and an output pattern and these patterns can match on multiple input/output ports.
2. Actions within the same actor can pattern match on the same ports but consume/produce a different number of tokens, *e.g.*:

```
actor A int in1 ==> int out1 :
  action actn1 in1:[x]   ==> out1:[x,x] : end
  action actn2 in1:[x,y] ==> out1:[x]   : end
end
```

Therefore to support dynamic dataflow with Fiacre's single-communication constraint, we give the Fiacre process ports a channel type. For each token in an action's input pattern, Fiacre's *front* function on the queue reads from this internal channel, the value is then removed with *dequeue* and the counter monitoring how many tokens are consumed is incremented. Likewise for an action's output patterns, a Fiacre queue receives all output tokens, and the counter monitoring how many tokens are produced is incremented by the size of that queue.

$$\frac{}{(\textbf{to}\ s, e) \overset{\epsilon}{\Rightarrow} (\texttt{target}\ s, e)} \quad \text{(to)}$$

$$\frac{(S, e) \overset{l}{\Rightarrow} (S', e')}{(\textbf{from}\ s\ S, e) \overset{l}{\Rightarrow} (S', e')} \quad \text{(from)}$$

$$\frac{(S_i, e) \overset{l}{\Rightarrow} (S', e')}{(\textbf{select}\ S_1\ [\,] \ldots [\,]\ S_n\ \textbf{end}, e) \overset{l}{\Rightarrow} (S', e')} \quad \text{(select)}$$

Fig. 7. Abstracting Actor Scheduling.

Actions that have multiple communication events, *i.e.* multiple tokens in input/output patterns or consuming/producing from/to multiple ports, are separated into multiple intermediate single-communication events when abstracted to Fiacre.

3.4.4. Actor schedules to process selections

The selection of actions in dataflow actors is determined by the actor's FSM between control states. Given an actor's FSM that specifies multiple next states from the current state σ, the abstraction maps to Fiacre's `select` statement as:

```
from σ select (S1;..; to σ')
              [] (Si;..; to σ'')
              [] ...
end
```

Where `[]` separates the non-deterministic choices. The following actor FSM example has one control state `s0`:

```
schedule fsm s0 : s0 (actn1) --> s0;
                  s0 (actn2) --> s0;
end
```

These FSMs are abstracted to Fiacre selection statements. Consider the following contrived example for `actn1` and `actn2`, where both consume a token but produce no tokens:

```
action actn1 in1:[x] ==> :
guard (x>0) end

action actn2 in1:[x] ==> :
guard (x<=0) end
```

Because the actions `actn1` and `actn2` have one communication event, and a guard (abstracted to an `on` statement after consuming `x`), the FSM will be abstracted to the following Fiacre selection statement:

```
from s0 select in1?x; on x>0; to s0
            [] in1?x; on x<=0; to s0
end
```

The abstraction of actor FSMs uses three Fiacre rules *(from)(to)* and *(select)*, shown in Fig. 7. This requires FSM transitions to be grouped by the control states, for all control states Σ in the FSM, that transitions start from. The *(from)* rule is used for each source state, the action used to transition between states is then abstracted and inlined into each corresponding transition, and finally the *(to)* rule is appended to these statements to arrive at the destination state, shown with set comprehension as follows:

$$\begin{aligned}\llbracket FSM \rrbracket = \{(\textbf{from}\ \sigma\ (\textbf{select}\ (\llbracket action \rrbracket; (\textbf{to}\ \sigma'))^+\ \textbf{end}))\\ \mid \sigma, \sigma' \in \Sigma\\ \wedge \{(action, \sigma')\ \mid (\sigma, action, \sigma') \in FSM\}\}\end{aligned}$$

This restructures the FSM to group transitions by their source state, to map actor FSMs to the syntactic structure of Fiacre's *select* statement.

3.5. Abstracting action firing to big step fiacre evaluation rules

The previous section abstracted data rates, procedures and actor FSMs to small step Fiacre rules. We now require a big step Fiacre rule to abstract the atomic firing of a dataflow actor's action.

The following big-step *(process-action)* rule abstracts the sequential composition of multiple small-step executions. It describes statement executions between *(from)* and *(to)* rules, *i.e.* originating from an initial control state and arriving at a target state, where $\rightarrow$ is a process action getting from control state s to s' and $\overset{l}{\Rightarrow}$ is a big-step statement execution with the *(process-action)* rule:

$$\frac{(\textbf{from}\ s\ S, e) \overset{l}{\Rightarrow} (\texttt{target}\ s', e')}{(s, e) \rightarrow (s', e')} \quad \text{(process-action)}$$

A transition from **from** to the target control state in the *(process-action)* rule happens in totality or not at all. If a transition includes an **on** statement then tokens produced/consumed will only be consumed if the **on** predicate evaluates to *true*. This peek semantics is equivalent to CAL's semantics for *guard*.

This big step rule corresponds to firing a dataflow actor's action, *i.e.* the *(fire)* or *(fire-guarded)* dataflow execution rules from Section 3.1, which capture the actor CAL code we intend to model check. One big step *(process-action)* evaluation corresponds to multiple small step evaluations of action selection *(select)*, production and consumption rates *(send)* and *(receive)*, and procedures inside an action body *i.e. (assignment), (if-then-else)* and *(foreach)*.

Consider an $action_1$ with an action body with a sequence of statements $S_1; S_2$, and the body of an $action_2$ with statement S_3. Abstracting this *(fire-guard)* to Fiacre is as follows:

$$\llbracket \sigma\langle (S_1; S_2) \leftarrow e\rangle \overset{s \mapsto s'}{\Rightarrow} \sigma'\langle (S_3) \leftarrow e_2\rangle \rrbracket \quad \text{(fire)}$$

$$= \llbracket S_1 \rrbracket = (S_1, e) \overset{l_1}{\Rightarrow} (S_2, e_1)$$

$$\llbracket S_2 \rrbracket = (S_2, e_1) \overset{l_2}{\Rightarrow} (S_3, e_2)$$

$$\frac{(\textbf{from}\ \sigma\ (S_1; S_2, e) \overset{l}{\Rightarrow} (\texttt{target}\ \sigma', e_2)}{(\sigma, e) \rightarrow (\sigma', e_2)} \quad \text{(process-action)}$$

Consider the same action, but this time with a guard predicating its firing. Abstracting this *(fire-guard)* to Fiacre is as follows:

$$\left\llbracket \frac{e \vdash E \qquad guard(s, E) \rightsquigarrow true}{\sigma\langle (S_1; S_2) \leftarrow e\rangle \overset{l}{\Rightarrow} guard(s, E)]s \mapsto s'\sigma'\langle (S_3) \leftarrow e_2\rangle} \right\rrbracket = \quad \text{(fire-guard)}$$

$$\llbracket S_1 \rrbracket = (S_1, e) \overset{l_1}{\Rightarrow} (S_2, e_1)$$

$$\llbracket S_2 \rrbracket = (S_2, e_1) \overset{l_2}{\Rightarrow} (S_3, e_2)$$

$$\frac{(\textbf{from}\ \sigma\ ((\textbf{on}\ E\ S_1); S_2, e) \overset{l}{\Rightarrow} (\texttt{target}\ \sigma', e_2)}{(\sigma, e) \rightarrow (\sigma', e_2)} \quad \text{(process-action)}$$

3.6. Tracking data rates

We introduce a `P_static` boolean and two numbers `P` and `P_prev` that count the number of produced tokens in the current and previous cycle respectively, to check for cyclostatic data rates. Variables `C_static`, `C` and `C_prev` are injected to track consumption rates.

To demonstrate this with an example, the CAL actor implementation on the left of Fig. 8 is a predicate filter that outputs only positive integer input tokens and discards negative integers. The actor scheduler is:

$$s0\langle pos \leftarrow \{\}\rangle \xrightarrow[x>=0]{[x] \mapsto [x]} s0\langle \bullet \leftarrow \{\}\rangle$$

$$s0\langle neg \leftarrow \{\}\rangle \xrightarrow[x>=0]{[x] \mapsto [x]} s0\langle \bullet \leftarrow \{\}\rangle$$

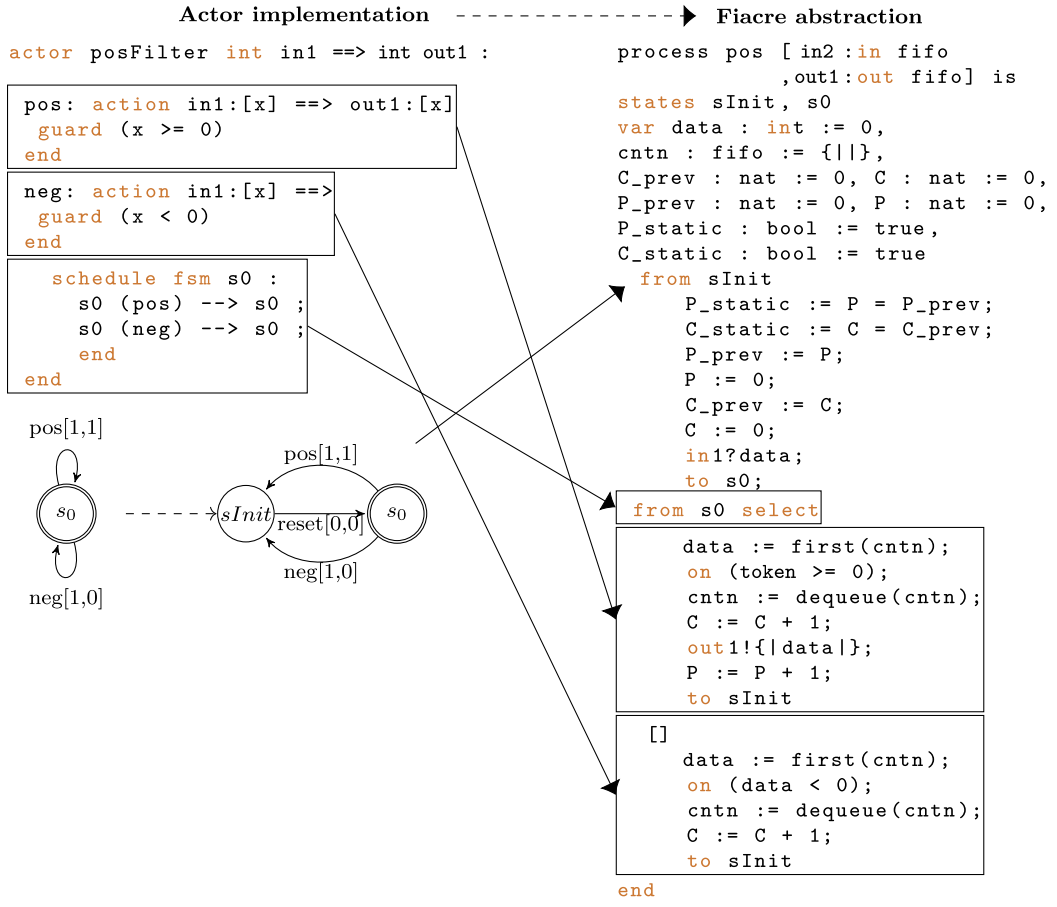


Fig. 8. Translating a Predicate Filter to Fiacre.

The Fiacre translation is on the right of Fig. 8. In the first complete cycle of a cyclo-static actor, *i.e.* $(s0, e) \rightarrow (s0, e)$, `P_static` and `C_static` will be false since this cycle computes the actor's production rate. In all subsequent cycles, these are *true* if the actor has a fixed data rate.

To determine data rates for cyclostatic actors, the abstraction to Fiacre intercepts all actor transitions to the user defined initial state, *e.g.* $s0$, introducing an additional Fiacre state *sInit* that compares previously recorded data rates for $s0 \rightarrow . \rightarrow s0$. The *sInit* $\rightarrow s0$ Fiacre transition also consumes tokens from the FIFO for this transition sequence.

4. Cyclo-Static actors as LTL formula

Once the mapping abstracts Fiacre process descriptions from actors, the Fiacre compiler translates them to enriched Petri nets ([37]) for the purpose of LTL verification by TINA to search for counterexamples of cyclostatic properties. LTL (Linear Temporal Logic) is a temporal logic in which a formula can encode time (logical) and future state observations along runs, where each state in time has a single successor.

4.1. LTL Model checking approach

The LTL formula we use has logical operators negation (**not**), conjunction (**and**), disjunction (**or**) and implication ($\Rightarrow$), and temporal operators *eventually* ($\Diamond$), *always* ($\Box$) and *until* (**until**). In addition, it has atomic properties asserting that a particular process instance in the Fiacre description is in some particular Fiacre state and that one of its variables has a particular value. In the following LTL formulas thse atomic properties are written **State(s)** and **Value(x=v)**, respectively.

Our LTL model checking approach is:

1. Translate the actor to a Fiacre process (Section 3).
2. Abstract an actor's initial FSM state and store variable values into three LTL formulas (Sections 4.2.1, 4.2.2 and 4.2.3).
3. Model check the Fiacre process for counterexamples of the LTL properties that model cyclostatic actors.
   - If a counterexample cannot be found, the actor is classified as a multirate static or cyclostatic actor and can be parallelised using the algorithm in Section 5.2.
   - Otherwise the actor is classified as dynamic and is not suitable for parallelisation.

4.2. The cyclo-static assertions

Cyclo-static actors always have an infinitely repeating periodic cycle of FSM transitions between control states. We capture cyclostatic dataflow semantics in the following LTL properties:

1. The variables in an actor's store are bound to their initial value after each periodic cycle. Property *cyclic_store*, Section 4.2.1.
2. There exists a periodic cycle between FSM control states. Property *periodic_sequence*, Section 4.2.2.
3. The data rates of an actor are static. Property *static_rate*, Section 4.2.3.

The model checker searches for violations of these LTL properties to identify dynamic actors. We now give more detail of these three LTL properties.

4.2.1. Periodic process configuration

This LTL property is for checking that all values in an actor's store are reset on completion of the periodic cycle. The LTL formula is generated from actor code, and ensures that all store variables are at their initial value when the initial FSM state is returned to. In the following example the actor's store is $\{x = 0\}$, which can be extracted directly from the programmer's code *e.g.* `int x:=0;`, and the actor's initial FSM state is `s0`, which the programmer specifies in the FSM declaration (Section 2.3):

```
property cyclic_store is
    [] (State(s0) and Value(x=0) => <> State(s0) and Value(x=0))
```

4.2.2. Periodic state transition sequence

The LTL property to verify the existence of a periodic cycle in an actor's state transition combines observations of control states with the *until* temporal property. Consider an actor with states $s0$, $s1$, $s2$ and $s3$, and a periodic cycle $s0 \rightarrow s1 \rightarrow s3 \rightarrow s2 \rightarrow s0$. The property for proving that this is the sole possible sequence, and that it infinitely occurs is:

```
/* formula for proving the periodic cycle */
property periodic_sequence is
     s0 and [] (To(s0,s1) or To(s1,s3) or To(s3,s2) or To(s2,s0))
```

where, for any two Fiacre states `i` and `j`, `To(i,j)` stands for the formula:

```
State(i) and (State(i) until State(j))
```

At some position along a run, `To(si,sj)` holds if the Fiacre state of the actor at that position is `si`, the actor is in Fiacre state `sj` at some latter position in the run, and it remains in state `si` until then. Property `periodic_sequence` asserts that, in each run, the initial state of the actor is `s0` and that at each position in the run exactly one of the `To(si,sj)` propositions in the disjunction holds. The absence of counterexamples proves an infinitely periodic FSM cycle for this actor. In practice, extracting this LTL formula for automated verification will require a CAL language extension for expressing explicit cyclic sequences that the programmer intends the model checker to verify.

4.2.3. Cyclo-Static periodic data rate

Finally, we check for the same static data rate for every loop through the verified periodic cycle with:

```
property static_production_rate is
  [] (Value(P_static=false) => <> [] Value(P_static=true)))
property static_consumption_rate is
  [] (Value(C_static=false) => <> [] Value(C_static=true))
property static_rate is
   static_consumption_rate and static_production_rate
```

The LTL *static_production_rate* property reads: at each state along each run, either `P_static` is *true* or it eventually becomes continuously *true*. A static consumption rate is checked with the *static_consumption_rate* property, which tests for the same temporal observations for cyclic consumption rates.

Whilst a simple mechanism, the `P_static` and `C_static` booleans enable the model checker to search for a single action firing sequence counterexample whereby the `P` and `C` data rate counters deviate from a fixed value after an initial periodic cycle. Should a counter example not exist, *i.e.* an actor is cyclo static, these two values are used by the parallelisation algorithm in Section 5.2.

4.3. Model checking an actor

Processes cannot be model checked against the LTL rules above without communicating with another process that feeds data to it and consumes its results. Therefore the following testbench is composed in parallel with the actor being model checked:

```
process testbench [in1:in fifo, out1:out fifo] is
  states idle
  var consumed : fifo,
      q : fifo := fillBuffer()
  from idle
    select
       out1!q; to idle
    [] in1?consumed; to idle
    end
```

Their parallel composition is expressed in a top level Fiacre component:

```
component network is
     port in1 : fifo,
          out1 : fifo
     par * in
        actor_under_test[in1,out1]
     || testbench[out1,in1]
     end
```

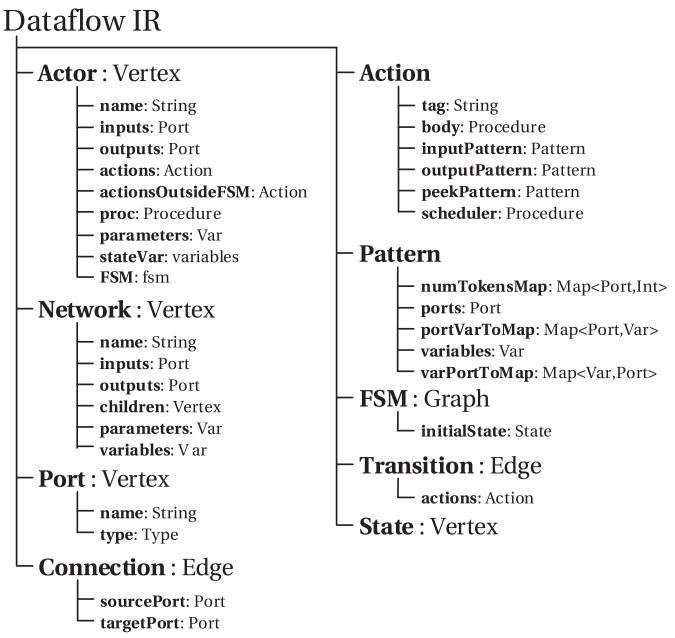


Fig. 9. Orcc IR for Dataflow Transformation (from [38] with permission).

5. Dataflow transformations

Our dataflow transformation system supports the parallelisation of static and cyclostatic actors. It is implemented using the Java based Orcc Intermediate Representation (IR) API, shown in Fig. 9, by instantiating the *Actor* and *Connection* classes to implement the parallelisation algorithm in Section 5.2.

5.1. Programming environment

We have extended the open source Orcc dataflow programming environment ([16]) with a refactoring tool, shown in Fig. 10, for parallelising static and cyclo-static actors which introduces *fork* and *join* actors to scatter/gather data. The user chooses the parallelism degree, *i.e.* how many actors to take the place of the previous single actor. Orcc has multiple backends for different target architectures. We have added

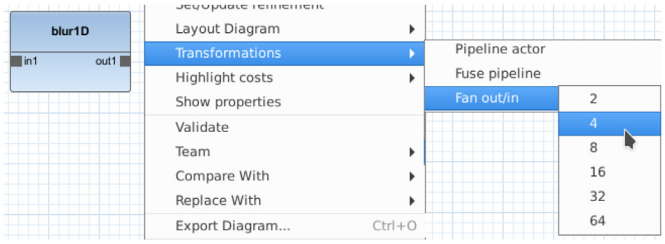


Fig. 10. Interactive Dataflow Transformation.

two additional backends to support the tool: 1) CAL, to generate new actor code for the fork and join actors and 2) Fiacre, for model checking actors. Fig. 11 shows the graphical consequence of applying the transformations.

5.2. Parallelisation algorithm

For an actor A consuming from edge u and producing to edge v, the transformation algorithm is:

1. Extract using model checking the consumption and production data rates $C_A^u(n)$ and $P_A^v(n)$ for a complete cyclostatic cycle, where n is the number of firings in a periodic cycle. This is done by parsing the P and C values from the output from the Tina model checker.
2. Create N parallel instances A_1 to A_N with the Orcc IR API, where N is the user-selected parallelism factor (Fig. 10).
3. Create a *fork* actor with Orcc IR that distributes data from edge u across $A_1, ., A_N$ in $C_A^u(n)$ chunks.
4. Create a *join* actor with Orcc IR to consume $P_A^v(n)$ chunks from actors A_1 to A_N.
5. Output the joined chunks as a sequential stream to edge v.

After this transformation the graph remains partially sequential due to the linear stream of data that the *fork* actor consumes then distributes. Fig. 12 shows the sequential nature of stream propagation as it broadcasts tokens to each parallel instance in sequence on an FPGA. In this case each parallel actor is receiving streams of 10 elements, 1 token per cycle. Here, the high *out1_SEND* signal instructs the first parallel actor that the data signal is valid for the first 10 cycles and the low signals *out2_SEND, out3_SEND* and *out4_SEND* instructs the other three actors to not read the data signal during these cycles. Stream gathering by the *join* actor is sequential also. This data propagation latency limits the speedup since some actors may be idle waiting for their next input stream, so there is a balance between the task size, potential parallel speedups and the latency overheads of parallelism.

6. Evaluation

This section evaluates the model checking based classification of actor models using three actors that implement (1) sparse matrix compression, (2) matrix row sorting and (3) dynamic time warp. One of these actors is identified as not being cyclostatic, the other two are, and hence are parallelisable. They are parallelised using the interactive transformation tool, replacing these sequential actors with up to 16 parallel actors. The speedup results are given in Section 6.3.

6.1. Counterexample

An actor can implement a sparse matrix compression algorithm incrementally as a matrix streams through an actor. It consumes the matrix in row major order, and outputs a *(row,column,value)* tuple for non-zero values. The transition rules of the actor are:

$$s0\langle compress \leftarrow \{w,h\}\rangle \xrightarrow[v!=0]{\langle v\rangle\mapsto\langle v,w,h\rangle} s0\langle \bullet \leftarrow \{w',h'\}\rangle$$

$$s0\langle ignore \leftarrow \{w,h\}\rangle \xrightarrow[v==0]{\langle v\rangle\mapsto\langle\rangle} s0\langle \bullet \leftarrow \{w',h'\}\rangle$$

The actor has one control state $s0$ and two actions *compress* and *ignore* that both transition from $s0$ back to $s0$. Each matrix value in the stream disambiguates action scheduling: *compress* fires on non-zero values, and *ignore* fires on zero values. The *compress* action increments width and height counters, as w' and h' in the store after the transition.

The actor is 25 lines of CAL code. The Fiacre description is 79 lines of code. The TINA model checker generates a synchronised Büchi automaton and a transition system from the LTL formula in Section 4, then looks for paths through the transition system for counterexamples. The model checker finds a counterexample for the *static_data_rate* property. State 60 in the Büchi automaton is:

```
60 : feed_1_sidle sparseArrayCompressor_1_ssInit
{ sparseArrayCompressor_1_vw=4,
  sparseArrayCompressor_1_vh=0,
  sparseArrayCompressor_1_vC=1,
  sparseArrayCompressor_1_vC_prev=1,
  sparseArrayCompressor_1_vP=1,
  sparseArrayCompressor_1_vP_prev=1,
  sparseArrayCompressor_1_vP_static=false,
  sparseArrayCompressor_1_vC_static=true,
  sparseArrayCompressor_1_vtokens={|2|} }
```

The model checker finds a transitional path $60 \rightarrow 76 \rightarrow 88 \rightarrow 97 \rightarrow 109 \rightarrow 115 \rightarrow 127 \rightarrow 47 \rightarrow 59 \rightarrow 76$. There is a loop from state 76 to itself. In this loop is state 97, which in the counter example is:

```
97 : feed_1_sidle sparseArrayCompressor_1_ss0
{ sparseArrayCompressor_1_vw=0,
  sparseArrayCompressor_1_vh=1,
  sparseArrayCompressor_1_vC=1,
  sparseArrayCompressor_1_vC_prev=1,
  sparseArrayCompressor_1_vP=0,
  sparseArrayCompressor_1_vP_prev=0,
  sparseArrayCompressor_1_vP_static=false,
  sparseArrayCompressor_1_vC_static=true,
  sparseArrayCompressor_1_vtokens={|0|} }
```

Therefore the model checker proves *static_production_rate* property false, since `P_static` is *false* at state 60 serves as the antecedent of the $\Rightarrow$ implication, and the consequent:

```
<> [] Value(P_static=true)
```

does not hold, because state 97 is always returned to. This is due to the nature of sparse arrays, number of non-zero values in the matrix determines the data rates.

6.2. Parallelising verified cyclo-static actors

Two benchmarks are now used to show the performance improvements of parallelisation of verified cyclostatic actors with the mechanised algorithm from Section 5.2.

6.2.1. Matrix row sorting

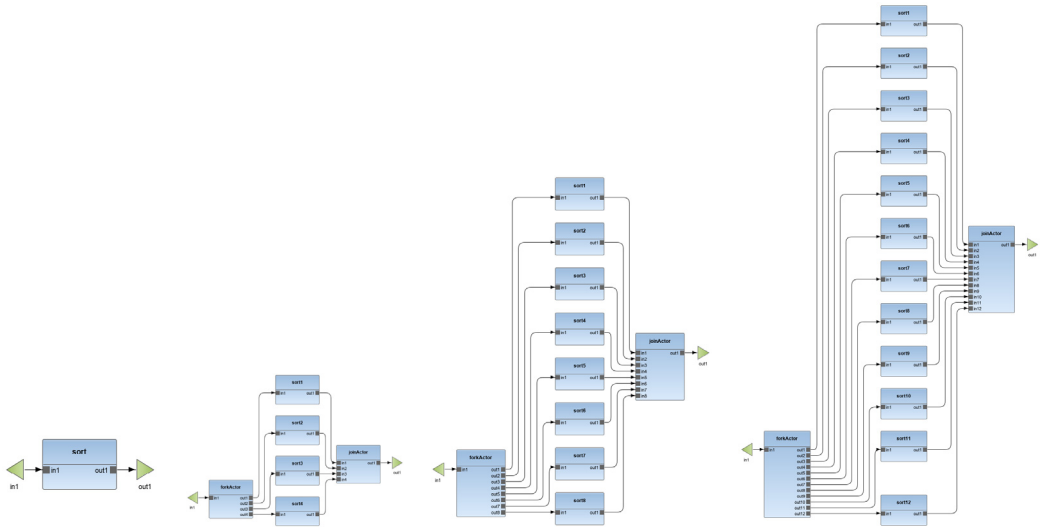
The matrix row sorting actor sorts every row in a matrix in ascending numerical order using bubble sort. The pre-sorted nature of the input determines the clock cycle latency to perform the sorting. There is also the latency of consuming and producing rows, one cycle per element. The actor's FSM is in Fig. 13. With an initial store $\{row = [], i = 0\}$, the scheduling is:

$$s0\langle receive \leftarrow \{elems,i\}\rangle \xrightarrow[i<width]{\langle a\rangle\mapsto\langle\rangle} s0\langle \bullet \leftarrow \{row+\{a\},i+1\}\rangle$$

$$s0\langle sort \leftarrow \{elems,i\}\rangle \xrightarrow[i==width]{\langle\rangle\mapsto\langle\rangle} s1\langle \bullet \leftarrow \{\{row'\},0\}\rangle$$

$$s1\langle output \leftarrow \{(e : elems'),i\}\rangle \xrightarrow[i<width]{\langle\rangle\mapsto\langle e\rangle} s1\langle \bullet \leftarrow \{\{elems'\},i+1\}\rangle$$

$$s1\langle reset \leftarrow \{\{\},i\}\rangle \xrightarrow[i==width]{\langle\rangle\mapsto\langle\rangle} s0\langle \bullet \leftarrow \{\{\},0\}\rangle$$



(a) Sequential (b) 4 actors (c) 8 actors (d) 12 actors

Fig. 11. Parallelising an Actor with the Dataflow Transformation Tool.

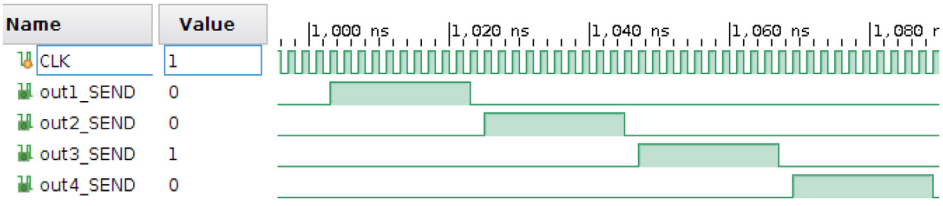


Fig. 12. Scattering the Data Stream from the *fork* Actor.

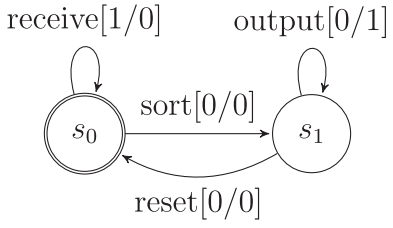


Fig. 13. FSM of the Matrix Row Sort Actor.

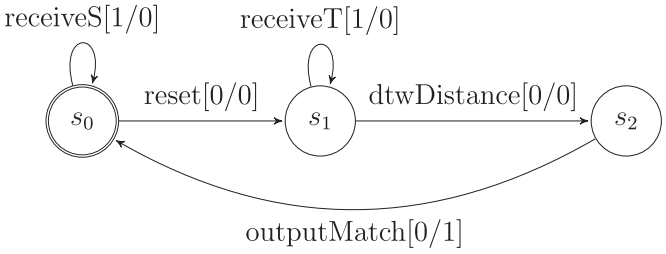


Fig. 14. FSM of the Dynamic Time Warp Actor.

The model checker is unable to find counterexamples of the LTL properties. For a matrix of width 100, the sorting actor A has a periodic cycle of 202 firings: 100× `receive`, 100× `output`, 1× `sort` and 1×`reset`. The fork actor scatters $C_A^v(202) = 100$ tokens to each parallel actor and the join actor gathers $P_A^u(202) = 100$ tokens from each actor.

6.2.2. Dynamic time warp

Dynamic time warping (DTW) is an algorithm to measure similarity of two temporal sequences. Applications of DTW include automated speech recognition. The actor takes two sequences S and T, then outputs the optimal match between them. Consuming each sequence element costs one cycle, followed by multiple clock cycles to compute the optimal match, then one cycle to output that match. The actor's FSM is in Fig. 14. With an initial store $\{s = [], t = [], dwt = [][], i = 0\}$ and a sequence length n, the scheduling is:

$$s0\langle receiveS \leftarrow \{s,t,dwt,i\}\rangle \xrightarrow[i<width]{\langle a\rangle\mapsto\langle\rangle} s0\langle \bullet \leftarrow \{s+\{a\},t,dwt,i+1\}\rangle$$

$$s0\langle reset \leftarrow \{s,t,dwt,i\}\rangle \xrightarrow[i==width]{\langle\rangle\mapsto\langle\rangle} s1\langle \bullet \leftarrow \{s,t,dwt,0\}\rangle$$

$$s1\langle receiveT \leftarrow \{s,t,dwt,i\}\rangle \xrightarrow[i<width]{\langle a\rangle\mapsto\langle\rangle} s1\langle \bullet \leftarrow \{s,t+\{a\},dwt,i+1\}\rangle$$

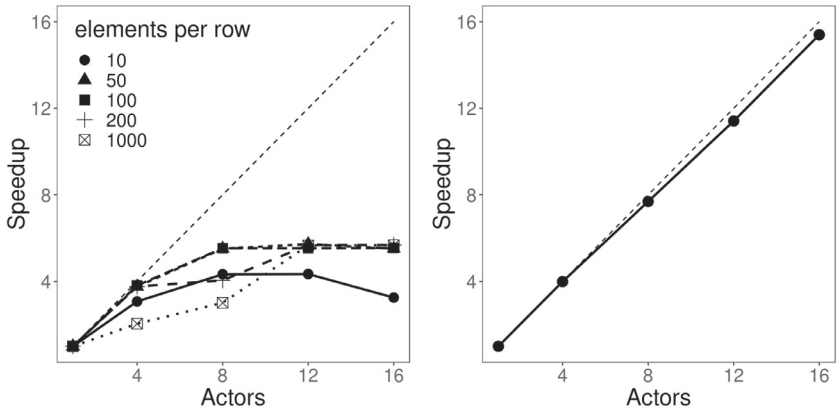
$$s1\langle dwtDistance \leftarrow \{s,t,dwt,i\}\rangle \xrightarrow[i==width]{\langle\rangle\mapsto\langle\rangle} s2\langle \bullet \leftarrow \{s,t,dwt,0\}\rangle$$

$$s2\langle dwtDistance \leftarrow \{s,t,dwt,i\}\rangle \xrightarrow{\langle\rangle\mapsto\langle dwt[n][n]\rangle} s0\langle \bullet \leftarrow \{[],[],[][],0\}\rangle$$

Again, the model checker is unable to find counterexamples of the LTL properties. For input sequence lengths of 40, the DTW actor A has a periodic cycle of 83 firings: 40× `receiveS`, 40×`receiveT`, 1× `reset`, 1×`dwtDistance` and 1× `outputMatch`. The fork actor scatters $C_A^v(83) = 80$ tokens to each parallel actor and the join actor gathers $P_A^u(83) = 1$ token from each actor.

6.3. Performance results

For the two benchmarks, the Orcc compiler generates an FPGA circuit description using the Xronos Verilog backend ([39]). The results in Fig. 15 show the parallel speedup using 4, 8, 12 and 16 actors. Speedup is T_1/T_n, where T_1 is the clock cycles with one actor and T_n is the cycles with n actors. The dashed line shows ideal linear speedup. The Verilog simulation testbench randomises the input values for 128 input sets for



(a) Matrix Row Sorting (b) Dynamic Time Warp

Fig. 15. Speedups after Parallel Actor Transformation.

both benchmarks, *i.e.* when using 16 actors each actor processes 8 inputs.

Fig. 15a shows matrix row sorting speedups for different row lengths. The number of elements in a matrix row, and the pre-sorted arrangement of its values, determines the clock cycle latency of sorting the elements since randomly ordered inputs require repeated sorting. The sequential algorithm for a row length of 1000 requires 1,546,477 clock cycles. There is a 5.6× speedup for 12 and 16 actors when processing rows of lengths between 10 and 200. From a speedup of 4.3 with 12 actors there is a drop in performance to a 3.3 speedup with 16 actors to sort just 10 elements. At such small workloads, most of the 16 actors will not be firing and instead the data propagation latency overheads result in degraded performance.

Fig. 15b shows dynamic time warping speedups for a length of 40 for sequences *S* and *T*. The sequential algorithm requires 1,332,979 clock cycles. Speedup with 16 actors is almost linear (15.4).

Speedups of 5.6× and 15.4× with our approach take significantly less time to achieve compared to using direct Verilog or VHDL optimisations. This is because our tooling automates fork/join data management and verifies the optimisation can be safely applied with model checking, which if done at the Verilog/VHDL level would require extensive testbench simulation.

The CAL and Fiacre implementations, generated hardware descriptions and raw results are in an Open Access dataset [40].

7. Discussion

The previous section demonstrated how parallelising actors can elicit speedups. However this is not always the case, *e.g.* when an actor is not on the critical path of a program's execution [41]. Worse still, rather than increasing performance it can have a countereffect of generating larger programs, which in turn can reduce achievable clock frequencies or exceed hardware resource constraints. Tooling can help direct a programmer to bottlenecks, which parallelism may alleviate.

In Section 7.1 we present a tool we have developed for Orcc that addresses this issue. The motivation is to present hardware costs to the user, *e.g.* identifying the actor with the longest datapath, which in turn affects the overall achievable clock speed for the entire dataflow graph. Then in Section 7.2 we quantify the CAL language features used in real world code. The use of some CAL language features potentially results in dynamic dataflow behaviours. The frequent occurrence of these language constructs in each actor justifies the need for model checking.

7.1. Hardware profiling

The Xronos FPGA backend for Orcc uses Xilinx's open source OpenForge [42] compiler to generate hardware cost reports for each actor at compile time. These are: 1) the number of BRAM blocks, and 2) the datapath depth for each action in an actor. The datapath depth determines the minimum latency (microseconds) between each clock cycle.

We have added to the Orcc programming environment a tool that lifts these hardware costs into the visual dataflow editor (Fig. 16), to communicate hardware bottlenecks to the user. Green actors have small datapath depths, relative to the orange critical actor(s) which have the longest datapath. BRAM counts for implementing actor internal variables are also shown to the programmer.

Table 1
Actor properties.

Property	Occurrence
Guard on input token value	15% of actions
Guard on store value	56% of actions
Shared input ports	49% of input ports
Shared output ports	41% of output ports
Actors with a store	78% of actors
Stores modified by multiple actions	62% of store variables

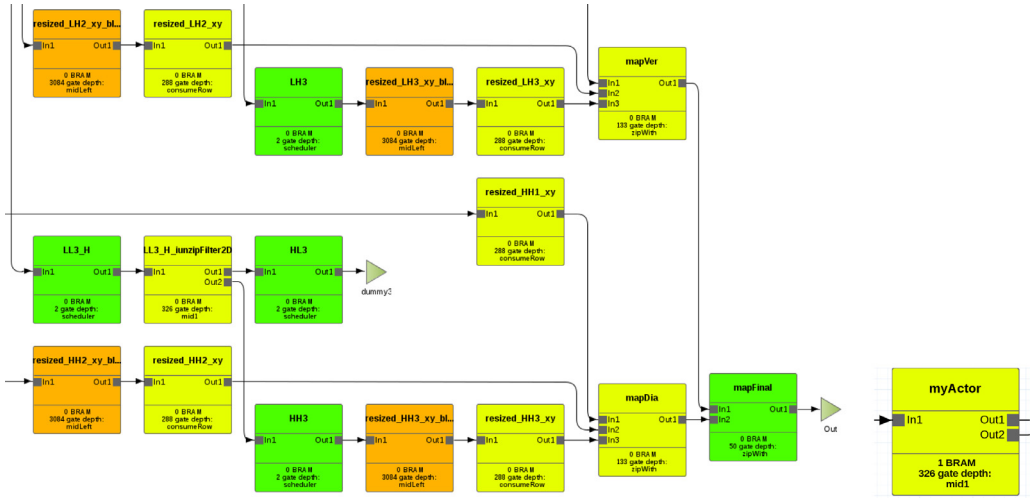
7.2. Real world dynamic dataflow programs

A static analysis approach with small actors may be able to identify static, cyclostatic and dynamic actors, *e.g.* an actor with just one action. However scaling static analysis to identify models of computation of larger actors, *e.g.* multiple guarded actions and store variables updated by multiple actions, would be very challenging and the authors are not aware of a dataflow framework that can do this automatically. That is, applying static analysis on actors implemented in a dynamic dataflow language to classify their dataflow model of computation.

We assess dataflow language features used in a real world code repository[2] to evaluate the size of CAL actors in practise. It includes 823 actors in 555 dataflow graphs written by 23 developers, across multiple domains including cryptography, image processing and network protocols.

Table 1 shows the frequency of CAL language features in this repository that can impact on the complexity of classifying an actor's model of computation, *e.g.* whether it has cyclostatic or dynamic data rates. An external control signal from an input port is in a guard predicate for 15% of actions, and internal control signals in the store appear in a guard predicate for 56% of actions, *i.e.* whose private variable values may determine runtime scheduling choices. Most actors are stateful, with 78% containing at least one variable in its store. Multiple actions consume

[2] https://github.com/orcc/orc-apps.



(a) Graph hardware critical path (b) one actor

Fig. 16. Visualising FPGA Resource Costs.

the same port for 49% of all input ports. Multiple actions write to the same port for 41% of all output ports. Multiple actions modify store variables for 62% of all store variables.

Static analysis would not be feasible at these scales of code complexity. Machine checked verification would be the only feasible option to overcome this, *e.g.* with the model checking approach that this paper presents.

Automating the verification and parallelisation approach for the full CAL language will require some additional engineering: (1) the implementation of a parser for the TINA model checking output, (2) an extension of the verification approach to support multiple input or output ports, and (3) a CAL language extension for explicit cyclic sequences to be expressed (Section 4.2.2). Supporting multiple actor ports (task 2) will likely be the most trivial, because there is a direct mapping from CAL ports to Fiacre ports (Section 3.3.1). Extending the syntax of CAL FSMs to support a programmer's claim of a deterministic cyclic sequence through states (task 3), for model checking to verify, will require an extension to Orcc's frontend *i.e.* changes to the Xtext grammar, and adding it to Orcc IR by extending the Orcc EMF (Eclipse Modelling Framework) model. Processing the output of the TINA model checker (task 1) will a require a parser, or a machine readable output format added to TINA's backend.

8. Conclusion

As the performance of embedded architectures increases, program sizes that they can support also grow, as does the complexity and dynamic nature of algorithms in programs. As embedded accelerators become more widely adopted, the importance of effective code optimisations grows. Previous work ([43]) explores the NP-hard problem of cyclic scheduling of multiple actors mapped to parallel processing architectures. In contrast, this paper presents a program transformation approach that parallelises a single, potentially stateful, actor into multiple actors to exploit parallel architectures. The approach separates cyclo-static from dynamic actors using LTL model checking. The verification phase takes seconds to run. Th graphical parallel refactoring tool speedily increases FPGA performance, *e.g.* a 15.4× speedup with 16 actors.

This paper is a step towards auto-parallelisation of dynamic dataflow programs. Verifying other parallelising transformations, *e.g.* task and pipelined parallelism, and verifying transformation of dynamic actors, would extend our approach. The broader aim of this work is to integrate automated formal verification more widely into optimising compilers and parallel runtime systems for embedded systems.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

We acknowledge the support of the Engineering and Physical Research Council grant references EP/K009931/1 (Programmable embedded platforms for remote and compute intensive image processing applications), EP/N014758/1 (The Integration and Interaction of Multiple Mathematical Reasoning Processes) and EP/N028201/1 (Border Patrol: Improving Smart Device Security through Type-Aware Systems Design), and the Scottish Funding Council for a SICSA Postdoctoral and Early Career Researcher Exchanges grant.

References

[1] D. Bhowmik, P. Garcia, A.M. Wallace, R. Stewart, G. Michaelson, Power efficient dataflow design for a heterogeneous smart camera architecture, in: 2017 Conference on Design and Architectures for Signal and Image Processing, DASIP 2017, September 27–29, 2017, IEEE, Dresden, Germany, 2017, pp. 1–6.
[2] R. Stewart, K. Duncan, G. Michaelson, P. Garcia, D. Bhowmik, A. Wallace, RIPL: A Parallel Image processing language for FPGAs, TRETS 11 (1) (2018) 7:1–7:24.
[3] E.A. Lee, D.G. Messerschmitt, Static scheduling of synchronous data flow programs for digital signal processing, IEEE Trans. Comput. 36 (1) (1987) 24–35.
[4] C. Brown, H. Loidl, K. Hammond, ParaForming: forming parallel haskell programs using novel refactoring techniques, in: TFP 2011, May 16–18, 2011, Revised Selected Papers, Springer, Madrid, Spain, 2011, pp. 82–97.
[5] C. Brown, M. Danelutto, K. Hammond, P. Kilpatrick, A. Elliott, Cost-Directed refactoring for parallel erlang programs, Int. J. Parallel Program. 42 (4) (2014) 564–582.
[6] C. Brown, K. Hammond, M. Danelutto, P. Kilpatrick, H. Schöner, T. Breddin, Paraphrasing: generating parallel programs using refactoring, in: FMCO 2011, October 3–5, 2011, Revised Selected Papers, Springer, Turin, Italy, 2011, pp. 237–256.
[7] G. Grov, G. Michaelson, Hume box calculus: robust system development through software transformation, Higher-Order Symbol. Comput. 23 (2) (2010) 191–226.
[8] K. Hammond, G. Michaelson, The design of Hume: a high-level language for the real-time embedded systems domain, in: International Seminar on Domain-Specific Program Generation, Dagstuhl Castle, Germany, March 23–28, 2003, Springer, 2003, pp. 127–142.
[9] L. Lamport, The temporal logic of actions, ACM Trans. Programm. Lang.Syst. 16 (3) (1994) 872–923.
[10] G. Grov, Reasoning About Correctness Properties of a Coordination Programming Language, Heriot-Watt University, Edinburgh, UK, 2009 Ph.D. thesis.

[11] W. Thies, M. Karczmarek, S.P. Amarasinghe, StreamIt: a language for streaming applications, in: Proceedings of the 11th International Conference on Compiler Construction, Springer-Verlag, London, UK, UK, 2002, pp. 179–196.
[12] M. Geilen, T. Basten, S. Stuijk, Minimising buffer requirements of synchronous dataflow graphs with model checking, in: DAC 2005, San Diego, CA, USA, June 13–17, 2005, ACM, 2005, pp. 819–824.
[13] X. Zhu, R. Yan, Y. Gu, J. Zhang, W. Zhang, G. Zhang, Static optimal scheduling for synchronous data flow graphs with model checking, in: FM 2015, Oslo, Norway, June 24–26, 2015, Springer, 2015, pp. 551–569.
[14] I. Grobelna, Model checking of reconfigurable FPGA modules specified by petri nets, J. Syst. Archit. - Embed. Syst.Des. 89 (2018) 1–9.
[15] J. Eker, J.W. Janneck, CAL Language Report Specification of the CAL Actor Language, Technical Report, EECS Department, University of California, Berkeley, 2003.
[16] H. Yviquel, A. Lorence, K. Jerbi, G. Cocherel, A. Sanchez, M. Raulet, Orcc: multimedia development made easy, in: ACM Multimedia Conference, October 21–25, 2013, ACM, Barcelona, Spain, 2013, pp. 863–866.
[17] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, Y. Zhou, Cilk: an efficient multithreaded runtime system, J. Parallel Distrib. Comput. 37 (1) (1996) 55–69.
[18] L. Dagum, R. Menon, Openmp: an industry standard API for shared-memory programming, IEEE Comput. Sci. Eng. 5 (1) (1998) 46–55.
[19] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J.-F. Nezan, S. Aridhi, Preesm: A dataflow-based rapid prototyping framework for simplifying multicore DSP programming, in: EDERC 2014 6th European Embedded Design in, 2014, pp. 36–40.
[20] K. Hammond, G. Michaelson, Hume: a domain-specific language for real-time embedded systems, in: GPCE 2003, Erfurt, Germany, September 22–25, 2003, Proceedings, in: Lecture Notes in Computer Science, Springer, 2003, pp. 37–56.
[21] J. Sérot, F. Berry, S. Ahmed, CAPH: a language for implementing stream-processing applications on FPGAs, in: Embedded Systems Design with FPGAs, Springer New York, 2013, pp. 201–224.
[22] Synflow, The Cx programming language, 2015, (https://synflow.gitlab.io).
[23] R. Lauwereins, M. Engels, M. Adé, J.A. Peperstraete, Grape-II: a system-Level prototyping environment for DSP applications, IEEE Comput. 28 (2) (1995) 35–43.
[24] G. Bilsen, M. Engels, R. Lauwereins, J.A. Peperstraete, Cycle-static dataflow, IEEE Trans. Signal Process. 44 (2) (1996) 397–408.
[25] E.A. Lee, T.M. Parks, Dataflow process networks, Proc. IEEE 83 (5) (1995) 773–801.
[26] K. Schneider, J. Brandt, Quartz: a synchronous language for model-based design of reactive embedded systems, in: S. Ha, J. Teich (Eds.), Handbook of Hardware/Software Codesign, Springer Netherlands, Dordrecht, 2017, pp. 29–58.
[27] MATLAB, Simulink, 2018, (https://uk.mathworks.com/products/simulink.html).
[28] D.G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, M. Abadi, Naiad: a timely dataflow system, in: SOSP 2013, Farmington, PA, USA, November 3–6, 2013, ACM, 2013, pp. 439–455.
[29] J.W. Janneck, I.D. Miller, D.B. Parlour, G. Roquier, M. Wipliez, M. Raulet, Synthesizing hardware from dataflow programs - an MPEG-4 simple profile decoder case study, Signal Process. Syst. 63 (2) (2011) 241–249.
[30] R. Stewart, D. Bhowmik, A.M. Wallace, G. Michaelson, Profile guided dataflow transformation for FPGAs and CPUs, Signal Process. Syst. 87 (1) (2017) 3–20.
[31] J. Eker, J.W. Janneck, E.A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S.R. Sachs, Y. Xiong, Taming heterogeneity - the ptolemy approach, Proc. IEEE 91 (1) (2003) 127–144.
[32] J.W. Janneck, Actors and their composition, Formal Aspect. Comput. 15 (4) (2003) 349–369.
[33] B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, F. Vernadat, Fiacre: an intermediate language for model verification in the topcased environment, ERTS 2008, Toulouse, France, 2008.
[34] B. Berthomieu, P. Ribet, F. Vernadat, The tool TINA - Construction of abstract state spaces for petri nets and time petri nets, Int. J. Prod. Res. 42 (14) (2004) 2741–2756.
[35] B. Berthomieu, S. dal Zilio, F. Vernadat, A FIACRE V3.0 Primer, Technical Report, LAAS-CNRS Université de Toulouse, France, 2012. http://projects.laas.fr/fiacre/doc/primer.pdf
[36] B. Berthomieu, J.-P. Bodeveix, M. Filali, H. Garaval, F. Lang, D.L. Botlan, F. Vernadat, S. dal Zilio, The Syntax and Semantics of Fiacre, Technical Report, LAAS-CNRS Université de Toulouse, France, 2012.
[37] T. Murata, Petri nets: properties, analysis and applications., Proc. IEEE 77 (4) (1989) 541–580.
[38] E. Bezati, S.C. Brunet, M. Mattavelli, J.W. Janneck, High-level synthesis of dynamic dataflow programs on heterogeneous MPSoC platforms, in: International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, SAMOS 2016, July 17–21, IEEE, Agios Konstantinos, Samos Island, Greece, 2016, pp. 227–234.
[39] E. Bezati, High-Level Synthesis of Dataflow Programs for Heterogeneous Platforms: Design Flow Tools and Design Space Exploration, School of Engineering, Ecole Polytechnique Fédérale de Lausanne, Switzerland, 2015 Ph.D. thesis.
[40] R. Stewart, Open Access dataset for "Verifying Parallel Dataflow Transformations with Model Checking and its Application to FPGAs", 2019, https://doi.org/10.17861/85ff96b4-2c6b-4f58-8322-74f0ab45f684
[41] S.C. Brunet, M. Mattavelli, J.W. Janneck, Buffer optimization based on critical path analysis of a dataflow program design, in: 2013 IEEE International Symposium on Circuits and Systems (ISCAS2013), Beijing, China, May 19–23, 2013, IEEE, 2013, pp. 1384–1387.
[42] E. Bezati, H. Yviquel, M. Raulet, M. Mattavelli, A unified hardware/software Co-synthesis solution for signal processing systems, in: DASIP 2011, Tampere, Finland, November 2–4, 2011, IEEE, Tampere, Finland, 2011, pp. 186–191.
[43] C. Hanen, A. Munier, A study of the cyclic scheduling problem on parallel processors, Discrete Applied Mathematics 57 (2–3) (1995) 167–192.

Dr Robert Stewart is an Assistant Professor at Heriot-Watt University. His interests are at the interface between programming languages and computer architectures. They span parallel programming functional languages for multicore HPC and embeded architectures, dataflow models for embedded systems and hardware verification.

Dr Bernard Berthomieu has interests in semantics and implementation of concurrent programming languages. Bernard is the developer of the programming language LCS, based on a higher order variant of Robin Milner's Calculus of Communicating Systems embedded into Standard ML. Bernard is also interested in the analysis techniques for Petri Nets, Time Petri Nets, and related formalisms for concurrent systems. Bernard is the developer of the TINA model checking toolbox.

Dr Paulo Garcia is an Assistant Professor of Systems and Computer Engineering at Carleton University. Paulo received his BSc, MSc and PhD degrees from the University of Minho, in 2008, 2011 and 2015. His research interests include languages for hardware-software co-design, computer architectures, system software and hybrid CPU-FPGA systems design.

Dr Idris Skloul Ibrahim is an Assistant Professor at Heriot-Watt University. His interests are ad hoc network protocols, computer systems, computer algebra on Cloud infrastructures, mobile applications and parallel dataflow transformations.

Professor Greg Michaelson is a Professor of Computer Science at Heriot-Watt University. He has BSc Computer Science from the University of Essex, MSc Computational Science from the University of St Andrews and a PhD from Heriot-Watt University. His expertise is in the design, analysis and implementation of programming languages, in particular functional languages for multi-processor platforms. He is a Fellow of the British Computer Society.

Professor Andrew Wallace is a Professor of Signal and Image Processing at Heriot-Watt University. Andrew received his BSc and PhD degrees from the University of Edinburgh in 1972 and 1975, with particular interests in LiDAR signal processing, video analytics and parallel, many core architectures. He has published extensively, receiving a number of best paper and other awards. He has secured funding from EPSRC, the EU and other industrial and government sponsors. He is a chartered engineer and a Fellow of the Institute of Engineering Technology.