Foundations for Verifiable Reactive Systems on Exposed Datapath Architectures with Buffered Processing Units

Markus Anders

September 27, 2018

Technical Report

University of Kaiserslautern
Department of Computer Science
Embedded Systems Group

Contents

1 Introduction **5**
1.1 Motivation . . . 5
1.2 Main Contributions . . . 7
1.3 Summary . . . 8

2 Preliminaries **9**
2.1 SCAD Architectures . . . 9
2.2 Dataflow Process Networks . . . 10

3 Operational Semantics for SCAD Architectures **13**
3.1 Formal Description . . . 13
3.2 Operational Semantics . . . 13

4 Deterministic SCAD Architectures **18**
4.1 DPN Model for SCAD Architectures . . . 18
4.2 Weak Bisimulation . . . 22
4.3 Conclusions from Kahn Criteria . . . 27
4.4 Transformation of DPNs to SCAD Architectures . . . 28

5 Periodic Branchless Move Programs **30**
5.1 Decision Procedure for Boundedness . . . 31
5.2 Symbolic Period Summary . . . 38
5.3 Transformation to Finite State Machines . . . 41

6 Bounded Move Programs **42**
6.1 Configuration Layouts . . . 42
6.2 Towards Analysis of Bounded Move Programs . . . 43

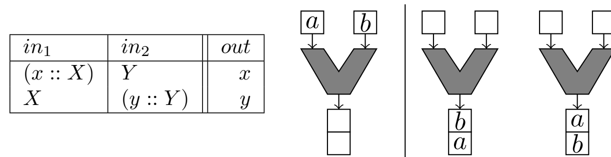
7 Conclusions, Applications and Future Work **47**
7.1 DPNs and SCAD Architectures . . . 47
7.2 Tool Proposals . . . 47
7.3 Open Problems . . . 48

1 Introduction

1.1 Motivation

Recent years have seen the emergence of exposed datapath architectures [1–4]. They expose all of their processing units and datapaths to the compiler. By using this design philosophy, the compiler may take full advantage of the available hardware. Compilers can thereby often avoid the use of a central register file, which helps exploiting more instruction-level parallelism (ILP). Furthermore, among other issues, sending all values through a central register file is leading to it being a common hot spot [5]. And while using several cores enables current processors to leverage thread-level parallelism (TLP) explicitly expressed by the programmer, this approach can not fully exploit existing parallelism on the instruction-level present in many applications [1]. Transport triggered architectures (TTA) [4] are a well-known class of exposed datapath architectures that only use a single type of instruction. This instruction moves values from one register inside the processor to another. These may not only be traditional registers inside a register file, but also input or output registers of processing units. The arrival of data then triggers the operations. Results can simply be transported away after some time using later instructions. Because data can be transported directly from output registers back to input registers, the compiler can often avoid the central register file when passing intermediate results. This approach gives great versatility without the need to change the instruction set. In other architectures, when adding a new processing unit or register, a change in the instruction set often becomes necessary, which can be disruptive to all aspects of a computer system [3]. Clearly, TTA utilizes static scheduling. Other benefits will therefore include better power efficiency and execution time analysis when compared to dynamically scheduled architectures. The synchronous control asynchronous dataflow (SCAD) architectures expand upon this idea by changing the registers of processing units to FIFO buffers.

There are several reasons why SCAD architectures can be conjectured to be well suited for hard real-time systems and reliable systems in general: First, processing units may have arbitrary delay times and can implement any circuit without fundamentally changing the instruction set or compilation tools. The FIFO buffers can be used to abstract away from precise timings. This can give great versatility when partly implementing a system as an application-specific integrated circuit (ASIC) is desired. Clearly, this can be achieved without loosing the capabilities of a general purpose processor. Secondly, worst-case execution time (WCET) analysis will be comparatively simple even when a lot of instruction level parallelism (ILP) is exploited [6]. This is due to ILP being utilized explicitly in SCAD programs without the need of dynamically scheduled execution. Furthermore, in [6] it was shown that this approach enables the utilization of a substantial amount of ILP. SCAD also maintains the benefits of most statically scheduled architectures in power-restricted environments. And thirdly, memory calls can be greatly reduced when using the same amount of wiring as register architectures, albeit for a severe runtime penalty [7]. This



in_1	in_2	out
$(x :: X)$	Y	x
X	$(y :: Y)$	y

Figure 1: A SCAD PU with its firing rules (a) may have several different results (b) depending on the run.

is still very desirable, since caches are known to be troublesome for real-time systems [8]. Reducing the total amount of memory calls can therefore have a tremendous effect on the WCET. Furthermore, good compilation can reduce or even eliminate the runtime penalty [6, 9]. But actually enabling the development of reliable systems on SCAD architectures will take more than the previously researched techniques. As these systems are often used in safety critical environments, verifiable correctness and safety are common requirements. Processing units can implement potentially any circuit and there are asynchronous behaviours in the execution of SCAD architectures themselves. Therefore, determinacy and correctness of the architecture has to be guaranteed first. Determinacy can not be guaranteed for all processing unit implementations and can thus not be taken for granted. Figure 1 shows that the given firing rule and initial configuration (a) may lead to different results (b). The operational semantics will not determine which of the rules should fire: the PU may first consume x and produce it to out, or y. Both rules are active in the initial configuration. Although this kind of behaviour might be desired in some cases, this paper will focus on finding PU implementations that have a functional behaviour for all given programs and inputs. This motivation is also not limited to hard real-time systems at all: in most applications, having provably deterministic behaviour of the underlying processor architecture is essential. And this paper will determine a large class of PU functions, for which deterministic behaviour is given. But furthermore, correctness of machine programs - even if they are the result of a model-based design flow where intermediate models are verified - may be required to be proven correct. The translation into SCAD machine code may not be trusted. Especially since machine code for SCAD turned out to be very difficult to read and reason about. Consider for example the PU firing rule, program and initial state given in Figure 2. Even for a simple program like this, it is already very tricky to see what it does at a glance. Therefore, an attempt will be made to provide a translation for SCAD programs to make them more readable and therefore trustworthy. The example can then be translated to $out_1 = ((a+c)+b)$. The method will use symbolic execution of the operational semantics, which is trivial in the example given - but also more difficult programs will be discussed. A formal verification methodology for some SCAD architectures and programs will also be outlined in this paper, and the limitations of this approach outlined. As the determinism proof will provide

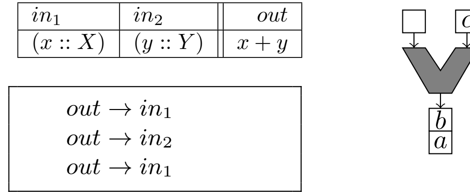


Figure 2: A PU firing rule, a program and an initial configuration

a translation to dataflow process networks (DPNs), the semantics of them can also be used for SCAD programs.

To summarize, it is clear that a sound theoretic foundation is necessary to enable the development of reliable systems on SCAD architectures. This paper aims at exploring some of these foundations.

1.2 Main Contributions

The contributions of this paper are mostly split in two parts. The first part (Section 3 and Section 4) will provide operational semantics and determinacy of SCAD architectures. It therefore mainly deals with the soundness of a large class of SCAD architectures. The second part (Section 5 and Section 6) will consider designing reliable systems on these sound SCAD architectures. First, simple programs that can easily be model-checked are shown. Then, more complicated programs are introduced. For those, it will mostly be considered how they can be made more readable. The contributions of this paper include:

- A framework of formal descriptions and operational semantics for general SCAD architectures in Section 3.
- A translation of SCAD architectures to DPNs with minor restrictions in Section 4.
- Using this translation to proof determinacy of SCAD architectures restricted to Kahn processing units in Section 4.
- A definition of periodic branchless move programs in Section 5.
- Providing a methodology to decide boundedness and model-check periodic branchless move programs in Section 5.
- A method to translate periodic branchless move programs into expression systems in Section 5.
- Definition of SCAD configuration layout-equivalence in Section 6.
- A proposal to translate bounded SCAD programs to `cmd` programs in Section 6.

1.3 Summary

First, in Section 2, preliminaries are discussed: SCAD architectures are informally described and the definition and operational semantics of DPNs is given. In Section 3, formal definitions and operational semantics for SCAD architectures are then provided. Then, in Section 4, the bisimulation of SCAD architectures and a special subset of DPNs is described and thereby determinacy of Kahn SCAD architectures proven. Section 5 introduces periodic branchless move programs and the aforementioned results and methods surrounding them. Building on that, Section 6 will explore how to handle more complicated move programs. Finally, Section 7 will give some conclusions, tool proposals and open problems.

2 Preliminaries

A few preliminaries must be provided before the main contributions of this paper can be given. First, an informal description of SCAD machines is given. Then, data flow process networks (DPNs) including their operational semantics are described. Finally, some results surrounding DPNs are cited.

2.1 SCAD Architectures

Based on [6, 7, 9], a SCAD machine consists of a control unit (CU) and processing units (PUs). These are connected by a synchronous move-instruction bus (MIB) and an asynchronous data transport network (DTN). SCAD is solely programmed by move instructions that transport values between PUs. The MIB is used to send move instructions from the control unit to the processing units while the DTN is used to transport values between the processing units. Each PU – shown in orange color in Figure 3 – can have several input and output ports with unique addresses (for simplicity, only one output two input PUs are shown in Figure 3). Every port has a FIFO (first-in first-out) buffer to store the values arriving at that port. A PU reads values from its input buffers and writes the computed results to its output buffers. A buffer may have several entries, each of which is a pair (adr, val). For input buffers, adr is the address of that output buffer that produces value val. If val has not yet been transported, the special symbol $\bot$ is used to denote this. For output buffers, adr denotes that input buffer to which val will be sent. $\bot$ indicates a value that has not yet been produced. In Figure 3, the red cells indicate adr fields and green cells are val fields. Note that values in output buffers are produced by their corresponding PUs, while values in input buffers are those sent via the DTN. A precise definition of SCAD operational semantics is given in Section 3. But to summarize the operational firing rules that will be given, here is a quick informal overview: programs in SCAD only consist of move instructions $src \to tgt$, where src is an output buffer and tgt is an input buffer. The `ADVANCE` rule adds such an instruction to the adr part of the FIFO buffers. tgt is enqueued in the adr part of src, and src in the adr part of tgt. `ADVANCE` can however only fire if the CU is not stalling. The `STALL` rule works similarly, only that $tgt = \mathtt{c}$. $\mathtt{c}$ denotes the input buffer of the CU. The CU reads move instructions sequentially according to a program counter. Sending a value to $\mathtt{c}$ means setting the program counter to a specific value, similarly to a typical `goto` instruction. Therefore, once this instruction is reached, the CU waits for the arrival of the value (which is asynchronous from adding the instruction itself). Therefore, the CU will stall from this point on. The CU can however continue working using the `CONTINUE` rule. Once a value arrives in $\mathtt{c}$, and therefore the CU, the program counter is set to the arriving value and the CU will continue fetching instructions. The asynchronous data transports on the DTN are described using the `DATAFLOW` rule. If the head of an output buffer has a value in val and an address in adr, the data can be transported to the input buffer tgt denoted in adr. It will be sent to the field in tgt where $adr = src$ and $val = \bot$ closest to the head of the

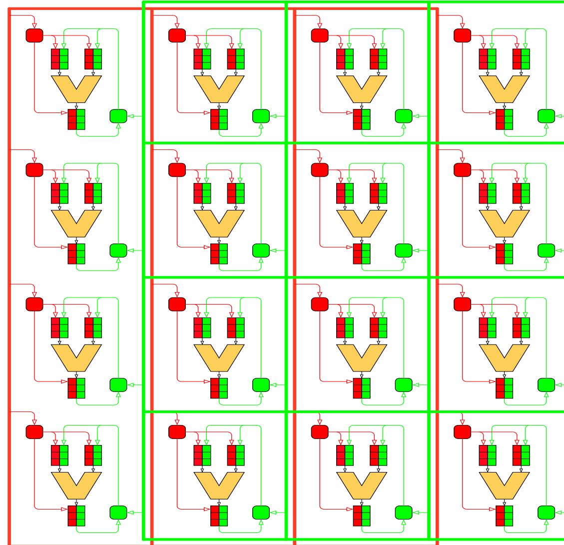


Figure 3: Basic SCAD Architecture

FIFO buffer. And last but not least PU operations consuming and producing values have to be performed. The rule for it is called `FIRE`. PU firings dequeue values from the heads of their input buffers, and enqueue them in their output buffers. This rule will mostly be reduced to the firing rule of Dataflow Process Networks described in the next section.

2.2 Dataflow Process Networks

Dataflow process networks (DPNs) as first described in [10] consist of processes that exclusively communicate via FIFO-buffers connecting them. It is assumed that these FIFO buffers may have unbounded size, implying that buffers can always be written to. A FIFO-buffer has at most one producing node that can write to it and at most one consuming node that can read from it. Processes can perform their computations independently from each other. The descriptions used are inspired by [11], which also provides a more detailed and complete version of the results used in this paper. The material given in this section should not be seen as a comprehensive definition of DPNs, but rather as a brief summary of the required results. The definition of operational semantics for SCAD architectures will also make direct use of these results.

The content of the buffers will be described using streams. The abbreviation $Stream_{\mathcal{D}}^{m} := (\mathcal{D}^{\infty})^{m}$ is used to denote a stream. A process node of a DPN reading from m inputs and writing to n outputs should therefore implement a function $Stream_{\mathcal{D}}^{m} \to Stream_{\mathcal{D}}^{n}$. The function may be described in terms of any programming language or by other means. In this paper, pattern matching

and firing rules will mostly be used to describe them. The state $\mathcal{S}$ of a DPN is simply a mapping from the buffer names to a respective $Stream^1_{\mathcal{D}}$. If a DPN has a set of buffers Z and data type $\mathcal{D}$, then $\mathcal{S} : Z \to Stream^1_{\mathcal{D}}$.

In order to describe the operational semantics of DPNs, some notation for stream expressions will be introduced. Stream expressions form a pattern that a stream will be matched to. Upper case variables will match with all streams. Lower case variables will denote variables matching with values from $\mathcal{D}$. Instead of lower case variables, values from $\mathcal{D}$ may also be used directly in stream expressions to match with themselves. $[\,]$ shall denote the empty stream. $(e :: \sigma)$ matches a stream with first element e followed by the stream expression σ'. The definition is summarized in Definition 1.

Definition 1. *Stream expressions.*

$$\sigma := \begin{cases} [\,] \\ S \\ (e :: \sigma) \end{cases}$$

A pattern is then a stream expression that is used to match with a stream. A stream expression σ matches with stream S if a substitution ϱ for the variables inside the stream expression exist such that $\varrho(\sigma) = S$. A firing rule of a node with input buffers $x_1, ..., x_n$ and output buffers $y_1, ..., y_n$ will have patterns $P_1, ..., P_m$ and finite sequences $\sigma_1, ..., \sigma_n$ for each input and output. We will write firing rules in tables, so if the node has p firing rules it will be written as follows:

x_1	...	x_m	y_1	...	y_n
$P_{1,1}$	...	$P_{1,m}$	$\sigma_{1,1}$	...	$\sigma_{1,n}$
...	...	...	...	...	...
$P_{p,1}$	...	$P_{p,m}$	$\sigma_{p,1}$	...	$\sigma_{p,n}$

Rule $i \in \{1, ..., p\}$ is called enabled in state $\mathcal{S}$ if each of the patterns $P_{i,1}$ ... $P_{i,m}$ match with their respective stream $\mathcal{S}(x_{i,j})$. Then, the node can fire using rule i. The rules may overlap and in that case, all matching rules can potentially fire, there are no precedence rules. Consumption of patterns is defined recursively on the pattern:

Definition 2. *DPN pattern consumption.*

$$cons(\sigma) := \begin{cases} 1 + cons(\sigma') & \textit{for } \sigma = (e :: \sigma') \\ 0 & \textit{else.} \end{cases}$$

Therefore, $[\,]$ and L consume no value. The pattern $(e :: \sigma')$ consumes the leading value e plus all values that are consumed by σ'.

Operational semantics are then simply defined by the firing rules of nodes. If a node fires, the finite sequences that are produced for each output buffer are concatenated to that buffer, and values consumed on inputs are removed from

input buffers. Formally, this is defined in Definition 3. The definition will make use of a *tail* function with the expected behaviour. It is formally defined in Definition 5.

Definition 3. *DPN operational semantics.*

$$\frac{\text{Rule } f \text{ is enabled in } \mathcal{S}}{\mathcal{S} \to \mathcal{S}'}$$

with

1. *Rule f has input patterns $P_1, ..., P_m$ for inputs $x_1, ..., x_m$ and output sequences $\sigma_1, ..., \sigma_n$ for outputs $y_1, ..., y_n$.*
2. *$\mathcal{S}'(z) = \mathcal{S}(z)$ for all buffers z not appearing as x_i or y_j.*
3. *$\mathcal{S}'(z) = \mathcal{S}(z).\sigma_j$ if $z = y_j$, but does not appear as some x_i.*
4. *$\mathcal{S}'(z) = tail^{cons(P_i)}(\mathcal{S}(z))$ if $z = x_i$, but does not appear as some y_j.*
5. *$\mathcal{S}'(z) = tail^{cons(P_i)}(\mathcal{S}(z)).\sigma_j$ if $z = x_i$ and $z = y_j$.*

Clearly, several rules in one process may be able to fire at once - and several processes may be able to fire at once. Therefore, operational semantics of DPNs in general are not determinstic. In order to achieve a deterministic behaviour, Kahn imposed certain restrictions on the behaviour of process nodes [10]. It is easy to see that these are equivalent to the ones given in Definition 4.

Definition 4. *Kahn requirements for process nodes.*

1. *All firing rules may not check for emptiness of a buffer.*
2. *Only blocking reads of buffers are allowed.*
3. *Process nodes must implement sequential functions.*

As shown in [12,13], these restrictions suffice to yield a deterministic behaviour. Therefore, the following lemma is given without proof.

Lemma 1. *If all process nodes of a DPN fulfill the Kahn requirements (Definition 4), its output is deterministic and latency-insensitive.*

As presented in [14] and [11], a more severe restriction than Kahn requirements are static DPNs. They have statically known consumption and production for every buffer z of the DPN. All firing rules of all nodes write the same amount $np(z)$ and read the same amount of values $nc(z)$ on every firing for a buffer z. For static DPNs, boundedness (among other problems) becomes decidable [11]. We will also introduce similar concepts to balance equations and periodic schedules in static DPNs, which can also be found in [11].

3 Operational Semantics for SCAD Architectures

In this section, operational semantics for SCAD architectures will be provided. First, formal descriptions for the components of SCAD architectures as presented in Section 2.1 are given. Then, based on these descriptions, operational semantics will be described.

3.1 Formal Description

A SCAD architecture is a triple $\mathfrak{A} = (\mathcal{D}, \mathcal{P}, \mathcal{F})$, consisting of datatype $\mathcal{D}$, ports $\mathcal{P}$ and processing units $\mathcal{F}$. Datatype $\mathcal{D}$ always contains a special symbol $\perp \in \mathcal{D}$ denoting the absence of a value. $\mathcal{P} \subseteq (\{\mathfrak{i}, \mathfrak{o}\} \times \mathbb{N}) \cup \{\mathfrak{c}\}$ are the available input and output ports. All SCAD architectures have a finite amount of ports $\mathcal{P}$. Input ports are defined as $in(\mathcal{P}) = \mathcal{P} \cap (\{\mathfrak{i}\} \times \mathbb{N})$ and output ports as $out(\mathcal{P}) = \mathcal{P} \cap (\{\mathfrak{o}\} \times \mathbb{N})$. The $\mathfrak{c}$ denotes the special port reserved for the CU. It is always assumed that $\mathfrak{c} \in \mathcal{P}$. $\mathcal{F}$ is the finite set of PUs. Every PU $(\mathcal{P}', f) \in \mathcal{F}$ also has a subset of input ports $in(\mathcal{P}') \subseteq in(\mathcal{P})$, output ports $out(\mathcal{P}') \subseteq out(\mathcal{P})$ and an associated set of firing rules f as defined for DPNs in Section 2.2 consuming values from a $Stream_{\mathcal{D}}^{|in(\mathcal{P}')|}$ and producing values to $Stream_{\mathcal{D}}^{|out(\mathcal{P}')|}$. The streams are those associated with the input and output ports. As the special port $\mathfrak{c}$ is part of the CU and not of any PU, $\mathfrak{c} \notin \mathcal{P}'$ must hold. The ports of two PUs $p, q \in \mathcal{F}$ are mutually exclusive, meaning $\mathcal{P}(p) \cap \mathcal{P}(q) = \emptyset$. Input and output from the SCAD architecture itself may be considered. If so, the input and output will be denoted by $\mathfrak{i}_{out}$ and $\mathfrak{o}_{in}$ in this paper. These will be used to input values to the SCAD architectures ($\mathfrak{o}_{in}$) and to output values from it ($\mathfrak{i}_{out}$). For simplicity, we require that all other input and output ports are connected to a PU.

3.2 Operational Semantics

To define operational semantics, a notion of state is introduced. A configuration $\mathcal{C} = (\mathfrak{P}, pc, st, \mathcal{A}, \mathcal{V})$ of a SCAD architecture must contain the contents of the *adr* and *val* buffers of all ports $\mathcal{P}$ as described in Section 2.1. Thus, for all ports the function $\mathcal{A} : \mathcal{P} \to Stream_{\mathcal{P}_\perp}^1$ with $\mathcal{P}_\perp = \mathcal{P} \cup \{\perp\}$ maps to a sequence of port addresses $\mathcal{P}$ and the special symbol $\perp$ denoting absence of a value. The same goes for values $\mathcal{V} : \mathcal{P} \to Stream_{\mathcal{D}}^1$ which maps to a sequence of the datatype $\mathcal{D}$ (note that $\perp \in \mathcal{D}$ holds by definition). $\mathfrak{P}$ is the program currently in execution and $pc \in \mathbb{N}$ is the current program counter. The $st \in \mathbb{B}$ flag indicates whether the CU is currently stalling or not.

A program $\mathfrak{P}$ consists of move instructions $l : src \to tgt$ where $l \in \mathbb{N}$, $src \in out(\mathcal{P})$ and $tgt \in in(\mathcal{P}) \cup \{\mathfrak{c}\}$. The labels l of instructions must be mutually exclusive. If $(l_1 : src_1 \to tgt_1), (l_2 : src_2 \to tgt_2) \in \mathcal{P}$ and $l_1 = l_2$, then $src_1 = src_2$ and $tgt_1 = tgt_2$ must hold. Therefore, the labels of two distinct instructions always differ. $src \to tgt$ will transport a value from output port src to input port tgt. $\mathfrak{c}$ is the special input port of the control unit. If the control unit is not stalling - which is true if $st = 0$ - it always fetches instructions

according to their label l and the state of the program counter pc. If a value is sent to $\mathfrak{c}$ it is used as the new value of the pc. As long as the control unit waits for this value, it is stalling, meaning $st = 1$.

The following definition of operational semantics is inspired by the descriptions in [6,15,16]. The CU reads instructions from $\mathfrak{P}$ and manages the program counter pc and stall flag st. The streams of $\mathcal{C}$ primarily act as FIFO buffers. PUs in $\mathcal{F}$ can fire if their associated firing functions f find enough values in their respective input buffers. The values they actually read and use are consumed and removed from the input buffers. Its results are written to the respective output buffers, and stored in the next configuration. To summarize, a step $\mathcal{C} \to \mathcal{C}'$ in $\mathcal{S}$ can be made if a new move instruction is read from $\mathfrak{P}$ and added to the address buffers (Rules 1, 2), a new value for the pc is set and the CU is unstalled (Rule 3), a move between PUs is performed (Rule 4) or an enabled firing rule of a PU is used (Rule 5). But in order to simplify the construction of these rules, we will first define some notations and functions. First, $head : Stream^1_{\mathcal{X}} \to \mathcal{X}$ and $tail : Stream^1_{\mathcal{X}} \to Stream^1_{\mathcal{X}}$ for $\mathcal{X} \in \{\mathcal{P}, \mathcal{D}\}$ with the expected behaviour are defined in Definition 5.

Definition 5. *Head and tail functions.*

$$head(S) := \begin{cases} x & \text{for } S = (x : S') \\ \bot & \text{for } S = \epsilon \end{cases}$$

$$tail(S) := \begin{cases} S' & \text{for } S = (x : S') \\ \epsilon & \text{for } S = \epsilon \end{cases}$$

Furthermore, as a lot of substitutions will be necessary on the buffers of configurations when data is read or written, some abbreviations that can be used to transform the mappings $\mathcal{A}$ and $\mathcal{V}$ of the configurations should be defined. In the following, $\mathcal{M} \in \{\mathcal{A}, \mathcal{V}\}$ can be assumed for all definitions. $\mathcal{M}[b' \mapsto S]$ will replace the content of buffer b' in $\mathcal{M}$ with stream S.

Definition 6. *Substitution on configuration mappings.*

$$\mathcal{M}[b' \mapsto S](b) := \begin{cases} \mathcal{M}(b) & \text{for } b \neq b' \\ S & \text{for } b = b' \end{cases}$$

We may write $\mathcal{M}[b_1 \mapsto S_1, b_2 \mapsto S_2]$ for $(\mathcal{M}[b_1 \mapsto S_1])[b_2 \mapsto S_2]$. Now the substitution of Definition 6 can be used to define queue operations directly on the configuration mappings. $\mathcal{M}[b \triangleleft x]$ enqueues x in the stream of buffer b. $\mathcal{M}[b \triangleright]$ dequeues one value from b. And $\mathcal{M}[b \triangleright b']$, dequeues a value from b and then enqueue that same value in b'. The precise definitions are given in Definition 7.

Definition 7. *Triangle notation.*

$$\mathcal{M}[b \triangleleft x] := \mathcal{M}[b \mapsto \mathcal{M}(b).x]$$

$$\mathcal{M}[b \triangleright] := \mathcal{M}[b \mapsto tail(\mathcal{M}(b))]$$

$$\mathcal{M}[b \triangleright b'] := \mathcal{M}[b \triangleright, b' \triangleleft head(\mathcal{M}(b))]$$

Again, we may directly chain the operations, allowing $\mathcal{M}[b_1 \triangleleft x_1, b_2 \triangleleft x_2] := (\mathcal{M}[b_1 \triangleleft x_1])[b_2 \triangleleft x_2]$ (or using $\triangleright$ instead of $\triangleleft$ for any of the respective operations). Finally, $first(X, x, Y, y)$ will find the first position where both stream X is x and stream Y is y. And $S[i \mapsto x]$ replaces position i with value x in stream S. These will be used to define the execution of data moves. Unfortunately, this definition is a bit cumbersome as a specific position inside the input buffer must be replaced. The $first$ function is necessary to find the first position inside an input buffer where the value is $\perp$ and the address is from the source output buffer, as was described in Section 2.1. The substitution will then be used to replace that position of the stream.

Definition 8. *First function and substitution on streams.*

$$first(X, x, Y, y) := i \text{ with } X_i = x, Y_i = y \text{ and } \forall j : X_j = x \wedge Y_j = y \rightarrow j \geq i.$$

$$S[i \mapsto x](j) := \begin{cases} S_j & \text{for } i \neq j \\ x & \text{for } i = j \end{cases}$$

Using these definitions, the operational firing rules can now be given.

Rule 1, ADVANCE. If the CU is not stalling (meaning $st = 0$), try to read instruction $pc : src \rightarrow tgt$. If it exists, enqueue tgt in $\mathcal{A}'(src)$ and src in $\mathcal{A}'(tgt)$. Then increment the program counter $pc' = pc + 1$ if $tgt \neq \mathfrak{c}$. Additionally enqueue $\perp$ in $\mathcal{V}'(tgt)$.

$$\frac{pc : src \rightarrow tgt \in \mathfrak{P} \quad tgt \neq \mathfrak{c}}{(\mathfrak{P}, pc, 0, \mathcal{A}, \mathcal{V}) \rightarrow (\mathfrak{P}, pc + 1, 0, \mathcal{A}', \mathcal{V}')}$$

with

$$\mathcal{A}' = \mathcal{A}[src \triangleleft tgt, tgt \triangleleft src], \mathcal{V}' = \mathcal{V}[tgt \triangleleft \perp]$$

Note that $\perp$ is not enqueued in the src output buffer. This is done to simplify the definition of PU firings. The value part of an output buffer only acts as a FIFO-buffer for the results of the PU anyway. Therefore, $\perp$ values would always be replaced in the production order. If more addresses then values are present, the missing corresponding values of the output buffer can just be imagined to be $\perp$. In a way, this is done by the *head* function that returns $\perp$ on the empty stream.

Rule 2, STALL. This is almost the same case as Rule 1, but $tgt = \mathfrak{c}$. In this case, do not increment pc and stall further instructions with $st' = 1$ to wait for the availability of the value.

$$\frac{pc : src \rightarrow \mathfrak{c} \in \mathfrak{P}}{(\mathfrak{P}, pc, 0, \mathcal{A}, \mathcal{V}) \rightarrow (\mathfrak{P}, pc, 1, \mathcal{A}', \mathcal{V}')}$$

with

$$\mathcal{A}' = \mathcal{A}[src \triangleleft tgt, tgt \triangleleft src], \mathcal{V}' = \mathcal{V}[tgt \triangleleft \perp]$$

Rule 3, `CONTINUE`. If the CU is stalling ($st = 1$), but a value $v = head(\mathcal{V}(\mathfrak{c})) \neq \bot$ is available, consume it and set $pc' = v$. Set $st' = 0$ to unstall the CU. And also dequeue the value and address from $\mathfrak{c}$.

$$\frac{pc' = head(\mathcal{V}(\mathfrak{c})) \neq \bot}{(\mathfrak{P}, pc, 1, \mathcal{A}, \mathcal{V}) \to (\mathfrak{P}, pc', 0, \mathcal{A}[\mathfrak{c}\triangleright], \mathcal{V}[\mathfrak{c}\triangleright])}$$

Rule 4, `DATAFLOW`. In order to fire this rule, for some port $p \in out(\mathcal{P})$ there must be a value v and address q available. So $v = head(\mathcal{V}(p)) \neq \bot$ and $q = head(\mathcal{A}(p)) \neq \bot$. The corresponding move instruction therefore was $p \to q$. As v is transported to a different port, it has to be dequeued with $\mathcal{V}'(p) = tail(\mathcal{V}(p))$ and $\mathcal{A}'(p) = tail(\mathcal{A}(p))$. For input port q we want to replace the value closest to the head of the queue where $\mathcal{V}(q) = \bot$ and $\mathcal{A}(q) = p$ with v. This can be described with $i = first(\mathcal{A}(q), p, \mathcal{V}(q), \bot)$ and $\mathcal{V}'(q) = \mathcal{V}(q)[i \mapsto v]$. To summarize, this yields the following rule:

$$\frac{p \in out(\mathcal{P}) \quad v = head(\mathcal{V}(p)) \neq \bot \quad q = head(\mathcal{A}(p)) \neq \bot}{(\mathfrak{P}, pc, st, \mathcal{A}, \mathcal{V}) \to (\mathfrak{P}, pc, st, \mathcal{A}', \mathcal{V}')}$$

with

$$\mathcal{A}' = \mathcal{A}[p\triangleright], \mathcal{V}' = \mathcal{V}[p\triangleright, q \mapsto \mathcal{V}(q)[i \mapsto v]]$$

and

$$i = first(\mathcal{A}(q), p, \mathcal{V}(q), \bot).$$

Rule 5, `FIRE`. The PU firing rule will mostly be reduced to the DPN firing rule (Definition 3). Analogously to DPN nodes, PU functions are defined on streams and are (in this paper) assumed to be firing rules. For the PU firing, we therefore treat the PU as a DPN node on the value ($\mathcal{V}$) part of the connected input and output buffers. We simply use the operational firing rules of DPNs to determine whether it can fire, and how streams are transformed. The patterns will determine consumption of values and outputs are concatenated to the output buffers. The only difference is that whenever values from $\mathcal{V}$ are consumed, their respective addresses get consumed from $\mathcal{A}$, too. So if n values are dequeued from $\mathcal{V}(p)$ by the firing rule, n values are dequeued from $\mathcal{A}(p)$ as well. In order to make the determinacy proof work, it will also be required that no $\bot$ values may be consumed by the firing rules. This restriction could be left out for other uses, though.

Execution terminates if all rules have fired exhaustively. A run $\mathcal{R}$ of a SCAD architecture is a sequence of configurations where for all adjacent configurations $\mathcal{C}_i.\mathcal{C}_{i+1}$ in the sequence $\mathcal{C}_i \to \mathcal{C}_{i+1}$ holds by any of the rules given before. We define an ASAP run by giving precedence to the rules. `FIRE` has the highest precedence, followed by `DATAFLOW` and then the CU instructions (Rules 1-3). The intuition of this definition is that operations are fired as soon as possible. New instructions are only added if no PU can fire and no data can be moved. So if for example `FIRE` is active, this rule has to be used before firing Rules 1-4 is allowed. Note that giving precedence to one of the CU rules would not change

the runs, since they are mutually exclusive. An ASAP run of a given program is still not canonical however, since multiple PU or data move rules may be active at the same time. By giving arbitrary precedence to firing rules and buffers, a unique *canonical run* may be defined.

Note that Rule 4 is only well-defined if for a move instruction $src \to tgt$ the corresponding entries in both address buffers of src and tgt exist (with a $\perp$ value in the value buffer of tgt at the corresponding position, of course). Since we always add the addresses to both buffers synchronously in Rules 1 and 2, applying rules will not lead to any issues with the definition itself: addresses are always added and removed simultaneously from src and tgt. A problem can therefore only occur when starting from a configuration that does not have corresponding entries in the address buffers. In this paper, it is therefore required that any initial configuration $\mathcal{C}_0$ has empty address buffers, meaning $\mathcal{A}(\mathcal{C}_0)(p) = \epsilon \; \forall p \in \mathcal{P}$. Using this assumption, the issue can be safely ignored for the sake of simplicity. This is also not a severe restriction: addresses stored in the initial configuration can just be added by prepending the corresponding move instructions to the program.

For our discussion on determinism, we now briefly define *Kahn SCAD architectures*: analogously to the restrictions imposed on the processes of Kahn Process Networks [17], PU functions must adhere to the restrictions given in Definition 4. As will be shown in the next section, these restricted SCAD architectures then have a unique deterministic behaviour for any program.

4 Deterministic SCAD Architectures

In this section, determinism and latency-insensitivity of Kahn SCAD architectures is proven. In order to achieve this, a given SCAD architecture $\mathfrak{A} = (\mathcal{D}, \mathcal{P}, \mathcal{F})$ is modelled using a DPN $c(\mathfrak{A})$. First, the basic structure and dataflow between nodes of $c(\mathfrak{A})$ are described. Then, firing rules for the nodes are given. And finally, weak bisimulation between $\mathfrak{A}$ and $c(\mathfrak{A})$ is shown and the final result given. Without loss of generality we will always assume that input ports are denoted as $in(\mathcal{P}) = \{\mathtt{i}1, ..., \mathtt{i}m\}$ and output ports as $out(\mathcal{P}) = \{\mathtt{o}1, ..., \mathtt{o}n\}$.

4.1 DPN Model for SCAD Architectures

The DPN has a node for every input port $\mathtt{i}1, ..., \mathtt{i}m, \mathtt{c}$ and output port $\mathtt{o}1, ..., \mathtt{o}n$ of $\mathfrak{A}$. Furthermore, it has a node for every PU $p \in \mathcal{F}$. To read and distribute move instructions there is also a node for the CU. These are all the nodes necessary for the construction. Figure 4 shows the basic structure of $c(\mathfrak{A})$.

The aforementioned nodes now have to be connected. The control unit (CU) is connected to all input ports with src_j and to all outputs with tgt_i. These buffers are similar to the *adr* buffers of the SCAD architecture. Furthermore, the CU is connected to itself with three buffers $\mathfrak{P}$, *pc* and *st*. Also, the input port $\mathtt{c}$ feeds into the CU. $\mathfrak{P}$ keeps a copy of the program in execution, *pc* is the program counter and *st* the stall flag. Data transports are simulated by the $\mathtt{i}j$ and $\mathtt{o}i$ nodes. For every PU $p = (\mathcal{P}', f) \in \mathcal{F}$ we connect the input ports $in(\mathcal{P}')$ to the PU node p with i_j, and then p to the output ports $out(\mathcal{P}')$ with o_i, using the respective indices j, i used in the enumeration of all ports for the respective port. Additionally, all outpout ports are connected in the same manner to $\mathtt{c}$. The actual values that the PUs consume and produce are in the i_j and o_i buffers. These are similar to the *val* buffers in the SCAD machine. One difference is that i_j buffers only contain values that can already be consumed. *val* buffers may contain $\perp$ values and even valid values after that. The i_j buffer can only contain values up to the first $\perp$ value. This is due to the fact that SCAD input buffers actually behave like FIFO buffers for every respective source address. This is then similar to the $dtn_{i,j}$ buffers - as these are literally FIFO buffers for every possible source address. The DTN connects all n outputs i to all m inputs j, labeled with $dtn_{i,j}$. This is also the reason why we required in the `FIRE` rule that SCAD PUs can only read values up to the first $\perp$ value. If input or output buffers are left without connections to or from a PU node, we add input edges to output buffers and output edges to input buffers from the environment. And next, the firing rules of the aforementioned nodes will be described.

CU. The CU node manages the control state and adds move instructions to the respective buffers. The inputs are $\mathfrak{P}$, *st*, *pc* and $\mathtt{c}$. The output consists of the same (except $\mathtt{c}$) and additionally src_j (for all in_j) and tgt_i (for all out_i), which are similar to the *adr* buffers of the SCAD architecture. In order to determine the output values the *st* flag is read first. If $st = 1$, the CU waits for a value v from the incoming buffer $\mathtt{c}$. The content of the *pc* buffer is read,

but then simply discarded. Then 0 is written to st and v to pc. If $st = 0$, read instruction $l : x \to y$ from $\mathfrak{P}$. If $l \neq pc$, write pc, st and the instruction back to their respective buffers. Instructions will be looped by successive firings in the $\mathfrak{P}$ buffer until the proper label $l = pc$ is found. If $l = pc$, write x to src_y and y to tgt_x. This is equivalent to writing the address to the adr buffer of a PU. If $y \neq \mathfrak{c}$, write $pc + 1$ to the pc stream, 0 to st and the instruction back to $\mathfrak{P}$. If $y = \mathfrak{c}$ the CU must stall, thus 1 is written to st. The algorithm used is summarized in the following sequential program:

```
st' = read(st) // read stall state
pc' = read(pc) // and program counter

if(st' == 0) { // not stalling
    (l : x → y) = read(𝔓)
    if(l != pc') { // l not correct
        write(𝔓, (l : x → y))
        write(pc, pc')
        write(st, st')
    } else { // l correct
        write(src_y, x) // ADVANCE
        write(tgt_x, y)
        write(𝔓, (l : x → y))
        if(y == 𝔠) { // if target is CU
            write(pc, pc') // STALL
            write(st, 1)
        } else {
            write(pc, pc' + 1)
            write(st, 0)
        }
    }
} else { // stalling
    v = read(𝔠) // CONTINUE
    write(pc, v)
    write(st, 0)
}
```

It should be clear to the reader that this sequential program can also be written in terms of firing rules. But since that is way more cumbersome, we omit this here and will refer to the code for better understanding. Furthermore, striking similarities to the CU rules of Section 3 can be found. The cases are annotated in the code with their respective name.

Note that the description as given clearly only needs sequential blocking reads and does not check for emptiness of a buffer. It therefore fulfills the Kahn criteria given in Definition 4. If the correct instruction is not at the head of the $\mathfrak{P}$ buffer, it is simply looped until the right one is found. This could be changed though by adding a feedback buffer for every single instruction.

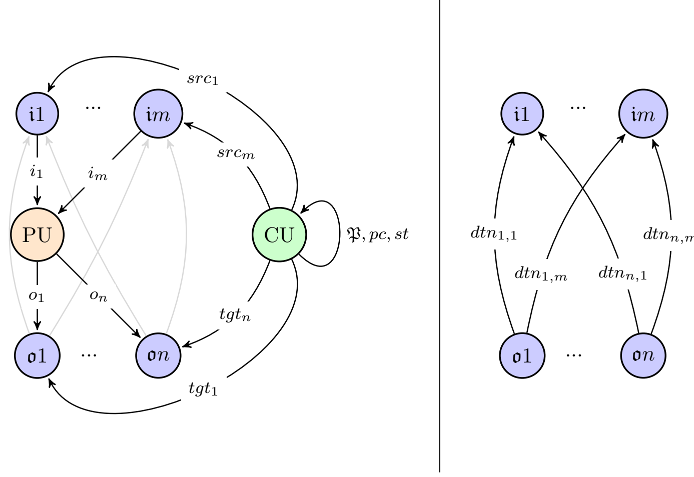


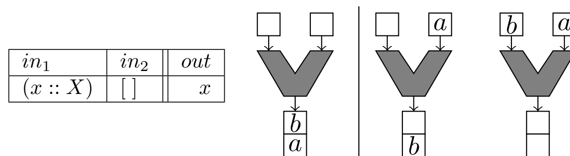
Figure 4: DPN model for a 1-PU SCAD architecture, DTN connections (gray) are not labelled (a) and DTN connections of the DPN model (b)

Nodes $\mathfrak{i}j, \forall j \in \{1, ..., m\}$**.** This node handles the values that will go to i_j (which is then directly connected to a PU). The incoming buffers are src_j (from CU) and $dtn_{i,j}$ (from all $\mathfrak{o}i$). First, address x must be read from src_j. The next value for i_j will come from output buffer x as was scheduled by the CU. Value v from buffer $dtn_{x,j}$ is then read accordingly. v is then written to i_j. These values can then be consumed by the PU. The rules are summarized in the following firing table:

src_j	$dtn_{1,j}$	...	$dtn_{n,j}$	i_j
$1 : S$	$x : X_1$	...	X_n	x
...	...	...	...	...
$n : S$	X_1	...	$x : X_n$	x

Nodes $\mathfrak{o}i, \forall i \in \{1, ..., n\}$**.** This node manages where values from o_i will be sent to. The inputs are tgt_i (from CU) and o_i (from PU), while outputs are $dtn_{i,j}$ to all $\mathfrak{i}j$. First, address y is read from tgt_i. The next value v of o_i is going to be sent to input buffer y as scheduled by the CU. Therefore, v is written to $dtn_{i,y}$. Again, the following firing table summarizes the rules:

tgt_i	o_i	$dtn_{i,1}$	...	$dtn_{i,m}$
$1 : T$	$x : X$	x	...	ϵ
...	...	...	...	...
$m : T$	$x : X$	ϵ	...	x



in_1	in_2	out
$(x :: X)$	$[\,]$	x

Figure 5: A SCAD PU with its firing rule and initial state (a) and a run that prevents the firing rule from firing (b)

PU p. This node implements PU firings of PU $p = (\mathcal{P}', f) \in \mathcal{F}$. It will thus simply use the firing rules f of p. To give an example, f could be defined as as a universal PU as utilized in [6, 9, 15]. A universal PU could be implemented using the following firing table:

opl	opr	opc	cps	out
$x : X$	$y : Y$	$\odot : O$	$n : N$	$(x \odot y)^n$

The table can be extended to any number of binary relations without loosing its sequential properties. Note that this trivially fulfills the Kahn criteria.

Some insights can be gained by comparing the given DPN model to the previous description of SCAD architectures. Also, the DPN model does have significant differences that will influence the bisimulation proof in Section 4.2. Here are some observations to consider:

Fixed-size Patterns. It is important to note that $\mathfrak{A}$ and $c(\mathfrak{A})$ may exhibit different behaviours if fixed-size patterns are used in PU firing rules. Consider the example in Figure 5. Now using SCAD operational semantics, if $out \to in_2$ and then $out \to in_1$ are to be executed, the sequence in Figure 5(b) will occur. Thus, the firing rule can never match the input stream and execution terminates without the PU ever firing. In the DPN this is not necessarily the case, though. In the following, a run is given that enables the PU to fire in the corresponding DPN: first, move instructions are added by the CU to src_1, src_2, tgt_1 and tgt_2. Then, $\mathtt{o2}$ may fire twice and put a into $dtn_{1,2}$, as well as b into $dtn_{1,1}$. Now $\mathtt{i1}$ and $\mathtt{i2}$ can both fire. Since they are independent, they can fire in any order. Therefore, it is possible for $\mathtt{i1}$ to fire first and put b into i_1. The PU then only sees b in i_1 while i_2 is empty, and can fire. This clearly implies that the constructed DPN and SCAD architecture are not bisimilar if fixed-sized pattern matching is present in a firing rule. As will be shown however, this problem is indeed limited to fixed-size pattern matching: if only prefixes of streams can be matched, this is no problem: the DPN always sees less or equal values in its PU input buffers. If only prefixes are matched, firing rules active in the DPN are also active in the SCAD architecture.

FIFO Buffers. By looking at the `DATAFLOW` rule it should be clear that input buffers of SCAD architectures are not FIFO buffers. In the DPN, they can

however be simulated by two stages of strict FIFO buffers: first, one for each possible source. And secondly, one for the input that may be consumed by the PU. This does however mean that precise analysis of input buffer sizes on the DPN does not directly carry over to the actual SCAD architecture. Maybe this can be simulated with a different construction. Boundedness in general should however carry over between the two models.

Consumption Limitation. Furthermore, the models are only equivalent if the PUs of the SCAD architecture can not consume and read values behind the first $\perp$ value in an input buffer. The DPN stores these values in different buffers (the $dtn_{i,j}$ buffers), not accessible to the PU. Therefore, for the sake of making the models equivalent, we assumed that SCAD architectures can not do this either.

Interconnection. The DTN interconnection network of the SCAD architecture is simulated by several nodes and connections in the DPN, since there are only point-to-point connections in DPNs. This is one of the reasons why we will only be able to show weak bisimulation of the two models.

Immediate Values. Immediate values may be added by adding another output buffer. They can also be connected to the CU by differentiating between different instruction types, and then sending values to the output buffer for immediate values. It is however not trivial to add immediate values to the instructions, implying that this is a different type of instruction.

4.2 Weak Bisimulation

Using the model $c(\mathfrak{A})$ and DPN semantics (Definition 3), a different way of defining operational semantics for SCAD architectures $\mathfrak{A}$ can be given. In this section, we will show with weak bisimulation that this approach is actually reasonable. Then, conclusions from results known for DPNs showing the determinacy of Kahn SCAD architectures are drawn in the next section.

First, it has to be defined how the configurations of the SCAD architecture and the constructed DPN are actually related to each other. For that, we give a relation $\mathcal{R}$ between SCAD architecture configurations and the state of the derived DPN $c(\mathfrak{A})$. Note that $\mathcal{R}$ can then also be used to transform initial configurations between the different models. (p, q) is in $\mathcal{R}$, where $p = (\mathfrak{P}, pc, st, \mathcal{A}, \mathcal{V})$ is the SCAD configuration and q the DPN configuration (mapping buffer names to streams), iff the following requirements 1-5 are fulfilled.

Requirement 1, Program Equivalence. Program $\mathfrak{P}$ is equivalent to $q(\mathfrak{P})$ when treated as a set (note that the labels make instructions unique):

$$\mathfrak{P} = \{i \mid i \in q(\mathfrak{P})\}$$

The intuition is simply that both models must be executing the same program.

Requirement 2, Control Flow. It is true that $pc = q(pc)$ and $st = q(st)$. This also implies $|q(pc)| = 1$ and $|q(st)| = 1$. This just means that the control state of both models must be equivalent.

Requirement 3, Output Equivalence. For all $\mathfrak{o}1$, ..., $\mathfrak{o}n$:

$$\mathcal{V}(\mathfrak{o}i) = q(o_i) \text{ and } \mathcal{A}(\mathfrak{o}i) = q(tgt_i)$$

The address and value part of output buffers in both models are directly related with $\mathcal{R}$.

Requirement 4, Input Equivalence 1. For all $\mathfrak{i}1$, ..., $\mathfrak{i}m$: exhaustively simulate firing of the $\mathfrak{i}j$ nodes in $c(\mathfrak{A})$ by replacing missing $dtn_{i,j}$ values with $\perp$ until src_j would be empty - then the resulting Stream $s \subseteq Stream^1_{\mathcal{D}}$ must fulfill $s = \mathcal{V}(\mathfrak{i}j)$. To define this formally we first define the function con_j to simulate the exhaustive firing in Definition 9.

Definition 9. *Exhaustive firing simulation.*

$$con_j(src, d_1, ..., d_n) = \begin{cases} \epsilon \text{ if } head(src_j) = \perp \\ head(d_{head(src)}).con'_j \text{ else.} \end{cases}$$

where

$$con'_j = con_j(tail(src), d'_1, ..., d'_n)$$

and

$$d'_i = \begin{cases} tail(d_i) & \text{if } head(src) = i \\ d_i & \text{else.} \end{cases}$$

Then, the stream s_j can be defined in terms of the initial configuration of $q(\mathfrak{i}j)$, with the results of the exhaustive firing appended to it.

$$s_j = q(\mathfrak{i}j).con_j(q(src_j), q(dtn_{1,j}), ..., q(dtn_{n,j}))$$

However, we need to further require that this is actually exhaustive. Let $dtn^*_{i,j} \forall i \in \{1, ..., n\}$ be the d_i parameters used in the last recursive call of con_j. Then, $dtn^*_{i,j}$ must be empty and Requirement 4 simply becomes:

$$s_j = \mathcal{V}(\mathfrak{i}j) \text{ and } dtn^*_{i,j} = \epsilon$$

Requirement 5, Input Equivalence 2. For all $\mathfrak{i}1$, ..., $\mathfrak{i}m$: removing the first $|q(i_j)|$ elements from $\mathcal{A}(\mathfrak{i}j)$, must be equivalent to $q(src_j)$ - the DPN does not contain addresses of values that are ready to consume, so the ones on the SCAD architecture must be ignored in $\mathcal{R}$.

$$tail^{|q(i_j)|}(\mathcal{A}(\mathfrak{i}j)) = q(src_j)$$

To reiterate, the tuple (p, q) consisting of SCAD configuration p and DPN configuration q is in $\mathcal{R}$, iff requirements 1-5 hold for them. Now using $\mathcal{R}$, we can finally show weak bisimulation of $\mathfrak{A}$ and $c(\mathfrak{A})$. This is done in the following Lemma 2.

Lemma 2. $\mathfrak{A}$ *and* $c(\mathfrak{A})$ *are weakly bisimilar.*

Proof. For the proof, we can always assume requirements 1-5 hold prior the firing any operation. Then, it must be shown that all of the requirements are satisfied after firing the respective operations. First, we show simulation of $c(\mathfrak{A})$ by $\mathfrak{A}$. Assume $(q,p) \in \mathcal{R}^{-1}$. For $q \to q'$ any of the DPN nodes can fire. We do a case distinction on the type of node (CU, $\mathfrak{i}j$, $\mathfrak{o}i$ or PU) that is firing:

Case CU. Either no action is necessary, or we do the equivalent one in SCAD. We further split this case into which rule needs to be fired in the SCAD architecture.

No Action. If $q(st) = 0$, we read instruction $l : x \to y$. If $l \neq pc$, we write all values back. The resulting state q' is still equivalent to p. The only thing changed is the order of the $\mathfrak{P}$ buffer, which does not matter for the relation $\mathcal{R}$ since we only demanded set equivalence of $\mathfrak{P}$ and $q(\mathfrak{P})$ in Requirement 1. The conditions of the other requirements are not changed by the firing and therefore still hold.

Continue. If $q(st) = 1$, $st = 1$ must hold as well (Requirement 2). We can only fire if $head(q(\mathfrak{c}))$ has a value, and if that is the case, so does $\mathcal{V}(\mathfrak{c})$ (Requirement 3). In SCAD, we then fire the `CONTINUE` rule. In both models, the values get consumed and we write $pc' = v$ and $st = 0$. Therefore, Requirement 2 and Requirement 3 are fulfilled in the next state. The other requirements are not touched and therefore still hold trivially.

Advance. If $q(st) = 0$ and $q(pc) = l \neq \mathfrak{c}$, the `ADVANCE` rule must be fired in the SCAD architecture. Both enqueue x in $\mathcal{A}(\mathfrak{i}y)$ and $q(src_y)$, as well as y in $\mathcal{A}(\mathfrak{o}x)$ and $q(tgt_x)$. SCAD additionally adds a $\perp$ to the tail of $\mathcal{V}(x)$. Because $\mathcal{A}(\mathfrak{o}x) = q(tgt_x)$ (Requirement 3), clearly $\mathcal{A}(\mathfrak{o}x).y = q(tgt_x).y$. Therefore, Requirement 3 still holds. Requirement 1 and 2 are not touched, and therefore still hold. It remains to show that requirements 4 and 5 are fulfilled. For the input buffer in the DPN, the missing $\perp$ will be inserted by the exhaustive firing con_x, which now has an additional recursion because it will be called on $q(src_x.y)$ with $y \neq \perp$, instead of just $q(src_x)$. Note that since before, $dtn^*_{x,y} = \epsilon$ (the state of the last recursion, Requirement 4), the added address y will definitely yield a $\perp$ at the tail of s'_x, with the rest being unchanged. This is because $head(\epsilon) = \perp$. This is convenient, since the SCAD architecture enqueues a $\perp$ to $\mathcal{V}(\mathfrak{i}x)$. Therefore, $s_x.\perp = \mathcal{V}(\mathfrak{i}x).\perp$, which then fulfills Requirement 4. Now, because of Requirement 4 and the fact that SCAD keeps an address for every value, it is clear that $|q(i_x)| \leq |\mathcal{A}(\mathfrak{i}x)|$. Values stored in $q(i_x)$ must be a prefix of the ones in $\mathcal{V}(\mathfrak{i}x)$, and $|\mathcal{V}(\mathfrak{i}x)| = |\mathcal{A}(\mathfrak{i}x)|$. Therefore, the *tail* function will at most remove $|\mathcal{A}(\mathfrak{i}x)|$ many values. Therefore, for $|\mathcal{A}(\mathfrak{i}x)|+1$ values, the last value will at least remain. This implies $tail^{|q(i_x)|}(\mathcal{A}(\mathfrak{i}x).y) = q(src_x).y$, which satisfies Requirement 5. Therefore, all requirements are satisfied and $(q',p') \in \mathcal{R}^{-1}$.

Stall. If $q(st) = 0$ and $q(pc) = l = \mathfrak{c}$, the `STALL` rule must be fired in the SCAD architecture. Then, $st' = 1$ and $q'(st) = 1$ are set, which satisfies Requirement 2. The rest of this case is equivalent to *Advance*.

Case $\mathfrak{i}j$. No action in the SCAD architecture is necessary. Since $\mathcal{R}$ only requires the result of exhaustive firing of $\mathfrak{i}j$ nodes to be equivalent to the SCAD configuration, Requirement 4 is still satisfied. In Requirement 5 we have one less value in src_j, but we also remove one more value from $\mathcal{A}(\mathfrak{i}j)$, which is still fine: $tail^{|q(i_j)|}(\mathcal{A}(\mathfrak{i}j)) = q(src_j)$ (Requirement 5) directly implies $tail^{|q(i_j)|+1}(\mathcal{A}(\mathfrak{i}j)) = tail(q(src_j))$. Therefore, the new DPN state q' still satisfies $(q', p) \in \mathcal{R}^{-1}$.

Case $\mathfrak{o}i$. Is simulated using the `DATAFLOW` rule in SCAD. Values and addresses are dequeued from the output buffer and are then transported to the respective position in the input buffer. The DPN uses the dtn buffers to determine this *respective position*, while the SCAD architecture uses the $first$ function.

Clearly, $j = head(q(tgt_i)) = head(\mathcal{A}(\mathfrak{o}i))$ and $v = head(q(o_i)) = head(\mathcal{V}(\mathfrak{o}i))$ must hold (Requirement 3). Also, if the firing rule of $\mathfrak{o}i$ is active, $v \neq \perp$ and $j \neq \perp$, which also implies that `DATAFLOW` can be fired in SCAD. The move being performed is $i \to j$, transporting value v from output buffer $\mathfrak{o}i$ to input buffer $\mathfrak{i}j$. Because of Requirement 3, $\mathcal{V}(\mathfrak{o}i) = q(o_i)$ and $\mathcal{A}(\mathfrak{o}i) = q(tgt_i)$ hold. This directly implies $tail(\mathcal{V}(\mathfrak{o}i)) = tail(q(o_i))$ and $tail(\mathcal{A}(\mathfrak{o}i)) = tail(q(tgt_i))$. Therefore, Requirement 3 is still satisfied. The premises of requirements 1 and 2 are not touched and therefore they remain valid. But the content of input buffer j will change. Therefore, requirements 4 and 5 remain to be shown. In the SCAD architecture $first(\mathcal{A}(\mathfrak{i}j), i, \mathcal{V}(\mathfrak{i}j), \perp)$ will find the closest address i to the head of the buffer where $\mathcal{V}(\mathfrak{i}j)$ is $\perp$, and then replaces that $\perp$ with v. In the DPN, exhaustive firing must have yielded a $\perp$ symbol at some position in the sequence s_j, because the SCAD architecture has a $\perp$ symbol and Requirement 4 was valid. Note here that this line of argument is only okay because it was required in Section 3 that configurations start with no values in address buffers, and we just assume that corresponding entries always exist. When pushing v into $dtn_{i,j}$, this clearly replaces the one closest to the head where the corresponding address is i. This is because all values with corresponding address are read from the $dtn_{i,j}$ buffer. Therefore, Requirement 4 still holds true. The amount of values removed at the head of $\mathcal{A}(\mathfrak{i}j)$ did not change, and src_j has not been touched. Therefore, Requirement 5 still holds as well.

Case PU. The content of the input streams i_j in the DPN is a prefix of the one in SCAD. If none of the $\mathfrak{i}j$ nodes can fire, the first value that is not in i_j but in the SCAD architecture can only be a $\perp$. The SCAD architecture can not read values beyond the first $\perp$. Therefore, the SCAD architecture sees more or equal amounts of values in the buffers. Since we defined semantics of PU firing as a DPN using the same firing rule, a rule firing in the DPN must be active in the SCAD architecture as well if matching with fixed size patterns is not allowed. The SCAD architecture additionally consumes its addresses when the rule fires.

This is however fine, since we do not care for the addresses that can be consumed, namely the first $|q(i_j)|$ in the SCAD architecture. And the rule firing can at most consume $|q(i_j)|$ tokens. Therefore, this case can simply be simulated using the `FIRE` rule using the same firing rule as the DPN. Requirement 5 still holds by the argument before. And as equal values are consumed and produced in both models, requirements 3 and 4 still hold as well. requirements 1 and 2 are not touched by this case.

Next, we show simulation of $\mathfrak{A}$ by $c(\mathfrak{A})$. Assume $(p, q) \in \mathcal{R}$. For $p \to p'$ we can perform any of the given rules 1-5 from Section 3.

Case `ADVANCE`. In this case, $st = 0$, so $q(st) = 0$, too. The DPN might need to find the appropriate instruction pc. We therefore fire the node as often as necessary until we get the instruction $l : x \to y$. This is fine, only set equivalence of $\mathfrak{P}$ and $q(\mathfrak{P})$ was required ,the order does not matter. Then, the corresponding addresses are enqueued in the appropriate buffers in both models. If the move instruction is $i \to j$, then i is enqueued in src_j in the DPN. In SCAD it is enqueued in $\mathcal{A}(\mathfrak{i}j)$. Since the amount removed at the head of this buffer does not change, Requirement 5 therefore still holds. The same is true for Requirement 3, the same addresses are enqueued in the respective output buffers. The other requirements are not touched. This clearly yields equivalent sates $(p', q') \in \mathcal{R}$.

Case `STALL`. This is basically the same case as `ADVANCE`. If $y = \mathfrak{c}$ we set $st = 1$ in both models and react in the same manner.

Case `CONTINUE`. In this case, $st = 1$, so $q(st) = 1$, too. The DPN node of the CU can then only fire once a value is available at $q(\mathfrak{c})$. This is the same case for SCAD, and both models will then set the pc to the value read and unstall.

Case `DATAFLOW`. A move action from output port i is simulated in the DPN by firing the appropriate $\mathfrak{o}i$ node. Consumption is equivalent in both models. Why this yields equivalent states was detailed in case $\mathfrak{o}i$ of the first direction.

Case `FIRE`. The SCAD architecture may have more values available in an input buffer $\mathfrak{i}j$. However, since it can only read up to the first $\perp$ value, we know that we can make the DPN i_j buffer equivalent up to the first $\perp$ by firing the $\mathfrak{i}j$ nodes (as ensured by Requirement 4). Then, the DPN has the same activated PU firing rules.

And this case concludes the bisimulation proof. Using the same relation $\mathcal{R}$, $\mathfrak{A}$ can simulate $c(\mathfrak{A})$ and vice versa. $\square$

4.3 Conclusions from Kahn Criteria

Now more substantial results can be given. We restrict ourselves to PUs that fulfill the Kahn criteria, as required by the Kahn SCAD architecture. This has some direct implications:

Lemma 3. *If the firing rules of the PU nodes fulfill the Kahn criteria, the output of $c(\mathfrak{A})$ and therefore of $\mathfrak{A}$ is uniquely determined and scheduling-independent.*

Proof. All given functions of the DPN fulfill the Kahn criteria. If additionally the PU nodes do, the statement follows for $c(\mathfrak{A})$ from Lemma 1. From Lemma 2 and the fact that $\mathcal{R}$ directly related output buffers, it therefore also follows for $\mathfrak{A}$. □

First, this finally proves that SCAD architectures produce uniquely determined and scheduling-independent results for a large class of PU implementations. But furthermore, in model-checking it is desirable to restrict the amount of states. In a DPN or SCAD architecture, the same program can yield a lot of runs. Certain runs can even lead to an infinite amount of states, even if that is not necessary. If the DPN does however fulfill the Kahn requirements, the task gets much simpler. If we want to show that a property holds for the result of all runs, we actually only need to show that it holds for one specific run. However, in order to create a finite transition system with operational semantics for model-checking this way, the SCAD architecture has to be bounded on this specific run. Therefore, to do model-checking or when modeling reactive systems in general, it is of interest to compute boundedness or memory requirements. We will now show that the restriction to Kahn requirements is probably not enough to make this decidable for the SCAD architectures.

Lemma 4. $\mathfrak{A}$ *and* $c(\mathfrak{A})$ *with PU functions restricted to Kahn requirements are Turing-complete.*

Proof. We provide an alternative simulation of queue machines on SCAD to the one described in [6]. Instead of translating a queue machine program to a SCAD program $\mathfrak{P}$, we use a fixed $\mathfrak{P}$ that can simulate all queue machine programs given as another input. Assume $\mathcal{S}$ with one PU which has input buffers P, C, Q and output buffers P', C', Q', mQ using the following firing function. First, read $(l : op)$ from P and $pc : C$ (which are read in every operation). If $l = pc$, do the following.

P	C	Q	P'	C'	Q'	mQ
$(l : \odot) : P$	$pc : C$	$x : y : Q$	$(l : \odot)$	$pc + 1$	$(x \odot y)$	l_1
$(l : dup) : P$	$pc : C$	$x : Q$	$(l : dup)$	$pc + 1$	$x.x$	l_2
$(l : swap) : P$	$pc : C$	$x : y : Q$	$(l : swap)$	$pc + 1$	$y.x$	l_2
$(l : goto\ i) : P$	$pc : C$	Q	$(l : goto\ i)$	i		l_0
$(l : ifGoto\ i) : P$	$pc : C$	$x : Q$	$(l : ifGoto\ i)$	$(x?i : pc + 1)$		l_0
$(l : halt) : P$	$pc : C$	Q	$(l : halt)$	-1		l_0

If $pc \neq l \geq 0$, just enqueue $(l : op)$ back to P', pc to C and l_0 to mQ. If $l = -1$, there is no firing rule - execution halts. Executing the following program the

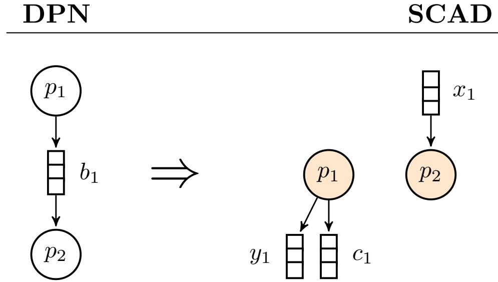


Figure 6: One buffer in the DPN becomes three buffers in the SCAD architecture.

SCAD architecture can clearly simulate execution of the queue machine program stored in input buffer P.

```
0 : mQ → c // initially wait for op
l_0 : P' → P // no values produced to Q
l_0 + 1 : C' → C
l_0 + 2 : mQ → c
l_1 : P' → P // 1 value produced to Q
l_1 + 1 : C' → C
l_1 + 2 : Q' → Q
l_1 + 3 : mQ → c
l_2 : P' → P // 2 values produced to Q
l_2 + 1 : C' → C
l_2 + 2 : Q' → Q
l_2 + 2 : Q' → Q
l_2 + 3 : mQ → c
```

Note that the amount of values produced to Q' may vary, which is why we need mQ to decide how many have to be moved. Each of the possibilities (0, 1, 2) have their own label and subroutine. □

The program in SCAD can closely simulate execution of a queue machine while using the same amount of memory in Q. Therefore, there is no hope that boundedness or other interesting characteristics are decidable for $\mathfrak{A}$ and $c(\mathfrak{A})$ in general. This will motivate further restrictions in Section 5.

4.4 Transformation of DPNs to SCAD Architectures

Since we just provided a translation from SCAD architectures to DPNs, one might wonder if SCAD architectures can express DPNs in a similar fashion. That simulating DPNs using SCAD architectures is possible at all is obvious, since both DPNs and SCAD architectures are Turing-complete (Lemma 4). It is however not clear how hard it may be to do so.

But as it turns out, all DPNs can be simulated with all of their asynchronous behaviours by a SCAD architecture and program using the following simple construction: Assume the DPN has nodes p_1, ..., p_k with buffers $b_1, ..., b_n$ connecting them. For each DPN node p_i, add a SCAD PU q_i. All buffers b_j are split into output buffers y_j connected to $in(b_j)$ and input buffers x_j connected to $out(b_j)$. Furthermore, for each y_j, we add another output buffer connected to the same PU called c_j. The way the buffers are split and connected can be seen in Figure 6. These buffers will send their value to the *control* PU. This has the following firing rule and input buffers:

$from_1$	$from_2$	...	$from_n$	$next$
$x : X_1$	X_2	...	X_n	$label_1$
X_1	$x : X_2$	...	X_n	$label_2$
...	...	...	...	...
X_1	X_2	...	$x : X_n$	$label_n$

For each DPN buffer, there is an input buffer. And if there is a value on an input buffer $from_j$, it will output an according $label_j$. Obviously, this node is not sequential. Next, we describe program $\mathfrak{P}$. Initially, add the following code:

```
1 : x_1 → y_1 // data transport
2 : c_1 → from_1 // inform control
3 : x_2 → y_2 // ...
4 : c_2 → from_2
...
2n − 1 : x_n → y_n
2n : c_n → from_n
2n + 1 : next → c // send to CU
```

Additionally, for each buffer b_j we add the following code:

```
label_j : x_j → y_j // data transport
label_j + 1 : c_j → from_j // inform control
label_j + 2 : next → c // send to CU
```

And finally, we need to extend the function of p_i to that of q_i. If p_i produces a value to b_j, produce it to both y_j and c_j. The principle of the transformation is simple: in the DPN, data is always transported from y_j to x_j automatically as soon as data arrives, as this is only a single buffer b_j. In SCAD, data has to be moved explicitly with move instructions. So for each buffer, we set up a single move instruction $y_j \to x_j$. As soon as a value is produced to y_j, the move instruction may fire and needs to be replenished. Therefore, we replenish it at $label_j$ of the program. The *control* PU is signaled via c_j, collects all of the replenish requests and sequentializes them for the CU. The problem with fixed-size patterns that was discussed in Section 4.1 may occur for the construction as well. But apart from this restriction, it can be seen that the constructed SCAD architecture will weakly bisimulate the given DPN.

5 Periodic Branchless Move Programs

In this section, periodic branchless move programs (on static SCAD architectures) will be discussed. These are very similar to periodic schedules of static DPNs and share many of their properties. They have decidable boundedness and can be model-checked quite easily. Therefore, we will restrict the PU functions of the SCAD architecture to fit static DPNs. The restriction is simply to only allow static consumption and production as defined for static DPNs in Section 2. Static PUs also fulfill the Kahn criteria by definition. Thus, results from the previous section directly apply. We will first discuss the definition of periodic branchless move programs. Then, a decision procedure for boundedness and other related problems are discussed. Thirdly, an easy form of denotation is given. And finally, a methodology for model-checking is outlined.

A *periodic branchless move program* will simply be denoted with $\mathfrak{P} \subseteq \mathbb{N} \times \mathcal{P} \times \mathcal{P}$ in this section. They can be written as

$1 : src_1 \rightarrow tgt_1$
$2 : src_2 \rightarrow tgt_2$
...
$n : 1 \rightarrow \mathfrak{c}$

where for all $i \in \{1, ..., n-1\}$ $tgt_i \neq \mathfrak{c}$ holds. Periodic branchless move programs therefore only include a single branch, from the last instruction back to the first one. All move instructions are executed in an infinite loop without additional branches. A run $\mathcal{R}$ of a SCAD program is *bounded*, iff there is a n such that $|\mathcal{V}(p)| \leq n \wedge |\mathcal{A}(p)| \leq n \quad \forall p \in (\mathcal{P} \setminus \{\mathfrak{o}_{in}, \mathfrak{i}_{out}\})$ for all $\mathcal{C} \in \mathcal{R}$. Note that input and output to the SCAD architecture itself is explicitly ignored in this definition. Making sure that these are bounded is assumed to be the task of the environment. It is also assumed that there is no input and output to the SCAD architecture at all until stated otherwise. Because it does not influence boundedness anyway, it would only create unnecessary case distinctions in the following discussion. As already mentioned, PUs will be restricted to static consumption and production patterns. To recapitulate, this means that for every input buffer $\mathfrak{i}j$ we know the number of tokens consumed $nc(\mathfrak{i}j)$, as well as for every output buffer $\mathfrak{o}i$ the number of tokens produced $np(\mathfrak{o}i)$ on every firing of the respective PU. As will be shown in the following, this suffices to make boundedness for periodic branchless move programs decidable.

It should be noted that the following results could also be derived using a different approach: as previously mentioned, periodic branchless move programs basically are periodic schedules of static DPNs (for reference on periodic schedules see [11]). All that has to be done to transform the schedule of a static DPN into a SCAD architecture and periodic branchless move program is the following: the PUs of the SCAD architecture are the nodes of the DPN similarly to the construction of Section 4.4. But there is no need to create the c_j buffer or *control* PU. The schedule always moves a static amount of values from y_i to x_i in a period. This can be encoded in the periodic branchless move program. The reverse is also possible by using (statically scheduled) multiplexer

and demultiplexer nodes, since the location to which all values have to be sent can be statically decided. Using these transformations, techniques and results known from static DPNs could also be applied to move programs. We will however refrain from doing this for the sake of developing methods and results directly for move programs. Also, some of these methods will become necessary in Section 6.

5.1 Decision Procedure for Boundedness

In this section, a decision procedure for boundedness will be derived and initial configurations of programs discussed. In order to do so, a few definitions will be given first. For an input buffer $\mathfrak{i}j$, let $m(\mathfrak{i}j) = |\{l : src_i \rightarrow tgt_j \mid \mathfrak{i}j = tgt_j\}|$. Analogously, for output buffer $\mathfrak{o}i$ let $m(\mathfrak{o}i) = |\{l : src_i \rightarrow tgt_j \mid \mathfrak{o}i = src_i\}|$. Clearly, $m(\mathfrak{i}j)$ denotes the amount of values moved to $\mathfrak{i}j$ and $m(\mathfrak{o}i)$ the amount of values moved away from $\mathfrak{o}i$. The amount of tokens consumed from input ports and produced to output ports will depend on the amount of values consumed and produced by the respective PU. The number of firings of the PU will depend on the number of tokens moved to its input buffers. Thus, the number of (potential) firings of PU p can be given as $NF(p) = \frac{m(\mathfrak{i}j)}{nc(\mathfrak{i}j)}$, where $\mathfrak{i}j \in \mathcal{P}(p)$. While in general this would yield different results depending on which $\mathfrak{i}j \in \mathcal{P}(p)$ is chosen, it is by definition well-defined for balanced periodic branchless move programs as given in Definition 10. In the following, we will only consider balanced programs, thus it is well-defined for all programs in this paper. The amount of values produced to output buffer $\mathfrak{o}i$ of PU p by firings is then $pf(\mathfrak{o}i) = np(\mathfrak{o}i) \cdot NF(p)$. Using these definitions the notion of program balance can now be given:

Definition 10. *A periodic branchless move program $\mathfrak{P}$ is balanced, iff*

1. $\forall \mathfrak{o}i \in \mathcal{P} : pf(\mathfrak{o}i) = m(\mathfrak{o}i)$
2. $\forall p \in \mathcal{F} : \forall \mathfrak{i}j_1, \mathfrak{i}j_2 \in in(\mathcal{P}(p)) : \frac{m(\mathfrak{i}j_1)}{nc(\mathfrak{i}j_1)} = \frac{m(\mathfrak{i}j_2)}{nc(\mathfrak{i}j_2)}$.

The intuition is that all input buffers of the same PU must agree on the amount of firings (2). Also, the amount transported away from an output buffer must equal the amount produced to it (1). Notice that not all of these values need to be natural numbers. The number of firings set up in an iteration may for example be $\frac{1}{2}$, implying that the PU will only fire every second iteration. After the second iteration, input values can be consumed and output move instructions added in the previous iteration can be executed. But as long as buffers stay balanced in the long run, this does not matter for boundedness. Therefore, the period λ of a PU p will be introduced. The period will denote after how many iterations the PU can potentially consume all its inputs and therefore become balanced again. Thus, the period depends on whether the number of firings is a natural number or not:

Definition 11. *Period of a PU p.*

$$\lambda(p) = \begin{cases} NF(p) & \text{if } NF(p) = \lfloor NF(p) \rfloor \\ \frac{1}{NF(p)} & \text{else.} \end{cases}$$

Assuming enough input values are actually available (depending on the periods of the other PUs), after every $\lambda(p)$ loop iterations, all input values can be consumed and move instructions on the output executed. Buffer sizes of the PU will then return to their initial sizes. If we want to achieve this for all PUs at the same time, we need to take a multiple of the periods of all PUs. Therefore, we define $\lambda(\mathfrak{P}) = lcm(\lambda(p_1), ..., \lambda(p_k))$ for all k PUs. $\lambda(\mathfrak{P})$ will be called the *least common period* of $\mathfrak{P}$.

The previous discussion does however implicitly assume that PUs can actually fire at all. But starting from an empty configuration, no PU could ever fire (reasonably assuming $nc(\mathfrak{i}j) > 0$). Therefore, we will now discuss initial configurations. First, a notion of desirable initial configurations is defined. These are supposed to be *sufficient* to execute a given program. For that, we state that the initial configuration is sufficient if it is able to fire all of the firings set up during the course of the first least common period $\lambda(\mathfrak{P})$.

Definition 12. *A configuration $\mathcal{C}$ of $\mathfrak{A}$ is a sufficient initial configuration for $\mathfrak{P}$, iff after $\lambda(\mathfrak{P})$ many iterations, there is a run such that every PU p can fire $NF(p) \cdot \lambda(\mathfrak{P})$ many times.*

We define that *minimal* sufficient initial configurations are the ones having the least amount of values inside the $\mathcal{V}$ mapping of a configuration. $\mathcal{A}$ mappings are generally assumed to be empty in initial configurations (see discussion in Section 3.2). A sufficient initial configuration with $\mathcal{V}_1$ is therefore minimal, iff for all sufficient initial configurations with $\mathcal{V}_2$ it holds that $\sum_{b\in\mathcal{P}} |\mathcal{V}_1(b)| \leq \sum_{b\in\mathcal{P}} |\mathcal{V}_2(b)|$. Finding such a minimal sufficient initial configuration seems desirable, but we will now show that finding it is NP-complete. First, NP-hardness is shown in Lemma 5. Then, an algorithm in NP-time is given in Lemma 6.

Lemma 5. *Finding a minimal sufficient initial configuration for a balanced periodic branchless program $\mathfrak{P}$ is NP-hard.*

Proof. The proof will reduce the feedback arc set problem, which is known to be NP-complete [18], to the given problem. The problem is to find a minimal size feedback arc set E, such that if all edges from E are removed, the resulting graph is acyclic. But first, we do a polynomial-time preprocessing step: given a directed graph G, we remove all edges that are not part of a cycle. This can be done quite easily: if an edge $e = (u, v)$ starts in u and ends in v, search for a path from v to u. If there is one, it is part of a cycle. If not, remove it from the graph. This is fine, since it suffices to look at the feedback arc set on the resulting graph G': the edges removed are by definition not part of a cycle, so they do not have to be removed anyway to make the graph acyclic. Therefore, they can not appear in a minimal feedback arc set. Furthermore, for the sake of simplicity, all vertices without edges are removed. Obviously, these do not influence the result as well. An example can be seen in Figure 7: the gray vertices and edges are the ones removed by the preprocessing step.

Now we construct program $\mathfrak{P}(G')$ for a given (preprocessed) directed graph $G' = (V, E)$. Every vertex $v \in V$ becomes a PU v. For every vertex v its

incoming edges $in(v)$ become input buffers of PU v, and all outgoing edges $out(v)$ become output buffers of the PU. The firing table will then look as follows:

$in(v)_1$	...	$in(v)_n$	$out(v)_1$	...	$out(v)_m$
$x :: X$	...	$x :: X$	y	...	y

The function itself does not really matter, important to note is that it simply reads one value from each input, and writes one value to each output. For every edge $e = (u, v) \in E$ of the graph, we furthermore add an instruction to $\mathfrak{P}$. Remember that u and v are vertices, and that both have a corresponding buffer e as either an output buffer (u) or an input buffer (v). Thus, for every edge $e = (u, v) \in E$ we add instruction $i : u.e \rightarrow v.e$ to the program. Since the construction is clearly supposed to yield a periodic branchless move program, a `goto` is appended to the tail giving the following structure:

$1 : u_1.e_1 \rightarrow v_1.e_1$
$\cdots$
$m : u_m.e_m \rightarrow v_m.e_m$
$m + 1 : 1 \rightarrow \mathfrak{c}$

The resulting program is balanced by Definition 10: every input is the target of one move instruction, meaning it gets one value each period. And every input is consumed exactly once - since every PU fires once. One value is produced to each output, and one value is moved away from it.

Considering the PU firing rule, it should be noted that it can only fire if *all* of its corresponding input buffers store a token at some point. Thus, in order for the initial marking to be sufficient, every edge in the graph must transport a value in the SCAD program at some point - otherwise some vertex or PU could not fire. Now consider a cycle $e_1, ..., e_n, e_1$ in the graph. If we extract the lines of the move program, it will look as follows:

$1 : v_1.e_1 \rightarrow v_2.e_1$
$2 : v_2.e_2 \rightarrow v_3.e_2$
$\cdots$
$n : v_n.e_n \rightarrow v_1.e_n$

Clearly, values are being passed along the cycle. For every cycle, at least one token has to be added to any sufficient initial configuration. If not, there would be cyclic dependencies among the corresponding PUs: if none of $e_1, ..., e_n$ had a token, all of the vertices $v_1, ..., v_n$ could not fire, because each would be missing a token on its corresponding edge. No firing rule would be active. Thus, if the sufficient initial marking of $\mathfrak{P}$ would not add a token for any of the input or output buffers e_i, the PUs along the cycle could not fire - which is a contradiction to it being sufficient. Thus, it must contain a token on each cycle. But of course, an edge can be part of more than one cycle, thus one token can serve more than one cycle. So in every minimal sufficient configuration, every cycle will have at least one token. By the previous argument, if we remove the edges corresponding

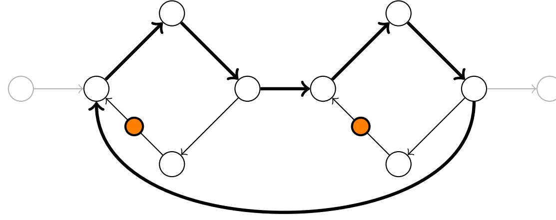


Figure 7: An apparent deadlock (cyclic dependencies), always creates a token-free cycle (thick lines). Vertices and edges not within cycles are ignored (gray).

to tokens in a minimal sufficient configuration, all cycles in the graph G' are broken. This makes it a feedback arc set. To show that it is minimal, we would have to argue that there is no smaller feedback arc set. In the next case we will argue that any minimal feedback arc set will also yield a sufficient configuration (that is executable), with the same amount of tokens in the configuration than edges removed. So if there was a smaller feedback arc set, there would also be a smaller minimal sufficient configuration - which is a contradiction, and will proof the minimality of the feedback arc set derived from the minimal sufficient configuration.

For every edge in a feedback arc set, we can add a token to the corresponding input buffer. It does not matter on which exact buffer it is placed, the token inside an output buffer can just be transported to the input buffer by the `DATAFLOW` rule. Again, minimality will follow from the fact that we can transform a valid feedback arc set to a valid sufficient configuration (and vice versa) using the same amount of tokens and edges removed. To show that this is a sufficient configuration at all however, we have to show that it is also really executable. There must be no deadlock, some vertex must be able to fire until all vertices have fired once. The idea to proof this goes as follows: if there is a deadlock, there must be cyclic dependencies of vertices waiting for the token of yet another cycle. But the cyclic dependencies themselves - which literally are edges without tokens - will form a cycle without a token, contradicting the assumption that every cycle has a token. Assume there is a deadlock. This means that no vertex (or PU, used interchangeably here) has a token on every incoming edge, and no data can be transported. This also directly implies that every vertex that does have a token on an incoming edge, must have another incoming edge without a token. If this was not the case, there was no deadlock: the corresponding PU could fire. Since no data can be transported, all tokens are also in their respective input buffers. By the previous discussion, it is known that on every cycle there is at least one token. Also remember that all edges are part of cycles. Choose any cycle $C_1 = c_{1,1}...c_{1,m_1}$ (of edges), and go to the token of that cycle c_{1,t_1}. Now since there is a deadlock, the corresponding PU v_1 must also have an incoming edge with no token on it. Let this (incoming) edge be c_{2,m_2}. If we follow this edge (in reverse), we are in another cycle

$C_2 = c_{2,1}...c_{2,m_2}$. Now again, this cycle must have at least one token. Now from c_{2,m_2} we travel in reverse through the cycle, until we are in the first vertex v_2 that has a token on an incoming edge c_{2,t_2}. Now since there is a deadlock, this v_2 must also have another incoming edge without a token, c_{3,t_3}. Notice that in order to get from v_1 to v_2 we did not traverse an edge with a token: c_{2,m_2} was by definition token-free, and we travelled in reverse $c_{2,m_2}...c_{2,t_2+1}$ to the first vertex that had an incoming edge with a token. This could already be the first one. But we did not have to use any edge with a token in order to reach vertex v_2 from v_1. Now since v_2 again has an incoming edge c_{3,t_3} without a token, the process can be repeated to reach a vertex v_3. And in fact, the process can be repeated indefinitely. But since the graph is finite, there has to be - after finitely many steps - a vertex v_i that has already been visited before. And then we are done: we have found a cycle from vertex v_i to itself again, using only token-free edges. This is a contradiction: as was discussed before, every cycle must get a token. The result is that the feedback arc set could not have been valid. The constructed cycle is also a cycle in G', even after the edges from the feedback arc set were removed (since tokens correspond to removed edges). Figure 7 illustrates the situation. Tokens are the orange circles on the edges. Apparently, a deadlock occurs here: both nodes that have tokens on their incoming edges can not fire because they have unresolved dependencies. But the cyclic dependency creates a cycle itself (thick lines).

Now does this only proof that there can be no deadlock in the beginning? No, since tokens only get passed along a cycle and do not get lost, the premise of the discussion will hold again after any amount of PU firings. Only when the move instructions of the first period are exhausted there can be a deadlock, but then every PU must have fired exactly once (since the program is balanced and has period $\lambda(\mathfrak{P}) = 1$) - which is sufficient. As long as move instructions of the period are to be executed, the argument will hold and proof that there can be no deadlock. This proofs that the derived configuration is sufficient. □

As a side remark, the construction used in the proof of Lemma 5 is not limited to periodic branchless move programs. Since the content of an output buffer is always sent to the same input buffer, the construction can directly be applied to static DPNs. Therefore, it also applies to petri nets. In petri nets, each cycle would correspond to a trap. The acyclic edges removed in the preprocessing step are source and sink places. To proof NP-completeness, membership in NP is now proven by giving an algorithm in NP-time.

Lemma 6. *Finding a minimal sufficient initial configuration for a balanced periodic branchless program $\mathfrak{P}$ is in NP.*

Proof. The following algorithm suffices to show this: let n be the amount of move instructions in a periodic branchless program $\mathfrak{P}$. Since it is balanced, no more tokens can be consumed within a period than $\lambda(\mathfrak{P}) \cdot n$. Therefore, no more than $\lambda(\mathfrak{P}) \cdot n$ tokens are needed in the initial configuration. Now guess an initial configuration $\mathcal{C}$ using 1 token. To check if it is sufficient, use the ASAP schedule and operational semantics to check whether the required amount of firings can

be performed (see Lemma 7 for more detail). If not, increase the amount of tokens and repeat. Because $\lambda(\mathfrak{P}) \cdot n$ surely suffices, the algorithm will terminate after polynomially many iterations. □

Finding a minimal sufficient initial configuration is therefore hard. But fortunately, checking whether a given configuration is a sufficient initial configuration is comparatrively easy - it can be done in polynomial-time:

Lemma 7. *Whether a configuration $\mathcal{C}$ is a sufficient initial configuration for balanced periodic branchless program $\mathfrak{P}$ is decidable in polynomial-time.*

Proof. Use the precedence as defined by the canonical schedule to execute the least common period of $\mathfrak{P}$: fire operations as soon as possible, until their limit of $NF(p) \cdot \lambda(\mathfrak{P})$ is reached. Execute all possible data moves. If enough firings can be performed, the configuration is sufficient. Clearly, otherwise there is no way to reach the amount of firings: the program has terminated, and the canonical schedule will yield the same result as any other (Lemma 3). Because as many tokens are produced to every output buffer as move instructions using it as a source have been added (Definition 10), output buffer sizes are the same as before. The same holds for input buffers: as many have been transported to them, as have been consumed (Definition 10). Therefore, the resulting configuration $\mathcal{C}'$ after executing the canonical schedule also has the same buffer sizes in every buffer as $\mathcal{C}$. □

After defining sufficient initial configurations, deciding boundedness can finally be discussed. As will be shown in the following lemma, the definition of balance actually suffices to do that if the program starts from a sufficient initial configuration. And since balance can clearly be decided in polynomial-time for a periodic branchless move program, it then directly follows that for a given initial configuration and program boundedness is decidable in polynomial-time.

Lemma 8. *A periodic branchless move program $\mathfrak{P}$ has a bounded run $\mathcal{R}$ starting from a sufficient initial configuration, iff it is balanced.*

Proof. If $\mathfrak{P}$ is balanced, we will show that there is a bounded run. After $i = (\lambda(\mathfrak{P}) \cdot j)$ (for some $i, j \in \mathbb{N}$) many iterations, the number of firings of any PU will surely be a natural number: for every $p \in \mathcal{P}$, $\lambda(\mathfrak{P}) \cdot j$ is a multiple of $\lambda(p)$. By Definition 11 either the number of firings is a natural number in itself, in which case $NF(p) \cdot i$ is, too. Or it is not, and the number of firings after i iterations $(\frac{1}{NF(p)} \cdot j') \cdot NF(p) = j'$ becomes a natural number j'. Now we will show that starting from a sufficient initial configuration $\mathcal{C}$, we can create a bounded run. After $\lambda(\mathfrak{P}) \cdot 1 = i$ iterations we know that each PU p can fire $NF(p) \cdot i$ times by the sufficient initial configuration in some run $\mathcal{R}$. Using the run from the sufficient initial configuration we fire them $NF(p) \cdot i$ many times. As outlined by Lemma 7 the resulting configuration $\mathcal{C}'$ will have the same buffer sizes as $\mathcal{C}$. Since only buffer sizes are relevant for how many tokens a PU consumes and produces, the argument will hold again after any least common period. Therefore, for every $i = (\lambda(\mathfrak{P}) \cdot j)$ (for some $i, j \in \mathbb{N}$), the buffer sizes will be

the same as in the initial configuration. Also, since there are only finitely many steps in-between, as well as production and consumption in the period iteself is finite, the described run is bounded.

For the other case, proof by contraposition is used. Assume $\mathfrak{P}$ is unbalanced. If there was an n bounding any run $\mathcal{R}$, a contradiction would occur in any case. The cases are from the definition of balance (Definition 10). If a program is unbalanced, there can be three reasons for it:
Case $pf(\mathfrak{o}i) < m(\mathfrak{o}i)$: If $pf(\mathfrak{o}i) < m(\mathfrak{o}i)$, after j iterations the address buffer will contain $j \cdot (m(\mathfrak{o}i) - pf(\mathfrak{o}i))$ unused move instructions. As $m(\mathfrak{o}i) - pf(\mathfrak{o}i) > 0$ and j can be chosen freely, any buffer bound n can be broken after enough iterations.
Case $pf(\mathfrak{o}i) > m(\mathfrak{o}i)$: If $pf(\mathfrak{o}i) > m(\mathfrak{o}i)$, after j iterations the value buffer will contain $j \cdot (pf(\mathfrak{o}i) - m(\mathfrak{o}i))$ values. As $pf(\mathfrak{o}i) - m(\mathfrak{o}i) > 0$ and j can be chosen freely, any buffer bound n can be broken after enough iterations.
Case $\frac{m(\mathfrak{i}j_1)}{nc(\mathfrak{i}j_1)} > \frac{m(\mathfrak{i}j_2)}{nc(\mathfrak{i}j_2)}$ (w.l.o.g.) for a PU p: PU p can only fire $j \cdot \frac{m(\mathfrak{i}j_2)}{nc(\mathfrak{i}j_2)}$ many times after j iterations. $\delta = \frac{m(\mathfrak{i}j_1)}{nc(\mathfrak{i}j_1)} - \frac{m(\mathfrak{i}j_2)}{nc(\mathfrak{i}j_2)} > 0$. Therefore, $j \cdot \delta \cdot nc(\mathfrak{i}j_1)$ leftovers are not consumed after j iterations. These can be leftover values or move instructions in $\mathfrak{i}j_1$. In the worst case (minimal buffer sizes), leftovers are split evenly between values and move instructions ($\frac{j \cdot \delta \cdot nc(\mathfrak{i}j_1)}{2}$ each) - clearly still making it possible to break any buffer bound n. □

An example of the given procedure to decide boundedness will now be given. Consider the following (static) firing rules of two PUs:

Firing rule PU 1.

in_1	in_2	out_1
$x :: X$	$y :: Y$	$(x + y)$

Firing rule PU 2.

in_3	out_2
$x :: X$	$x.x$

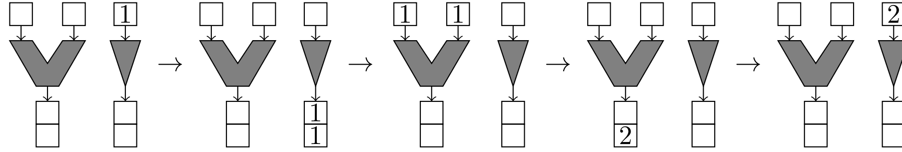
Clearly, $np(out_2) = 2$. Also, $nc(in_1) = nc(in_2) = nc(in_3) = np(out_1) = 1$. Now consider the following periodic branchless program:

$1 : out_1 \to in_3$
$2 : out_2 \to in_1$
$3 : out_2 \to in_2$
$4 : out_2 \to in_1$
$5 : 1 \to \mathfrak{c}$

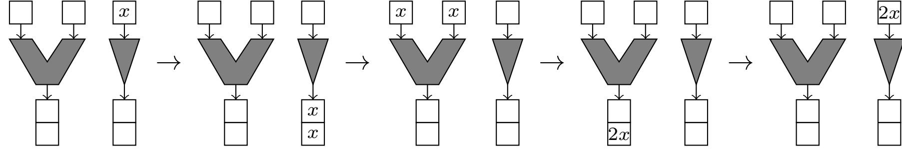
Balance equations for in_3 are fine: $nc(in_3)$ is the target of one value, this sets up 1 firing of PU 2, which then consumes the value. However, on out_2 there is a problem: one firing of PU 2 only produces 2 values, but there are 3 move instructions with out_2 as the source. And indeed this program would not be bounded for a sufficient initial configuration: every iteration, one additional move instruction in the address buffer out_2 can not be executed. After i iterations, i move instructions are left over. Therefore, infinitely many move instructions would amass in the address buffer of out_2. But a simple change can fix the program: by removing instruction $4 : out_2 \to in_1$ the program becomes balanced.

$1 : out_1 \rightarrow in_3$
$2 : out_2 \rightarrow in_1$
$3 : out_2 \rightarrow in_2$
$4 : 1 \rightarrow \mathfrak{c}$

All balance equations are in check and this program does have a bounded run on a sufficient initial configuration. But what does this program actually calculate? To answer this question, a intial configuration must be given. Consider the following initial configuration and execution of a period:



While this can already give an insight into what the program does, we can go one step further and replace all initial values with variables, and write down the calculations symbolically:



Because the size of the resulting configuration is always the same as the initial configuration (Lemma 7), we can write down an expression for each of the initial token that expresses the value after the next period. In this case this would simply be $y = 2x$. From Lemma 3 we also know that executing the run in this ASAP manner suffices to capture the result of all possible runs. This idea will be further developed in the next section.

5.2 Symbolic Period Summary

The goal of this section is to summarize execution of a least common period as a set of symbolic expressions, or to be more precise by a set of expressions consisting of functions presented in Definition 13. First, we will assume that the least common period is a single iteration of the loop itself ($\lambda(\mathfrak{P}) = 1$): we can always do this by unrolling the loop body of the periodic branchless move program $\lambda(\mathfrak{P})$ many times. Next, a given behaviour of PUs as a function is assumed. A PU p is given with input buffers $\mathfrak{i}1, ..., \mathfrak{i}m$ and output buffers $\mathfrak{o}1 ... \mathfrak{o}n$. The function of the PU will then be split into component functions for all of the outputs. Thus, there is a function $f_1, ..., f_n$ for every output. Since more than one token can be produced to each output, these are split again for every token $1, ..., n_i$ produced to the output i with $f_{i,1}, ..., f_{i,n_i}$. Clearly, n_i is the same for every firing since production of PU functions is static. $f_{i,1}$ will be the first token produced to the output, and f_{i,n_i} the last one. A PU function can also consume more than one token from each input. This is unrolled into the parameters of the function, finally giving the unrolled component function:

Definition 13. *Unrolled component function for the x-th value of output buffer i, assuming the stated inputs in the parameters.*

$$f_{i,x}(z_{1,1}, ..., z_{1,m_1}, z_{2,1}..., z_{1,m_2}, ..., z_{m,m_m})$$

The parameter $z_{j,y}$ represents the y-th value of input buffer j. The first value is the one consumed first, at the head of the buffer.

But when translating a periodic branchless move program into a set of expressions, how many expressions are actually needed? In Lemma 7 it was shown that after every least common period executed, the resulting configuration $\mathcal{C}'$ has the same sizes in all buffers as the sufficient initial configuration $\mathcal{C}$. So given any buffer z, assumed to be filled with $z_1, ..., z_d$ values in a configuration $\mathcal{C}$ as large as the initial, we want to know expressions $Z'_1, ..., Z'_d$ (which may depend on *all* values of $\mathcal{C}$ and make use of unrolled component functions) calculating their next values $z'_1, ..., z'_d$ after another least common period, or one iteration, has passed. In order to find out their values, the least common period will be symbolically executed. This will tell us how the tokens travel and are used within an iteration. Datatype $\mathcal{D}$ of $\mathfrak{A}$ is changed to the set of expressions on symbolic function symbols. The sufficient initial configuration is replaced with $\mathcal{C}_{\text{sym}}$, in which actual values in $\mathcal{V}$ are replaced with unique symbols: $\mathcal{V}_{\text{sym}}(z)(j) = z_j$ if $\mathcal{V}(z)(j) \neq \perp$ where z_j is a unique symbol for this buffer slot. The unrolled component functions $f_{i,x}$ of every PU p will be replaced with a new component function $g_{i,x}$. First, for every unrolled component function $f_{i,x}$, a symbolic function symbol $F_{i,x}$ is added to the domain. Then, instead of producing the result of the actual unrolled component function as the x-th value to output buffer i, $g_{i,x}$ will substitute the expressions given as parameters into the symbolic component function $F_{i,x}$ itself. Thus, instead of actually calculating the function value, it returns a new expression using the corresponding symbolic function symbol, essentially saving which calculation would have taken place:

$$g_{i,x}(z_{1,1}, ..., z_{m,m_m}) := [F_{i,x}(Z_{1,1}, ..., Z_{m,m_m})]^{z_{i,j}}_{Z_{i,j}}$$

It should be obvious that, since only buffer sizes matter and are unchanged, $\mathcal{C}_{\text{sym}}$ is still a sufficient initial configuration and can therefore yield a $\mathcal{C}'_{\text{sym}}$ with the same buffer sizes after the period (Definition 12). By construction, it is also easy to see that by replacing the symbols $F_{i,x}$ with the original functions $f_{i,x}$ and values inside the expressions to those corresponding to $\mathcal{C}$, will yield $\mathcal{C}'$ from $\mathcal{C}'_{\text{sym}}$ again. But furthermore, for every buffer z and every value z_j in the initial configuration, we also get the desired symbolic expression from the value part of C'_{sym}. Every value of the resulting configuration is a corresponding symbolic expression for its necessary calculations and dependencies.

$$Z'_j = \mathcal{V}'_{\text{sym}}(z)(j)$$

To specify, actually calculating the resulting value z'_j from expression Z'_j can be done using the following method: all function symbols $F_{i,x}$ are substituted back to actual functions $f_{i,x}$, while symbolic values z_j of all buffers z and positions j

are substituted with actual values d_j of the desired configuration. Using these substitutions, all functions can be applied and the value calculated.

$$z'_j = [Z'_j]^{f_{i,x},d_j}_{F_{i,x},z_j}$$

But what are these expressions good for? So in the general case, it seems difficult to make the point that these symbolic expressions will be easier to read for humans, or provide any computational benefits. That this can be the case at all can be seen in the example given in the previous section. However, an expression Z'_j may have dependencies from any other buffer y and any position in that buffer of the previous configuration. But this is not necessarily the case: if dependencies are limited and only a specific buffer position is of interest, parts of the configuration that this expression does not depend on does not have to be calculated. And furthermore, simulating SCAD move instructions can be omitted. Instead of simulating how moves travel through the machine, the operations themselves can directly be calculated. Also, simplifications may be possible. In any case we have seen that treating the least common period as a single operation has one big advantage: the resulting configuration has the same sizes as the initial one. At this point, we can reconsider input and output to the SCAD architecture itself. In a period, the architecture will always read the same amount of input values m and write the same amount of output values n. To the expressions input variables $i_1, ..., i_m$ and output variables $o_1, ..., o_n$ can thus be added. Would this have influenced the previous results on boundedness and the semantics of a period? The definition of boundedness explicitly ignored the content of input and output buffers, which is why they make no difference for boundedness itself. The amount of values produced and consumed is static, which is why adding or removing input and output buffers from all firing rules does not influence previously given results, if it is assumed that input buffers can always be consumed and output buffers can always be written to. Moreover, output buffers may not be read again, which is why their content can not influence any following states. The values of input buffers may however influence the values of other buffers. Since input values are directly controlled by the environment while configurations are not, we will now define a function *next* to denote this: For a fixed configuration $\mathcal{C}$ and input vector σ of length m, assume that configuration $\mathcal{C}'$ is the result of firing a least common period from $\mathcal{C}$ and σ. Then, we can denote the following:

Definition 14. *Least common period effect.*

$$next(\mathcal{C}, \sigma) = \mathcal{C}'$$

Clearly, the expressions derived by symbolic execution can be used to calculate *next*. In the next section, these definitions will be used to derive a FSM for any balanced periodic branchless move program on a finite datatype.

5.3 Transformation to Finite State Machines

For a balanced periodic branchless move program $\mathfrak{P}$ on a finite datatype $\mathcal{D}$ a finite state machine $(\mathcal{D}, \mathcal{C}, \mathcal{C}_0, \rightarrow, F)$ can then be created quite easily. The states of the finite state machine are configurations of the SCAD architecture. Starting from a sufficient initial configuration $\mathcal{C}_0$ we know that, after executing any amount of least common periods, the resulting configuration $\mathcal{C}'$ will have the same buffer sizes as $\mathcal{C}_0$. Since we have just required that the datatype $\mathcal{D}$ shall be finite, we know there can only be finitely many configurations when always executing least common periods as a whole. The transition $\rightarrow$ contains $C \xrightarrow{\sigma} C'$ (where σ is the input vector of $\mathcal{D}$), iff $next(\mathcal{C}, \sigma) = \mathcal{C}'$. Since it is known that every least common period will write the same amount of output values to the output buffers, a similar output vector can be added to the FSM. Furthermore, because for every possible input vector σ there is only one successor, the FSM is potentially deterministic, if there is only one initial state. This is indeed the case, because we use the given sufficient initial state as the initial state of the FSM. Not all possible configurations $\mathcal{C}$ of the SCAD architecture must therefore be considered, since only reachable states from the initial state are of importance. The FSM could then be used for model-checking. It is needless to say that there is an exponential overhead involved when transforming move programs to FSMs using the aforementioned method.

6 Bounded Move Programs

In this section, an attempt will be made to understand the behaviour of more move programs beyond the restrictions of the previous section. When doing so, some simplifications made possible by peridioc branchless move programs completely vanish. First, some equivalences for SCAD configurations are introduced. Then, using these, an attempt will be made to translate more move programs into another, hopefully more useful form.

The main idea is to allow arbitrary branches while keeping the other restrictions on boundedness and static consumption. One key observation is that assignments to the CU $src \rightarrow \mathfrak{c}$ directly correspond to `goto` instructions. The scheme described in this section will try to reduce the rest of the program to basic blocks that only make use of the unrolled component functions, and replace the CU assignments with `goto` instructions. All FIFO buffer storage is replaced with a finite amount of registers. The hope is that these programs can then be dealt with using existing methods for `cmd` programs. Therefore, only the SCAD specific problem of dealing with varying buffer contents when entering a basic blocks is considered, while leaving the rest of the analysis up to other methods.

6.1 Configuration Layouts

The idea of this section is to try to capture properties that made periodic branchless programs easy to handle in more general terms: the goal is to get rid of the control flow restriction, while still keeping some of the simplicity that came with it. One key advantage of periodic branchless programs was that after executing a least common period, all buffer sizes were back to the ones from the initial configuration. Therefore, a notion of buffer size equivalence for $\mathcal{V}$ is introduced. We will however also require $\mathcal{A}$ buffers to be equivalent. The reason for this will be elaborated in the following.

Definition 15. *Layout equivalence. Two configurations* $(\mathfrak{P}_1, pc_1, st_1, \mathcal{A}_1, \mathcal{V}_1) = \mathcal{C}_1$ *and* $(\mathfrak{P}_2, pc_2, st_2, \mathcal{A}_2, \mathcal{V}_2) = \mathcal{C}_2$ *of the same architecture* $\mathfrak{A}$ *are layout-equivalent, iff for all buffers* $z \in \mathcal{P} : \mathcal{A}_1(z) = \mathcal{A}_2(z)$ *and* $|\mathcal{V}_1(z)| = |\mathcal{V}_2(z)|$*.*

Another advantage was that after every iteration, the peridioc branchless program ended up in the same control state again. Thus, a notion of control state equivalence is introduced as well.

Definition 16. *Control equivalence. Two configurations* $(\mathfrak{P}_1, pc_1, st_1, \mathcal{A}_1, \mathcal{V}_1) = \mathcal{C}_1$ *and* $(\mathfrak{P}_2, pc_2, st_2, \mathcal{A}_2, \mathcal{V}_2) = \mathcal{C}_2$ *of the same architecture* $\mathfrak{A}$ *are control-equivalent, iff* $\mathfrak{P}_1 = \mathfrak{P}_2, pc_1 = pc_2$ *and* $st_1 = st_2$*.*

Furthermore, address buffers became empty after every period by design, because all programs were balanced. But how bad is having values in $\mathcal{A}$, really? Note that values in address buffers do actually matter, even with static consumption and if only symbolic execution is considered. Actual values in value buffers did not matter for how a program with static consumption is symbolically executed, since values are consumed and produced regardless of what they

are specifically. However, if two configurations have the same buffer sizes, but differing values in $\mathcal{A}$ buffers, they may produce different expressions through symbolic execution: because the same values may end up in entirely different buffers, their semantics may be entirely different. Consider a configuration $\mathcal{C}$ with n values in output address buffers. Each of these may take any value from $in(\mathcal{P})$. Therefore, there are potentially $|in(\mathcal{P})|^n$ distinct behaviours or symbolic expression systems resulting from the same buffer size layout. This is why we required $\mathcal{A}$ buffers to be fully equivalent in Definition 15. Another reasonable restriction would be to require that $\mathcal{A}$ buffers become empty after every basic block, which would mimic the behaviour of periodic branchless move programs. The restriction of Definition 15 is however clearly more general. Using the previous equivalences the following simple lemma can now be derived.

Lemma 9. *If two configurations $\mathcal{C}_1$ and $\mathcal{C}_2$ are layout-equivalent and control-equivalent, symbolic execution until the first branch will yield equivalent symbolic expressions.*

Proof. Because both configurations are layout-equivalent, $\mathcal{C}_1$ and $\mathcal{C}_2$ will yield the same $\mathcal{V}$ in $\mathcal{C}_{\text{sym}}$. This is because every literal value in the configurations will just get a unique symbol anyway - making the value itself irrelevant. $\mathcal{A}_1$ and $\mathcal{A}_2$ are equivalent by definition. And since $\mathcal{C}_1$ and $\mathcal{C}_2$ are control-equivalent as well, the rest of corresponding symbolic configurations is equivalent, too. Therefore, symbolic execution will start from the exact same configuration in both cases, producing equivalent symbolic expressions until the first branch. The branches themselves may depend on actual values. □

In order to reason about bounded move programs, our definition of boundedness for runs is lifted to programs.

Definition 17. *A program $\mathfrak{P}$ is called bounded for an initial configuration $\mathcal{C}$, iff for any sequence of inputs there exists a bounded run.*

It should be clear that even with static consumption, having no restrictions on the program itself makes SCAD architectures Turing-complete. Thus, there is no hope that boundedness is actually decidable. However, we will now just assume that all given programs are bounded. Since we will explore the set of all layout- or control-distinct states, it is necessary to assume boundedness for the sake of termination.

6.2 Towards Analysis of Bounded Move Programs

This section will introduce a scheme to translate bounded move programs on SCAD architectures with static consumption to a `cmd` program (that is able to call unrolled component functions of Definition 13). The idea is as follows: from Lemma 9 it is known that starting from layout- and control-equivalent configurations, the same set of expressions is calculated. From the resulting configuration this process can be repeated without actually calculating values, because for the symbolic execution procedure we only need to know buffer sizes.

We replace instructions sending values to the CU with a `goto` and assign each buffer slot in configurations to a register. Since boundedness of the program was required, only a finite amount of registers will be necessary. Thus, we explore the reachable layout and control states of the program and translate each distinct one to a symbolic basic block in the `cmd` programs. If a layout- and control-equivalent configuration has already been translated to a basic block, it will be reused. For bounded programs, this method will then terminate since there can only be finitely many distinct configurations. However, we need to require the existence of following subroutines: for every branch of a SCAD program, a Δ function that will translate SCAD labels to `cmd` labels, as well as a set L that can predict possible branch targets. For programs resulting from structured programming, it should be possible to implement these. Consider the following example: if the branch target for the `goto` instruction has to be derived for $l_1 \cdot b + l_2 \cdot \neg b$, where l_1 and l_2 are literal labels and b is a boolean, then Δ can simply replace the literals l_1 and l_2 with their `cmd` counterparts $\delta(l_1)$ and $\delta(l_2)$. Thus, this would give $\Delta(l_1 \cdot b + l_2 \cdot \neg b) = \delta(l_1) \cdot b + \delta(l_2) \cdot \neg b$.

A `cmd` version of a given bounded move program $\mathfrak{P}$ with initial configuration $\mathcal{C}_0$ can be created as follows. It should be clear from the previous section, that a given basic block of a SCAD program may have to be translated into several ones in the `cmd` version: depending on the layout when starting the basic block, different symbolic expressions will emerge from the same SCAD instructions. Remember that $\mathcal{C} = (\mathfrak{P}, pc, st, \mathcal{A}, \mathcal{V})$. Starting from the initial configuration $\mathcal{C}_0$, basic blocks are translated as follows:

The initial configuration of this basic block is called $\mathcal{C}$. Before creating a new basic block, we check whether there is a basic block tagged with a layout and control equivalent configuration $\mathcal{C}_j$. If so, its initial `cmd` label l_j is simply reused for this basic block. No new basic block has to be created. This is fine, since it was shown in Lemma 9 that they result in the same expression system anyway. But if not, the basic block must be created. We start by symbolically executing from $\mathcal{C}$ on until the first branch $src \to \mathfrak{c}$. The resulting expression system will be called Z. And $src \to \mathfrak{c}$ will be translated into `goto` $V_{\text{sym}}(src)(i)$ (where i is the last entry of src, the value that is sent to the CU). Now add the following piece of code for the basic block in the `cmd` program:

$$l : next(R) \leftarrow Z;$$
$$R \leftarrow next(R);$$
$$\texttt{goto}\ \Delta(V_{\mathsf{sym}}(src)(i))$$

The basic block is tagged with its initial configuration $\mathcal{C}$. $next(R) \leftarrow Z$ is supposed to mean the following: for every buffer z_i and slot j, i.e. $z_{i,j}$, used in $\mathcal{V}_{\text{sym}}$, there is an expression $Z_{i,j}$. We create a register $r_{i,j}$ for every buffer slot in every buffer ever used in the program. And $next(R) \leftarrow Z$ means that we assign $next(r_{i,j}) \leftarrow Z_{i,j}$ - the corresponding expression for that buffer slot. Clearly, expressions Z also need to use the corresponding register values $r_{i,j}$ (instead of the ones symbolically inserted) in the goto program. This is also why we then advance $R \leftarrow next(R)$ after they have been read - before finally writing them

to R, all expressions must have been executed. Otherwise, they may depend on previous values that have already been overwritten. l is simply a new unique `cmd` label. The Δ function is supposed to map the *pc* labels of the SCAD program into labels of the `cmd` program. However, how to find out which labels of the SCAD program are even possible to reach are left for specific implementations. It is however reasonable to assume that for many applications, this may just be a choice of several literal labels encoded in the program. The problem is however no different for `cmd` programs, which is why we omit further discussion. For now, assume the result of this method is a set of labels L to which - depending on specific state and input - may be jumped to. In order to find out what the corresponding labels and basic blocks of the `cmd` program are, the basic blocks may even have to be created first. Therefore, the method is called recursively on the configuration $(\mathfrak{P}, l, 0, \mathcal{A}_{\text{sym}}, \mathcal{V}_{\text{sym}})$ for every label $l \in L$. These basic blocks are assumed to be reachable and therefore need to be translated. Also, their `cmd` labels may be necessary for the final code of the `goto` instruction of the current basic block. The ideas are summarized in the following pseudocode:

```
translate_basic_block(&P, &B, &δ, C) {
    // First, check whether equivalent basic block has
    // been translated.
    if(B includes layout- and control equivalent C_j)
        δ(C) := δ(C_j)
        return;

    // If not, basic block is added and executed.
    δ(C) := l with ∀C_j : δ(C_j) ≠ l // generate unique label l
    B := B ∪ {C}
    C_sym := symbolic_execute(C)

    // Let L be reachable labels from following branch,
    // recursively call method.
    // Labels δ(C_j) of possible branch targets C_j
    // may be required before adding code.
    for l ∈ L {
        translate_basic_block(P,B,δ,(𝔓,l,0,𝒜_sym,𝒱_sym))
    }

    // Then, using the set of symbolic expressions Z
    // from C_sym, add code to P.
    P << δ(C) : next(R) ← Z;
    P << R ← next(R);
    // Assuming src → 𝔠 is the following branch,
    // replace all labels using method Δ.
    P << goto Δ(𝒱_sym(src)(i))
}
```

Note that since function Δ and set L are not specified, this is not a complete method, yet. A mapping of SCAD configurations to `cmd` labels is however kept in δ. How to find out labels L and how to translate labels of SCAD programs within expressions to the labels in δ is left to specific implementations. Moreover, a proof of correctness and evaluation for specific architectures are clearly missing from this discussion. They were however deemed to be out of scope for this paper and the presented method should be viewed as a possible scheme to tackle these kinds of problems. Considering optimization, the creation of some basic blocks could be omitted: if there are symmetries in the created expressions (e.g. commutativity $x + y = y + x$), two layouts may after all lead to the same results. Thus, the condition that the initial configurations have to be control and layout equivalent could potentially be softened up by weaker conditions on the resulting expressions.

The idea of traversing all possible relevant states can also be observed in other strongly related problems. Reachability and coverability problems in petri nets, which are tightly connected to static DPNs, can be solved using a similar approach. Furthermore, scheduling of dynamic dataflow graphs also has comparable solutions.

7 Conclusions, Applications and Future Work

In this section, conclusions, applications and future work are given. First, the relation between SCAD architectures and DPNs observed in this paper is discussed. Then, tool proposals based on some of the results are made. And finally, open problems and possible future work are outlined.

7.1 DPNs and SCAD Architectures

One important outcome of this paper was showing a tight connection between DPNs and SCAD architectures. Clearly, there are some significant differences: on one hand, SCAD architectures have a program counter that limit their expressiveness compared to DPNs. On the other hand, more liberal communication between buffers is possible using the DTN, which enhances expressiveness. It was however shown that both of these differences can be bridged using simple constructions: the program counter by applying the construction in Section 4.4 and communication with the model in Section 4.1. Therefore, both models of computation can easily be expressed by the other. This gave us some meaningful results: first of all, known PU restrictions of DPNs such as the Kahn requirements can be directly applied to SCAD to give provably deterministic and latency-insensitive architectures. Kahn SCAD architectures form a reasonable, reliable subset of SCAD architectures. But even more results for PU restrictions (e.g. endochronous DPNs) could be applied to SCAD architectures. In this paper, we furthermore applied static consumption and production of static DPNs to SCAD architectures. Section 5 utilized periodic schedules of static DPNs to give a subset of SCAD move programs that can be easily analyzed. And Section 6 used an approach similar to the scheduling of dynamic dataflow graph and reachability in petri nets to analyze bounded move programs. To summarize, it should be clear that there are very beneficial connections between SCAD architectures and DPNs.

7.2 Tool Proposals

Many of the presented results can be used as a foundation to create different kinds of tools for SCAD architectures and programs.

Simulator. A SCAD simulator can directly be implemented using the operational semantics given in Section 3. While SCAD simulators already exist for specific PU functions, using the given operational semantics may have some advantages: As long as PU functions fulfill the Kahn requirements, the simulator based on operational semantics will have provable deterministic behaviour. In that case, only a single run has to be simulated to cover the results of all possible behaviours. But if PU functions do not fulfill Kahn requirements, the operational semantics can also be used to exhaustively simulate all possible behaviours.

Model-Checker. As outlined in Section 5, periodic branchless move programs on static SCAD architectures can be translated into a set of expressions. The method presented could be implemented into a tool to automatically present a different version of a program for a given initial configuration layout. This may have great benefits for the readability of a given program. Furthermore, periodic branchless move programs on static SCAD architectures can also be translated into a FSM. This can then obviously directly be used to do model-checking. The tool could be built on the symbolic translator, as least common periods are seen as atomic operations in the FSM and computational benefits may arise from first translating the period to an expression system.

`cmd` Translator. A proposal is made in Section 6 to create a disassembler translating SCAD programs to `cmd` programs. While the proposal is missing important details, it could be used as a starting point for a complete method for specific architectures. This could then assist in removing some of the complications introduced by FIFO buffers in SCAD programs. However, a proper evaluation of the proposed methodology is left for future work.

7.3 Open Problems

This paper also left many questions unanswered and problems unsolved. First, the view taken on SCAD in this paper has been more or less that of a classical processor architecture: no matter the program and execution delays, the result should be deterministic. There is a different way to view this, though. If HW/SW co-design is considered, programs may be a fixed entity as well. Taking this view, even architectures that generally have non-deterministic behaviours may be deterministic for a fixed program and therefore be feasible. Moreover, the DPN construction used in Section 4 to model SCAD architectures could not handle fixed-size patterns. All following results then depended on PUs not using them. How to handle fixed-size patterns has not been explored by this paper. The model furthermore dependeded on SCAD architectures not being able to read input values after the first $\perp$ symbol in the input buffer. Thus, proofing the determinacy of out-of-order implementations inside PUs - if they can read those values - is not possible within the boundaries of these definitions. Another direction that has not yet been explored is that DPNs have a special form of denotational semantics using fixed-points [11]. Even though SCAD architectures were directly mapped to these kinds of DPNs, this paper did not examine the potential of using denotational semantics. Decidability of boundedness as given in Section 6.1 has also not been explored. Since unrestricted programs are Turing-complete, there is no hope for the general case, though. It would however be interesting to have a subset of programs for which boundedness is decidable, but which are also more expressive than the periodic branchless programs. A strong motivation for this is that SCAD programs face the problem of boundedness within the processor itself - while register machine programs are bounded within the processor by definition.

List of Figures

1 Example of non-deterministic SCAD firing rules 6
2 Example of SCAD firing rule and program 7
3 Basic SCAD Architecture 10
4 DPN model for SCAD architectures 20
5 Example of fixed-size SCAD firing rules 21
6 DPN buffer transformed to SCAD buffers 28
7 Apparent deadlock example 34

References

[1] F. Yazdanpanah, C. Alvarez-Martinez, D. Jimenez-Gonzalez, and Y. Etsion, "Hybrid dataflow/von-Neumann architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1489–1509, June 2014.

[2] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe, "Space-time scheduling of instruction-level parallelism on a raw machine," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, D. Bhandarkar and A. Agarwal, Eds. San Jose, California, USA: ACM, 1998, pp. 46–57.

[3] D. Burger, S. Keckler, K. McKinley, M. Dahlin, L. John, C. Lin, C. Moore, J. Burrill, R. McDonald, and W. Yoder, "Scaling to the end of silicon with EDGE architectures," *IEEE Computer*, vol. 37, no. 7, pp. 44–55, July 2004.

[4] H. Corporaal, "TTAs: Missing the ILP complexity wall," *Journal of Systems Architecture*, vol. 45, no. 12-13, pp. 949–973, June 1999.

[5] M. Özsoy, Y. Koçberber, M. Kayaalp, and O. Ergin, "Dynamic register file partitioning in superscalar microprocessors for energy efficiency," in *International Conference on Computer Design (ICCD)*. Amsterdam, The Netherlands: IEEE Computer Society, 2010, pp. 515–520.

[6] A. Bhagyanath and K. Schneider, "Exploring the potential of instruction-level parallelism of exposed datapath architectures with buffered processing units," in *Application of Concurrency to System Design (ACSD)*, A. Legay and K. Schneider, Eds. Zaragoza, Spain: IEEE Computer Society, 2017, pp. 106–115.

[7] M. Anders, A. Bhagyanath, and K. Schneider, "On memory optimal code generation for exposed datapath architectures with buffered processing units," in *Application of Concurrency to System Design (ACSD)*. Bratislava, Slovakia: IEEE Computer Society, 2018.

[8] G. Buttazzo, *Hard Real-Time Computing Systems*, 3rd ed., ser. 24. Springer, 2011.

[9] A. Bhagyanath and K. Schneider, "Optimal compilation for exposed datapath architectures with buffered processing units by SAT solvers," in *Formal Methods and Models for Codesign (MEMOCODE)*, E. Leonard and K. Schneider, Eds. Kanpur, India: IEEE Computer Society, 2016, pp. 143–152.

[10] G. Kahn, "The semantics of a simple language for parallel programming," in *Information Processing*, J. Rosenfeld, Ed. Stockholm, Sweden: North-Holland, 1974, pp. 471–475.

[11] K. Schneider, "The synchronous programming language Quartz," Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, Internal Report 375, December 2009.

[12] P. Panangaden, V. Shanbhogue, and E. Stark, "Stability and sequentiality in dataflow networks," in *International Colloquium on Automata, Languages and Programming (ICALP)*, ser. LNCS, M. Paterson, Ed., vol. 443. Warwick University, England: Springer, 1990, pp. 308–321.

[13] E. Lee and T. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, May 1995.

[14] S. Bhattacharyya and E. Lee, "Looped schedules for dataflow descriptions of multirate signal processing algorithms," *Formal Methods in System Design (FMSD)*, vol. 5, no. 3, pp. 183–205, December 1994.

[15] A. Bhagyanath and K. Schneider, "Exploring different execution paradigms in exposed datapath architectures with buffered processing units," in *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, Y. Patt and S. Nandy, Eds. Samos, Greece: IEEE Computer Society, 2017, pp. 1–10.

[16] A. Bhagyanath, T. Jain, and K. Schneider, "Towards code generation for the synchronous control asynchronous dataflow (SCAD) architectures," in *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, R. Wimmer, Ed. Freiburg, Germany: University of Freiburg, 2016, pp. 77–88.

[17] R. Back and H. Mannila, "A refinement of Kahn's semantics to handle non-determinism and communication (extended abstract)," in *Principles of Distributed Computing (PODC)*. ACM, 1982, pp. 111–120.

[18] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman, 1979.