Adaptive streaming applications : analysis and implementation models
Zhai, J.T.

Citation
Zhai, J. T. (2015, May 13). *Adaptive streaming applications : analysis and implementation models*. Retrieved from https://hdl.handle.net/1887/32963

Version:	Not Applicable (or Unknown)
License:	Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden
Downloaded from:	https://hdl.handle.net/1887/32963

Note: To cite this publication please use the final published version (if applicable).

Cover Page

Universiteit Leiden

The handle http://hdl.handle.net/1887/32963 holds various files of this Leiden University dissertation

Author: Zhai, Jiali Teddy
Title: Adaptive streaming applications : analysis and implementation models
Issue Date: 2015-05-13

Chapter 3

Automated Analysis Model Construction: Deriving CSDF from Equivalent PPN

Mohamed A. Bamakhrama, **Jiali Teddy Zhai**, Hristo Nikolov, Todor Stefanov, "A Methodology for Automated Design of Hard-Real-Time Embedded Streaming Systems", *In Proceedings of the Conference on Design, Automation and Test in Europe (DATE'12)*, pp. 941–946, Dresden, Germany, March 12-16, 2012.

IN this chapter, we present an approach to convert a PPN to its input-output equivalent CSDF graph. As discussed previously, a wide range of powerful analysis techniques exist for the CSDF MoC, whereas it is easier to generate code for the PPN MoC. Considering the Daedalus^{RT} design flow shown in Figure 1.7 on page 13, deriving PPNs from SANLPs can be done in the PNgen compiler [125]. The code generation for the PPN MoC has been addressed in the ESPAM [96] tool.

Notation	Meaning
α	an input/output argument for a PPN function
$\mathcal{S}$	a sequence
Φ	a set of input/output ports associated with a process variant
v	a process variant

Table 3.1: Additional notations used in Chapter 3 besides the ones introduced in Chapter 2.

Hard real-time scheduling of acyclic CSDF graphs has been proposed in [22]. To have a fully automated design flow for designing hard real-time streaming systems, automated derivation of the CSDF MoC is the only missing step. From a high-level point of view, the contribution of this chapter enables to derive the equivalent CSDF graph from any SANLP.

It has been shown in [37] that a PPN without parameters is equivalent to a CSDF graph where the production/consumption sequences consist only of 0s and 1s. A '0' indicates that a token is not produced/consumed, whereas a '1' indicates that a token is produced/consumed. This chapter focues on the algorithm to derive the input-output equivalent CSDF graph from a PPN. Then, we demonstrate the applicability of the algorithm on a set of benchmarks in terms of time complexity. Finally, in the context of the Daedalus^{RT} design flow, we show that it is fast to derive CSDF graphs for three real-life streaming applications. Consequently, derivation of CSDF graphs enables to design multiple streaming applications on a single MPSoC platform in a short amount of time.

In addition to the notations introduced in Chapter 2, extra notations used in this chapter are summarized in Table 3.1.

3.1 The Algorithm

The procedure to derive a CSDF graph from its equivalent PPN is depicted in Algorithm 1. It consists of two main steps, namely 1) topology derivation and 2) consumption/production sequence derivation for input/output ports of each CSDF actor. Deriving the topology of the CSDF graph is straightforward. That is, the nodes, input/output ports, and edges in the CSDF graph have one-to-one correspondence to those in the PPN. Recall the SANLP described in Listing 1 on page 27 and its equivalent PPN shown in Figure 2.3. The derived CSDF graph is shown in Figure 3.1. It can be seen that the topology of the derived CSDF graph is the same as that of the PPN shown in Figure 2.3. Below, we focus on the second step which derives the consumption/production sequences for an input/output port of a CSDF actor.

The second step consists of three sub-steps. In the first sub-step (see line 3 in Algorithm 1), for each CSDF actor, we find the access pattern of the corresponding PPN process to its input/output ports. A more general MoC, Stream-based Functions [68], captures the regular access pattern of a function to its input/output ports using *function variant*. In this thesis, we introduce the notion of *process variant*, which captures the consumption/production behavior of the process.

Definition 3.1.1 (Process Variant)**.** A process variant v of a PPN process is defined by a tuple (D_v, Φ), where D_v is the variant domain with $D_v \subseteq D_P$, and Φ is a set of

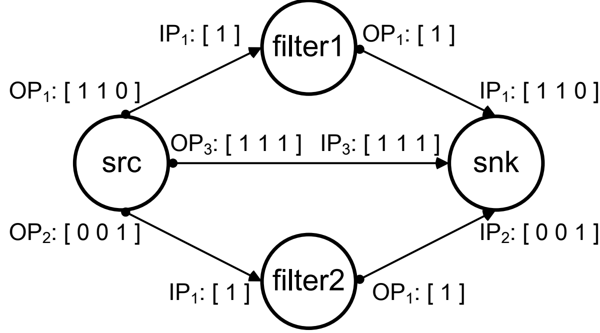


Figure 3.1: CSDF graph equivalent to the PPN shown in Figure 2.3.

Algorithm 1: Procedure to derive the CSDF MoC

Input: A PPN
Result: The equivalent CSDF graph
1 Derive the topology of the CSDF graph;
2 **foreach** *CSDF actor in the CSDF graph* **do**
3 Derive process variants (see Definition 3.1.1) for its corresponding PPN process ;
4 Derive a repetitive pattern of process variants ;
5 **foreach** *input/output port of the CSDF actor* **do**
6 **foreach** *process variant in the derived pattern* **do**
7 Generate consumption/production sequence ;

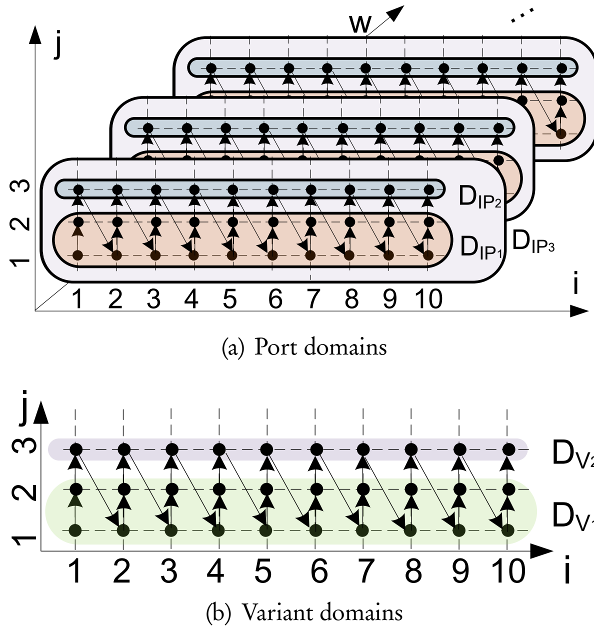
input/output ports.

For example, consider process snk shown in Figure 2.3 on page 29. One of the process variants is $(D_v, \{IP_1, IP_3\})$, where

$$D_v = \{(w, i, j) \in \mathbb{Z}^3 \mid w \geq 0 \wedge 1 \leq i \leq 10 \wedge 1 \leq j \leq 2\}.$$

According to Definition 3.1.1, for all iterations in domain D_v during the execution of process snk, this process always reads data from input ports IP_1 and IP_3.

The infinite repetitive execution of a PPN process is initially represented by a polyhedron. (e.g., see D_{snk} in Equation (2.8)). Therefore, we project out dimension w which denotes the `while`-loop from all the domains because it is irrelevant for the subsequent steps. As a result, the execution of a PPN process is represented by a polytope. Algorithm 2 is devised to derive the process variants for each PPN process.



(a) Port domains

(b) Variant domains

Figure 3.2: Domains of process snk in Figure 2.3.

Standard integer set operations are applied to the process domains. The basic idea is that, each port domain bound to a process function argument is intersected with all other port domains. The intersected domain and the difference between two port domains are then added to the set of process variants. In this way, all process variants are iteratively derived.

Consider process snk in Figure 2.3 on page 29. Its process function `snk(in1, in2)` has two arguments represented as a set $\mathcal{R} = \{\texttt{in1}, \texttt{in2}\}$, which is the input to Algorithm 2. The port domains bound to `in1` are D_{IP_1} and D_{IP_2}, while the port domain bound to `in2` is D_{IP_3}. These port domains are illustrated in Figure 3.2(a), surrounded by bold lines. Following the procedure described in Algorithm 2, we start with projecting out dimension w in the port domains, which yields:

$$\begin{aligned}
\texttt{in1}: D'_{IP_1} &= \{(i,j) \in \mathbb{Z}^2 \mid 1 \le i \le 10 \wedge 1 \le j \le 2\}, \\
D'_{IP_2} &= \{(i,j) \in \mathbb{Z}^2 \mid 1 \le i \le 10 \wedge j = 3\}, \\
\texttt{in2}: D'_{IP_3} &= \{(i,j) \in \mathbb{Z}^2 \mid 1 \le i \le 10 \wedge 1 \le j \le 3\}.
\end{aligned}$$

Algorithm 2 produces the set of process variants $\mathcal{V} = \{v_1, v_2\}$, where

$$\begin{aligned} v_1 &= (D_{v_1}, \{IP_1, IP_3\}), \\ D_{v_1} &= D'_{IP_1} \cap D'_{IP_3} = \{(i,j) \in \mathbb{Z}^2 \mid 1 \leq i \leq 10 \wedge 1 \leq j \leq 2\}, \\ v_2 &= (D_{v_2}, \{IP_2, IP_3\}), \\ D_{v_2} &= D'_{IP_2} \cap D'_{IP_3} = \{(i,j) \in \mathbb{Z}^2 \mid 1 \leq i \leq 10 \wedge j = 3\}. \end{aligned}$$

Process variant domains D_{v_1} and D_{v_2} are also illustrated in Figure 3.2(b). Process snk reads data from input ports IP_1 and IP_3 in variant domain D_{v_1}, whereas it reads data from input ports IP_2 and IP_3 in variant domain D_{v_2}.

In the second sub-step, (see line 4 in Algorithm 1), we find a one-dimensional, repetitive pattern of the process variants derived in the first sub-step. To find the repetitive pattern, we first project out dimension w in the process domain D_P to obtain domain D'_P. For a PPN process P, we build a sequence $\mathcal{S}_{D'_P}$ of the iterations $I \in D'_P$ according to their lexicographic order (see Definition 2.1.6 on page 25) as follows:

$$\mathcal{S}_{D'_P} = [I_1, \ldots, I_i, I_j, \ldots, I_{|D'_P|}],$$

where

$$I_i \prec I_j, \;\; \forall 1 < i < j < |D'_P|.$$

Next, we replace each iteration in sequence $\mathcal{S}_{D'_P}$ with the process variant to which the iteration belongs. For a PPN process P, the sequence of process variants $\mathcal{S}_P$ is given by

$$\mathcal{S}_P = [v_1, \ldots, v_i, \ldots, v_{|D'_P|}] \text{ and } I_i \in v_i, \;\; \forall 1 < i < |D'_P|.$$

For example, for process snk in Figure 2.3 and D_{snk} given in Equation (2.8) on page 28, the process domain after projecting out the w dimension is given as

$$D'_{snk} = \{(i,j) \in \mathbb{Z}^2 \mid 1 \leq i \leq 10 \wedge 1 \leq j \leq 3\}. \tag{3.1}$$

The sequence of the iterations in process domain D'_{snk} is

$$\mathcal{S}_{D'_{snk}} = [(1,1),(1,2),(1,3),(2,1),\ldots,(10,3)].$$

The lexicographic order of iterations in the sequence is represented using the arrows in Figure 3.2(b). The corresponding sequence of the process variants of process snk is

$$\begin{aligned} \mathcal{S}_{snk} = [&v_1, v_1, v_2, v_1, v_1, v_2, v_1, v_1, v_2, v_1, v_1, v_2, v_1, v_1, v_2, \\ &v_1, v_1, v_2, v_1, v_1, v_2, v_1, v_1, v_2, v_1, v_1, v_2, v_1, v_1, v_2]. \end{aligned} \tag{3.2}$$

Algorithm 2: Procedure to derive process variants of a process

Input: $\mathcal{R}$: the set of process function arguments

Result: $\mathcal{V}$: a set of process variants

1. $\mathcal{V} \leftarrow \emptyset$;
2. **foreach** $\alpha \in \mathcal{R}$ **do**
3.   **foreach** D_{port} *bound to* α **do**
4. $y \leftarrow (D_{port}, \{port\})$;
5.   **if** $\mathcal{V} = \emptyset$ **then**
6. $\mathcal{V} \leftarrow \{y\}$;
7.   **else**
8. $\mathcal{X} \leftarrow \mathcal{V}$;
9.   **foreach** $V \in \mathcal{V}$ **do**
10. $D_{intersect} \leftarrow v.D_{port} \cap y.D_{port}$;
11.   **if** $D_{intersect} \neq \emptyset$ **then**
12. $x_{intersect} \leftarrow (D_{intersect}, \{v.ports\} \cup \{y.ports\})$;
13. $x_{diff1} \leftarrow (v.D_{port} - y.D_{port}, \{v.ports\})$;
14. $x_{diff2} \leftarrow (y.D_{port} - v.D_{port}, \{y.ports\})$;
15. $\mathcal{X} \leftarrow \mathcal{X} \cup \{x_{intersect}\}$;
16.   **if** $x_{diff1}.D_{port} \neq \emptyset$ **then**
17. $\mathcal{X} \leftarrow \mathcal{X} \cup \{x_{diff1}\}$;
18.   **if** $x_{diff2}.D_{port} \neq \emptyset$ **then**
19. $\mathcal{X} \leftarrow \mathcal{X} \cup \{x_{diff2}\}$;
20.   **else**
21. $\mathcal{X} \leftarrow \mathcal{X} \cup \{y\}$;
22. $\mathcal{V} \leftarrow \mathcal{X}$;
23. **foreach** $v \in \mathcal{V}$ **do**
24.   **if** $|IP \in v.ports| \neq |\alpha_{in} \in \mathcal{R}|$ **then**
25. $\mathcal{V} \leftarrow \mathcal{V} \setminus v$;

Essentially, the length of the derived sequence is equal to the cardinality of process domain D'_P of a PPN process P, i.e., $|D'_P|$. Since $|D'_P|$ can be very large, the derived sequence might be very long. Thus, we express the sequence using the shortest repetitive pattern that covers the whole sequence. This shortest repetitive pattern can be found efficiently using a data structure called *suffix tree* [119]. In a

suffix tree, the *root* node is defined as the node with only outgoing edges and the *leaf* nodes are defined as the nodes with only incoming edges. The remaining nodes are called *internal*. A suffix tree has the following properties:

- A suffix tree for a sequence $\mathcal{S}$ of characters can be built in $O(|\mathcal{S}|)$ time [119].
- Each edge in the suffix tree is labeled with a non-empty subsequence starting from character $\mathcal{S}[i]$ to character $\mathcal{S}[j]$, where $1 \leq i \leq j \leq |\mathcal{S}|$.
- No two edges out of a node in the tree can have labels beginning with the same character. That is, the starting character of the label is different for all outgoing edges of a node in the suffix tree.
- A subsequence obtained by concatenating all subsequences found on the path from the root node to any internal node i occurs k times in the whole sequence, where k is the number of leaf nodes that node i has.
- The suffix tree is padded with a terminal symbol $.

Once a suffix tree is constructed for the sequence of process variants $\mathcal{S}_P$ according to the algorithm presented in [119], our problem can be formulated as: search the tree for the shortest repetitive pattern that covers the whole sequence $\mathcal{S}_P$, i.e., a subsequence of $\mathcal{S}_P$. Our problem can be solved based on finding the longest repeated substring in a given string, which can be solved in linear time. In our problem, we first pre-process the constructed suffix-tree to count the number of leaf nodes for each internal node. Among all outgoing edges of the root node, only the branch that has the same starting process variant as $\mathcal{S}_P$ is selected to explore. Then, a Breadth First Search (BFS) procedure is used, because shorter subsequences found at the levels closer to the root node are more likely to be the solution of our problem. For every path starting from the root to any internal node, the BFS procedure concatenates the labels on the edges. This concatenation results in a subsequence $\mathcal{S}_{sub}$ which occurs k times in the original sequence $\mathcal{S}_P$. Finally, we select the subsequence $\mathcal{S}_{sub}$ with the largest occurrence k that satisfies

$$|\mathcal{S}_{sub}| \times k = |\mathcal{S}_P|. \tag{3.3}$$

Similar to the longest repeated substring problem, our problem also has the linear time complexity.

Recall that the sequence of process variants $\mathcal{S}_{snk}$ for process snk is given in Equation (3.2). The corresponding suffix tree is constructed and illustrated in Figure 3.3. The shadow node denotes the root node and the solid nodes denote the leaf nodes. The others are internal nodes. It can be seen that every edge is labeled

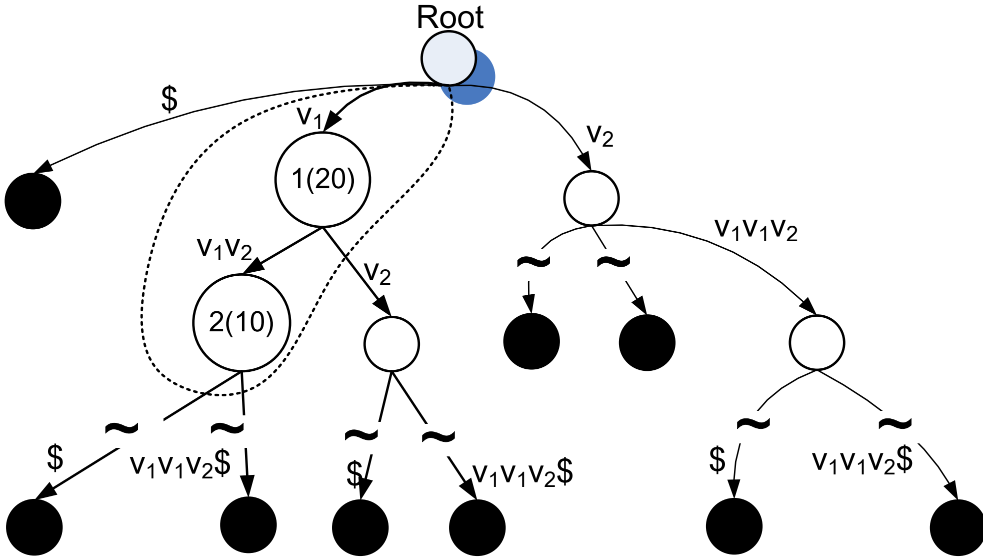


Figure 3.3: Suffix tree for the sequence of process variants $\mathcal{S}_{snk}$. The tildes represent the omitted part of the tree.

with a subsequence of process variants that occurs in the whole sequence $\mathcal{S}_{snk}$. In the pre-processing, computing for instance the number of leaf nodes for node 1 results in 20 (shown in the bracket in node 1). It means, the subsequence v_1 occurs in $\mathcal{S}_{snk}$ 20 times. In the beginning of BFS, only the edge connecting the root node to node 1 is selected to explore, because process variant v_1 labeled on the edge is the same as the first process variant in sequence $\mathcal{S}_{snk}$. In the next step, node 2 is selected and v_1 is concatenated with $v_1 v_2$ labeled on the edge connecting node 1 and node 2. It yields

$$\mathcal{S}_{sub} = [v_1, v_1, v_2].$$
$$|\mathcal{S}_{sub}| \times 10 = 30,$$
$$|\mathcal{S}_{snk}| = 30.$$

$\mathcal{S}_{sub}$ is shown in Figure 3.3 surrounded by a dashed line. At this step, the procedure terminates because the criteria, namely Equation (3.3), is satisfied.

In the last sub-step (see lines 5-7 in Algorithm 1), a consumption/production sequence is generated for each port of a CSDF actor. This is done by building a table in which each row corresponds to an input/output port, and each column corresponds to a process variant in the repetitive pattern derived in the second sub-step. If the input/output port is in the set of ports of the process variant, then its entry in the table is 1. Otherwise, its entry is 0. Each row in the resulting table represents a consumption/production sequence for the corresponding input/output port.

		Repetitive pattern		
		v_1	v_1	v_2
Input/output ports	IP_1	1	1	0
	IP_2	0	0	1
	IP_3	1	1	1

Table 3.2: Consumption/production sequences for actor snk in Figure 3.1.

Considering process snk, the consumption/production sequences of CSDF actor snk are generated as shown in Table 3.2. It can be seen that the consumption/production rates sequences for the ports are the same as the ones shown in Figure 3.1.

3.2 Experimental Results

We present in this section the experimental results of automated deriving CSDF graphs for some real-life applications specified as SANLPs (see Definition 2.1.8 on page 26). The main focus here is to demonstrate the applicability of our approach in terms of time-complexity. Then, we further demonstrate the application of automated CSDF derivation in the context of our Daedalus^{RT} framework for designing hard real-time streaming systems.

We took two applications, Filterbank and FMRadio, with original C code available from the StreamIT benchmark suit [54] and several reasonably complex[1] benchmarks from Polybench [101]. The characteristics of all benchmarks are shown in columns 2-4 in Table 3.3. For example, the Filterbank benchmark contains 367 lines of code in the SANLP. This excludes the code for each function in the SANLP. We believe that it is reasonably complex to express high-level behavior for most of real-life applications. Different benchmarks also vary in complexity of the access pattern to data arrays. For example, the ADI benchmark has very complex access pattern. Complex access pattern potentially increases the length of derived production/consumption sequences for input/output ports of CSDF actors. Our algorithm presented in this chapter was coded in C++ and integrated in PNtools as part of Daedalus^{RT} shown in Figure 1.7 on page 13. All experiments were conducted on an Intel Core 2 Duo T9600 CPU running at 2.80 GHz with 4GB memory in Linux Kubuntu 10.4.

The last column in Table 3.3 shows the running time needed to derive the corresponding CSDF graph for each benchmark. We can see that our algorithm is able to derive CSDF graphs in short amount of time. Note that the running time

[1]Other benchmarks either have simpler and less data dependencies or less number of tasks than the ones we selected.

Table 3.3: Characteristics of benchmarks and running times to derive their corresponding CSDF graphs.

Benchmarks	No. of actors	No. of channels	Lines of code (in SANLP)	Running time (sec.)
Filterbank	69	89	367	1.60
FMRadio	28	39	195	0.66
ADI[1]	28	167	209	7.26
FDTD[2]	17	71	144	0.89
Gauss[3]	11	26	75	7.82
Gram-Schmidt	9	20	48	1.85
Regularity detector	8	11	54	2.86

[1] ADI: Alternating direction implicit solver
[2] FDTD: 2D finite difference time domain kernel
[3] Gauss: 2D gauss blur filter for image processing

Table 3.4: Execution times of the phases in the Daedalus^{RT} flow for three streaming applications on a single MPSoC platform.

Phase	Time	Automation (Yes/No)
Parallelization	0.48 sec.	Yes
WCET analysis	1 day	No
Deriving the CSDF graphs	**5 sec.**	**Yes**
Deriving the platform/mapping	0.03 sec.	Yes
System synthesis	2.16 sec.	Yes
Total	$\sim$ 1 day	-
Total (excl. WCET analysis)	$\sim$ 7.67 sec.	-

here includes the time starting from SANLPs to CSDF graphs. In addition, the time to derive the implementation model, i.e., PPNs, using the PNgen compiler is also included. In this way, we can see clearly the benefits of starting from a SANLP and resulting in its equivalent CSDF graph. As mentioned previously, our approach can be readily integrated into, e.g., the $\sum$C toolchain [21], to greatly speedup application development process.

In the second experiment, we took three streaming applications specified in SANLPs, an edge-detection filter (Sobel) from the image processing domain, the Motion JPEG (M-JPEG) video encoder from the video processing domain, and the M-JPEG video decoder. Using the Daedalus^{RT} framework, we generated a functional implementation that can be synthesized in Xilinx Platform Studio 13.2 targeting

Xilinx Virtex-6 FPGA ML605 evaluation kit [15]. Table 3.4 shows the running time of each design phase in the Daedalus^{RT} design flow. We observe that, if the CSDF graphs of the three applications were derived manually by hand, it would take several days. Instead, using the solution presented in this chapter, deriving these three CSDF graphs takes 5 seconds. Thus, the automated CSDF derivation is one of the key enablers for a fully automated and fast design flow.