Identifying Communication Models in Process Networks derived from Weakly Dynamic Programs

Dmitry Nadezhkin, Todor Stefanov
Leiden Embedded Research Center, Leiden University, The Netherlands
{dmitryn,stefanov}@liacs.nl

***Abstract*—Process Networks (PNs) is an appealing computation abstraction helping to specify an application in parallel form and realize it on parallel platforms. The key questions to be answered are how a PN can be derived and how its components can be realized efficiently on a given parallel system. In this paper we present a novel approach of communication model identification in PNs derived from nested loops programs with data-dependent control statements, which we call Weakly Dynamic Programs (WDP). Identifying communication models at compile-time allows us to select the most efficient realization of communication components in a PN. We show how our approach is seamlessly integrated into our existing procedure of *automated* PN derivation from WDP programs. This paper can be considered as an important complementary work that makes our automated derivation of WDPs to PNs complete.**

I. Introduction

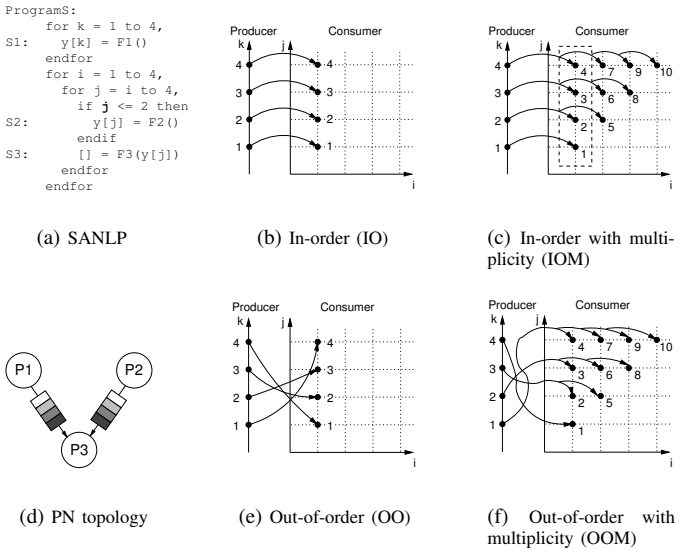
The majority of modern applications used in the area of multimedia, imaging and signal processing have a high computational demand. To satisfy this demand, on the one hand, new parallel hardware platforms are emerging [1]. On the other hand, to utilize the parallel resources provided by the platform, application designers have to draw thorough attention to the concurrency properties of the applications. The concurrency, however, introduces an ambiguity into the execution flow of a program, such as data races, data incoherency, deadlocking and data starvation. A widely accepted approach in dealing with concurrency is to use parallel models of computations (MoC) [2], which bring all such variability under control. Additionally, a model specifying the application in a parallel form can be mapped onto the parallel platform in a systematic and possibly automated way.

One of the models which is widely used for specifying streaming applications is the Kahn Process Network (KPN) model [3]. The KPN is a deterministic model with distributed memory and control. Semantically, a KPN is a unidirectional graph in which the nodes are processes executing a sequential code and the edges are FIFO channels which communicate data between processes. A process may be in two states: firing or communicating. The former case occurs when a data to be processed is available locally. In the latter case, the process sends/receives data tokens to/from other processes. Every FIFO channel implemented as a point-to-point communication has one Producer and one Consumer processes, thereby forming a Producer/Consumer pair (P/C pair). An example of two P/C pairs P1/P3 and P2/P3 is shown in Figure 1(d).

In any point-to-point communication, the firings of the Producer process generate data tokens in a certain order. We call it the production order. The tokens are sent to the Consumer process over the FIFO channel. In order to fire, the Consumer process needs the data tokens in a certain order. We call it the consumption order. When the production and consumption orders are the same, we say that the P/C pair communication is *in-order*. Otherwise, the P/C pair communication is *out-of order*. Consider, for example, Figure 1(b). It depicts the Producer and Consumer processes, where the points on the coordinate systems designate the firings of the processes and the arrows reflect the data dependencies between firings. The numbers at the points show the production/consumption orders. Figure 1(b) shows that the production order of the Producer process coincides with the consumption order of the Consumer process. This is an example of *in-order* communication in a P/C pair. Similarly, Figure 1(e) illustrates that the consumption order of the Consumer is reversed to the production order of the Producer process. This is an example of *out-of-order* communication.

In a P/C pair, it may occur that the Consumer process may need to reuse in future firings a token that has just been received from the Producer process. In such case we say that the P/C pair communication has a *multiplicity*. For example, consider the firing of the Producer and Consumer depicted in Figure 1(c). The production and consumption orders are the same, thus, the P/C pair communication is *in-order*. Additionally, we may notice, for example, that the data token needed for firing 3 of the Consumer process, will be needed on firings 6 and 8. Thus, the P/C pair communication has a multiplicity. Likewise, a P/C pair communication may be *out-of-order* and has a *multiplicity*. An example of such P/C pair communication is depicted in Figure 1(f). The four different types of P/C pair communication described above, determine four communication models between processes. They are: *in-order (IO)*, *out-of-order (OO)*, *in-order with multiplicity (IOM)* and *out-of-order with multiplicity (OOM)*.

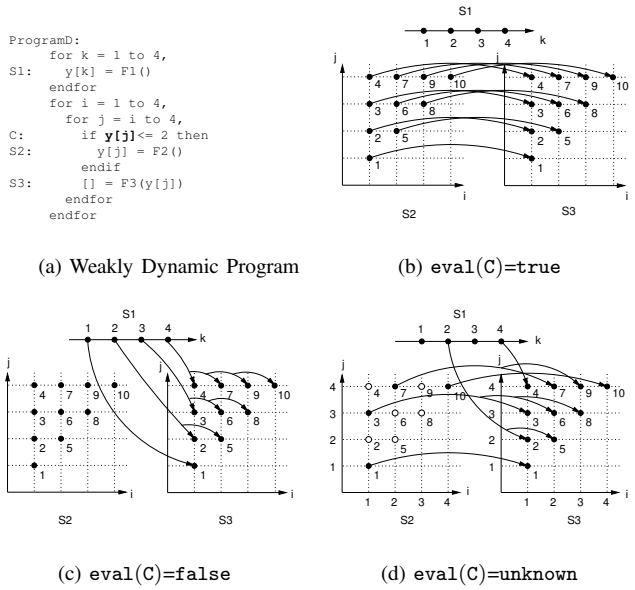
In order to implement a KPN, all communication models have to be realized over a FIFO channel. The *in-order* models can be implemented with a FIFO in a straightforward way, as the order of writing into the FIFO channel and the order of reading from it are the same. The *out-of-order* models would require a FIFO channel augmented with a controller implementing the reordering. In a similar manner, the models with *multiplicity* would require a FIFO channel with additional memory to store



(a) SANLP (b) In-order (IO) (c) In-order with multiplicity (IOM)

(d) PN topology (e) Out-of-order (OO) (f) Out-of-order with multiplicity (OOM)

Fig. 1: SANLP program, P/C pair and types of communication models.



(a) Weakly Dynamic Program (b) `eval(C)=true`

(c) `eval(C)=false` (d) `eval(C)=unknown`

Fig. 2: WDP program and Dependencies examples.

the tokens which will be reused later.

The difference in realization puts the communication models into a hierarchy. The realization of the *OOM* model is the most general, as it is built of all components present in the realizations of the other models. In other words, all P/C pair communication models can be implemented with the realization of the *OOM* model. However, the more general a realization is, the more resources it needs and more run-time overhead is introduced. More importantly, the *IO*,*OO* and *IOM* communication models might be implemented with simpler realization. Therefore, it is important to identify at compile-time the communication model of a pair of processes in order to instantiate the most efficient realization. This paper addresses this important identification problem. The main novel contribution of this paper is a formal procedure for identifying communication models in process networks derived from *Weakly Dynamic Programs* (WDP).

A. Related work

To the best of our knowledge, not much attention has been devoted to automatic communication model identification. An automatic procedure exists for communication model identification while translating *static* affine nested loop programs (SANLP) into functionally equivalent KPNs [4]. In SANLPs the memory array subscripts, loop bounds and conditional control structures are affine constructs of surrounding loop iterators, program parameters and constants. An example of a static program is given in Figure 1(a). In the communication model identification procedure two integer linear problems (ILPs) have been formulated: *Reordering Problem* (RP) and *Multiplicity Problem* (MP). We have found that the RP is not precise and lead to more complex realization of communication models as will be shown in Section III-D. Thus, one of the novel contribution of this paper is the formulation of a new RP.

Besides this, the major novel contribution of our paper is the communication model identification procedure while translating to a PN a more general class of applications, which are defined in [5] and called *Weakly Dynamic Programs*. In this class of affine nested loops programs, the static constraint is relaxed, i.e., the conditions in control structures might be dependent on some information that is not known at compile-time and may change at run-time. For example, the program shown in Figure 2(a) is an affine nested loop program with a control statement at line C that depends on the evaluation of some data unknown at compile-time.

An approach for automatic translation of WDP programs into equivalent KPNs has been studied in [5]. However, a compile-time procedure for communication model identification has not been investigated and our paper can be considered as an important complementary work that makes the translation of WDP to KPNs complete. Additionally, we give a new formulation of the *Reordering Problem* in the context of WDP programs.

In the following section we give a motivating example demonstrating why it is more difficult to identify the communication models in dynamic programs compared to static programs. In Section III, we present a formal framework used for communication model identification in static programs. In Section IV we present our novel solution approach to identify the communication models in Weakly Dynamic Programs. Section V concludes the paper.

II. Motivating example

The overall challenge of communication model identification in dynamic programs is how to deal at compile-time with uncertainties introduced by dynamic control structures. In this section we will demonstrate that: 1) an exact communication

model identification in PNs derived from WDP programs is not possible at compile-time; 2) an approach used for communication model identification while translating static programs into equivalent PNs cannot directly be applied to WDP programs, and 3) nevertheless, at compile-time, for any P/C pair we can identify the most general communication model which can be used to realize all possible data dependency patterns that may occur in different instances of the dynamic program.

Consider the WDP program shown in Figure 2(a). It is an affine nested loops program with a condition at line C evaluating some run-time data. Depending on the evaluation, either statement S1 or S2 produces the data for every firing of statement S3, and therefore, the PN graph corresponding to this WDP consists of two P/C pairs: S1/S3 and S2/S3.

Assume, first, that the condition at line C always evaluates to *true*, and, thus, all the data needed by statement S3 is produced by statement S2 only. The data dependency relations depicted in Figure 2(b), determine the communication model of this P/C pair to be *IO*. The opposite case is when the condition at line C always evaluates to *false*. Depicted in Figure 2(c), this time, relations exist between statement S1 and S3 only. As some data tokens will need to be reused in future firings, the communication model is *IOM*. In general, the result of condition evaluation at line C is arbitrary and unknown at compile-time. In this case, on some firings, the data needed by statement S3 is produced by statement S1, on other firings by statement S2. An example of possible relations is depicted in Figure 2(d), where the communication models of S1/S3 and S2/S3 P/C pairs are *IOM*.

The three abovementioned examples of possible data dependency patterns correspond to different instances of a dynamic program, where an instance of a program is its evaluation with particular input dataset. Therefore, the dependency patterns in WDP programs are data dependent and are unknown at compile-time. This makes an exact compile-time communication model identification in PNs derived from WDP programs impossible.

In static programs, different instances of a program correspond to one and the same single dependency pattern which is known at compile-time. Therefore, the communication model identification approach used for static programs cannot be applied to dynamic programs, because in WDPs the data dependency relation in a P/C pair is not unique and unknown at compile-time.

Although an exact identification of communication models in dynamic programs is not possible, still we can analytically identify at compile-time the communication models of a P/C pair in all possible instances of a dynamic program. Based on this information, we can realize the communication of a P/C pair with the most general communication model which implements all possible data dependency relations. For example, we may observe in Figure 2 that the production/consumption orders in S1/S3 and S2/S3 pairs of the PN considered above are the same. Thus, the communication in all P/C pairs is *in-order*. Moreover, in some program instances a multiplicity in the communication is possible and, according to the realization hierarchy of communication models, the most general model for S1/S3 and S2/S3 pairs is *IOM*. Therefore, we can implement the communication in S1/S3 and S2/S3 pairs as *IOM* model.

III. Background and notations

In this section we formally describe the steps needed to translate a sequential *static* program into an equivalent PN specification, including the communication model identification procedure. This is necessary in order to understand the major contribution of this paper presented in Section IV-B, i.e., our novel communication model identification procedure for *weakly* dynamic programs. The steps will be illustrated with the example program given in Figure 1(a) called `ProgramS`. This program is a static affine nested loops program, as memory array indices and conditions are affine constructs of surrounding loops iterators. In this program statements S1 and S2 write to a one-dimensional array `y` and statement S3 reads from it.

The procedure of PN derivation from a static program consists of two steps. In the first step, a dependence analysis is applied to the source code of a program. The goal of this analysis is to determine if evaluation of a statement depends on evaluation of other statements and to find these evaluations. For example, in `ProgramS`, the goal of the dependence analysis is to find whether statement S3 depends on statements S1 or S2 via array y and at which iterations. Or in other words, for every element of array y read at a given iteration of statement S3, the dependence analysis finds which statement, S1 or S2, and at what iteration, writes data to the given array element. The result of the analysis forms the dependency relations between iterations of statements writing/reading to/from the array. For example, from the program code illustrated in Figure 1(a), we can easily see, that S3 depends on S2 when $1 \leq j \leq 2$, and S3 depends on S1 for iterations: $2 < j \leq 4$. For dependence analysis we use a technique called Exact Array Dataflow Analysis (EADA) [6] which will be described in Section III-B.

In the second step of the PN derivation procedure, the topology of the PN is built and the communication models of all P/C pairs are identified and realized using FIFO channels. The second step will be explained in Section III-C.

A. Notations

An iteration vector x of a statement is built of iterators of surrounding loops. The set of values of an iteration vector for which a statement is executed represents an iteration domain, denoted by $\mathbf{D}()$. For example, the iteration domain of statement $S2$ in `ProgramS` is: $\mathbf{D}(S2) = \{1 \leq i \leq 4 \wedge i \leq j \leq 4 \wedge j \leq 2\}$. An evaluation of a statement W on iteration x is called an operation and denoted as $\langle W, x\rangle$. By "$\prec$" we denote ordering of operations. An operation $\langle W, x\rangle$ is evaluated before an operation $\langle R, y\rangle$ according to the program sequence if: 1) x lexicographically precedes y; or 2) if $x = y$ and statement W precedes statement R in the program text. As described in [6], order "$\prec$" can be expanded to a system of linear inequalities. With "$\max$" we denote the lexicographical maximum operator.

B. Exact Array Dataflow Analysis

Consider two statements W and R, and operations $\langle W, x\rangle$ and $\langle R, y\rangle$, where the first writes to an array and the second reads from it. To find whether the operation $\langle W, x\rangle$ is a source for operation $\langle R, y\rangle$ we need to build a system of linear inequalities:

$$\begin{array}{lll} \mathbf{Q}_{WR}(y) = \{x \mid & x \in \mathbf{D}(W), & \text{(c1)} \\ & \mathcal{I}_W(x) = \mathcal{I}_R(y), & \text{(c2)} \\ & \langle W, x\rangle \prec \langle R, y\rangle. & \text{(c3)} \end{array} \quad (1)$$

The first constraint (c1) states that the source iteration x has to exist, i.e., it has to belong to the iteration domain of a W statement. The constraint (c2) specifies that if there is a dependency between two operations, both have to access the same array element. To access an array element, operation $\langle W, x\rangle$ uses an affine indexing function $\mathcal{I}_W()$ and operation $\langle R, y\rangle$ uses indexing function $\mathcal{I}_R()$. The (c3) constraint determines an order of operations, i.e., source $\langle W, x\rangle$ has to be evaluated *before* operation $\langle R, y\rangle$.

While there might be many operations satisfying system (1), i.e., writing to the same array element, only one of them writes data after all the others and before reading by $\langle R, y\rangle$ occurs. In other words, the source operation is the lexicographical maximum between all operations satisfying system $\mathbf{Q}_{WR}(y)$:

$$\mathbf{K}_{WR}(y) = \max \ \mathbf{Q}_{WR}(y). \quad (2)$$

In general, there might be several statements $W_1,\ldots,W_m$ writing to the same array. In this case, we have to consider all pairs $W_1/R,\ldots W_m/R$. The actual source is the "last" operation between all operations of all statements:

$$\sigma(\langle R, y\rangle) = \max \ \{\langle W_k, \mathbf{K}_{W_k R}(y)\rangle \mid \ k \in [1, m]\}. \quad (3)$$

For example, consider `ProgramS`. There are two statements, S1 and S2 writing to array y. Consider two pairs S1S3 and S2S3 and let us build the following systems of linear constraints $\mathbf{Q}_{\mathrm{S1S3}}((i_3, j_3))$ and $\mathbf{Q}_{\mathrm{S2S3}}((i_3, j_3))$ as depicted in Table I.

$\mathbf{Q}_{\mathrm{S1S3}}((i_3, j_3))$	$\mathbf{Q}_{\mathrm{S2S3}}((i_3, j_3))$	
$1 \le k \le 4$	$1 \le i_2 \le 4 \wedge i_2 \le j_2 \le 4$ $j_2 \le 2$	(c1)
$k = j_3$	$j_2 = j_3$	(c2)
`true`	$\langle S2, (i_2, j_2)\rangle \prec \langle S3, (i_3, j_3)\rangle$	(c3)

TABLE I: Examples of system (1) for S1S3 and S2S3 statements.

After solving the ILP problems formulated in Table I and finding "max" in Equation 3, the source operation $\sigma(\langle \mathrm{S3}, (i_3, j_3)\rangle)$ for the data read by statement S3 is:

$$\begin{array}{llll} \textbf{if} & j_3 \le 2 & \textbf{then} & \langle S2, (i_3, j_3)\rangle \\ & & \textbf{else} & \langle S1, (j_3)\rangle. \end{array} \quad (4)$$

The abovementioned dependency relations are illustrated in Figure 4(b).

C. Synthesizing a PN

In this section we consider the second step of the procedure for PN derivation from a static program, i.e., the topology of the PN is built and the communication models of all P/C pairs are identified.

The dependence analysis applied on the source code of a static program allows us to specify the program in a Single Assignment Code (SAC) form as shown in [6]. The SAC program is functionally equivalent to the original program, with a difference that every variable is written exactly once. This makes possible to realize data communication between statements with FIFO abstraction which is the communication mechanism used in PNs. An example of the SAC form of the program given in Figure 1(a) is illustrated in Figure 4(a). The main difference between these programs is that statements in the SAC program communicate point-to-point via unique dedicated arrays `y_1` and `y_2` that guarantee the original data dependencies.

From the specification of `ProgramS` in SAC form depicted in Figure 4(a), we can build the topology of the PN depicted in Figure 1(d). First, every functional statement becomes a process in the PN. For example, in Figure 4(a), lines 1–4 determine the computational and communicational structure of process P1; lines 5–9 determine process P2 and lines 5–6 and 10–15 determine process P3. Second, variables or arrays that are used to communicate data between statements become channels in the PN. For example, arrays `y_1` and `y_2` form two FIFO channels to communicate data to P3 from processes P1 and P2, respectively. Furthermore, channels connect pairs of processes which form P/C pairs. In the example, the P/C pairs are P1/P3 and P2/P3. Third, mapping functions are derived which establish relationships between firings of processes forming a P/C pair. Mapping function in a P/C pair gives for each iteration of statement corresponding to a Consumer process, the iteration of statement corresponding to a Producer process. For example, for P1/P3 pair, the mapping function is:

$$f_{P1P3} : \mathbb{Z}^2 \to \mathbb{Z} : k = (0\ 1) \begin{pmatrix} i_3 \\ j_3 \end{pmatrix},$$

and for P2/P3 pair, the mapping function is:

$$f_{P2P3} : \mathbb{Z}^2 \to \mathbb{Z}^2 : \begin{pmatrix} i_2 \\ j_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i_3 \\ j_3 \end{pmatrix}.$$

From Figure 4(b) we can identify the domains of mapping functions f_{P1P3} and f_{P2P3}. The f_{P1P3} function maps all iteration points of statement S3 with $j_3 > 2$ to the iteration domain of statement S1. Thus, the domain of mapping function $\mathbf{D}(f_{P1P3})$ is: $i_3 \in [1, j_3], j_3 \in [3, 4]$. Analogically, the domain $\mathbf{D}(f_{P2P3})$ of mapping function f_{P2P3} is: $i_3 \in [1, 2], j_3 \in [i_3, 2]$. The domains of mapping functions can be derived from Equation 3.

D. Communication Model Identification: Static Programs

Having built the topology of a PN, the last step of the PN derivation is to identify the communication models of its FIFO

channels.

In Section I, we explained that the communication model of a channel depends on the order of firings of the Producer and Consumer processes. Relations between firings of processes in a P/C pair are expressed by the mapping functions. In [7], the ordering in the communication models of a P/C pair is defined as follows:

Definition III.1

A P/C pair is **in-order** iff the mapping function f preserves the token order, i.e., every two Consumer iteration points $y^1 \prec y^2$ are mapped onto two Producer iteration points $x^1 = f(y^1)$ and $x^2 = f(y^2)$ such that $x^1 \preceq x^2$. If a P/C pair is not in order we call it **out-of-order**.

According to Definition III.1, the example of communication model given in Figure 1(c) will be classified as OOM. However, in Section I we considered this example as IOM, because first the tokens are read **in-order** (see dashed box in Figure 1(c)) and they are reused later locally. Thus, we conclude that Definition III.1 is not precise and lead to more complex realizations of communication models. Instead, we present a novel formulation of ordering in communication models as follows:

Definition III.2

A P/C pair is **in-order** iff the mapping function f preserves the token order, i.e., every two Consumer iteration points $y^1, y^2 \in \mathrm{LmP}(\mathbf{D}(f)) \wedge y^1 \prec y^2$ are mapped onto two Producer iteration points $x^1 = f(y^1)$ and $x^2 = f(y^2)$ such that $x^1 \preceq x^2$. If a P/C pair is not in order we call it **out-of-order**.

The LmP($\mathbf{D}(f)$) set used in Definition III.2 stands for the *Lexicographically minimal Preimage* (LmP) and is defined in Figure 3 below. It is an Integer Linear problem finding the Consumer iteration points that read the tokens from the Producer for the first time.

$$\mathbf{objective}: \quad y_m = min_{\mathrm{lex}}\{y(x)\}, \qquad \mathbf{subject\ to}: \begin{cases} y \in C(N), \\ x = f(y). \end{cases}$$

Fig. 3: The Linear minimal Preimage ILP problem.

For example, in Figure 1(c), the LmP is marked by the dashed box and according to Definition III.2 this P/C pair is **in-order**. Similarly, for our running example shown in Figure 4(b), the LmP corresponds to the dashed box and the communication model of a P/C pair formed by statement S2 and S3 is **in-order**.

The definition of multiplicity in a P/C pair we take from [7].

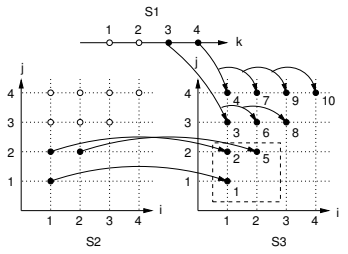
Definition III.3

A P/C pair is **without multiplicity** iff the mapping function f is injective, i.e., $\forall\ y^1, y^2 \in \mathbf{D}(\mathrm{C}))\ s.t.\ y^1 \neq y^2 \Rightarrow f(y^1) \neq f(y^2)$. Otherwise we say that the P/C pair is **with multiplicity**.

For example, in our running example shown in Figure 4(b), we see that there are at least two different iteration points of S3 which correspond to a single iteration point of S1. Therefore, the P/C pair formed by statements S1 and S3 has a **multiplicity**.

```
1:  for k = 1 to 4,
      if k >= 3,
S1:     y_1[k] = F1()
    end end
5:  for i = 1 to 4,
      for j = i to 4,
7:      if j <= 2,
S2:       y_2[i,j] = F2()
        end
10:     if j <= 2,
11:       in_0 = y_2[i,j]
        else
13:       in_0 = y_1[j]
        end
S3:     [] = F3(in_0)
16: end end
```

(a) SAC code



(b) Data dependencies

Fig. 4: An example of a SAC code and dependencies.

To analytically determine the communication type of an arbitrary P/C pair we specify the *Reordering Problem* (RP) and the *Multiplicity Problem* (MP) as given in Figure 5, which correspond to Definitions III.2 and III.3, respectively.

$$\begin{cases} y^1, y^2 \in \mathrm{LmP}(\mathbf{D}(f)), \\ y^1 \prec y^2, \\ f(y^1) \succ f(y^2). \end{cases} \qquad \begin{cases} y^1, y^2 \in \mathbf{D}(f), & (c1) \\ y^1 \neq y^2, & (c2) \\ f(y^1) = f(y^2). & (c3) \end{cases}$$

(a) Reordering Problem (RP) (b) Multiplicity Problem (MP)

Fig. 5: Reordering and Multiplicity Problems in static programs.

The RP and MP problems are integer linear problems, meaning that if, for example, there is an integer solution satisfying RP, then the communication model is *out-of-order*. Otherwise, the the communication model is *in-order*. Similarly, if there is an integer solution satisfying MP, then the communication model is *with-multiplicity*, and otherwise the communication model is *without-multiplicity*.

IV. Solution Approach for WDP programs

In this section we formally describe our novel compile-time procedure for communication model identification while translating a WDP program into equivalent PN.

In Section III-D, in Definitions III.1 and III.3 we reviewed the Reordering and Multiplicity problems which are used to identify communication models of channels while translating *static* programs into equivalent PNs. Due to the fact that Definition III.1 leads to more complex realizations of communication models, we redefined the RP in Definition III.2. The same Definitions III.2 and III.3 of communication models hold for Weakly Dynamic Programs (WDPs). However, for such programs, we cannot completely build problems like in Figure 5 and solve them at compile-time. This is because some

information is missing due to data-dependent **if**-conditions and, thus, mapping functions used in formulations of the Reordering and Multiplicity problems are not unique and cannot be exactly determined at compile-time. This has been shown by the example in Section II.

In the same example we explained that there might be many instances of a dynamic program, and, therefore, in every instance there is a unique set of mapping functions. To capture all unknown information at compile-time our novel approach is to define and use parameterized mapping functions with parameters that are determined at run-time.

We introduce single assignment form called *dynamic* Single Assignment Code (dSAC) which is an extension of the SAC presented in Section III-C. We use dSAC to derive parameterized mapping functions. The derivation of the dSAC form is based on a modified version of parameterized dependence analysis algorithm called Fuzzy Array Dataflow Analysis (FADA) developed by Feautrier et all and described in [8].

Additionally, we explain how to derive the topology of a PN and how to set up parameters at run-time. Finally, we present how to define integer linear problems used to identify the communication model of channels in a PN derived from dynamic programs. In the following section, we describe the FADA algorithm as it is an important part of our solution.

A. Fuzzy Array Dataflow Analysis

Consider two statement W and R of a *weakly dynamic* program. Operation $\langle W, x\rangle$ writes to and operation $\langle R, y\rangle$ reads from the same array. Moreover, let statement W be surrounded by a data-dependent **if**-condition. As an example, consider Figure 2(a): statements $S2$ and $S3$ as W and R, respectively, and **if**-condition C surrounding statement $S2$.

In Section III-B, we explained that in order two operations $\langle W, x\rangle$ and $\langle R, y\rangle$ of a *static* program to be dependent, they should comply to the system of linear inequalities (1). By analogy to the static case, to find whether operation $\langle W, x\rangle$ is a source for operation $\langle R, y\rangle$ in a *dynamic* program, we need to build a system of linear inequalities:

$$\mathbf{Q}_{WR}(y,\alpha) = \{x \mid \begin{array}{ll} x \in \mathbf{D}(W), x = \alpha, & (c1)\\ \mathcal{I}_S(x) = \mathcal{I}_R(y), & (c2)\\ \langle S, x\rangle \prec \langle R, y\rangle. & (c3)\end{array} \tag{5}$$

The meaning of constraints $(c2)$ and $(c3)$ are the same as in system (1): operations should access the same array element and the writing operation should occur before the reading operation. We will explain the meaning of constraint $(c1)$. As statement W is surrounded by data-dependent **if**-condition, exact operations of W cannot be determined at compile-time. Thus, for any reading operation $\langle R, y\rangle$ it is impossible to determine the *exact* source operation. The idea of the FADA algorithm is to introduce a parameter which would hide unknown information, i.e., a parameter is used to indicate at which iteration a writing operation $\langle W, x\rangle$ may occur.

The meaning of constraint $(c1)$ is that we do not know exactly at which iteration points $x \in \mathbf{D}(W)$ writing to the array occurs. But we assume that this happens for iterations $x = \alpha$, where α is a free parameter which values have to be determined at run-time. Because operations satisfying system (5) are not exact, we call them *approximated* sources. For example, consider possible dependency relations depicted in Figure 2(d) between statements of `ProgramD`. Let us look at iteration $(i, j) = (3, 3)$ of statement S3. Also, assume that result of condition evaluation at line C was true on iteration $(i, j) = (1, 3)$ and was false on iterations $(2, 3)$ and $(3, 3)$. The source for operation $\langle S3, (3, 3)\rangle$ is $\langle S2, (1, 3)\rangle$. Thus, for iteration $(3, 3)$ of statement S3, $\alpha = (1, 3)$.

Similar to EADA, there might be many operations satisfying system (5), but the source is the lexicographical maximum between all such operations:

$$\mathbf{K}_{WR}(y,\alpha) = \max \mathbf{Q}_{WR}(y,\alpha). \tag{6}$$

In general, there might be several statements $W_1, \ldots, W_m$ writing to the same array element. For each W_k, $k = [1..m]$, we find approximate source. In the EADA algorithm, the source operation computed in Equation 3 is the maximum between all operations of all statements. However, in FADA, we cannot compute the maximum as all approximated sources are parameterized. Therefore, to find the source, we combine all approximate sources as described in [8]:

$$\sigma(\langle R, y\rangle, \alpha) = \max\{\langle W_k, \mathbf{K}_{W_k R}(y)\rangle | \quad k \in [1, m]\}. \tag{7}$$

For example, consider `ProgramD` depicted in Figure 2(a). There are two statements $S1$ and $S2$ writing to array y and one statement $S3$ which reads from it. For every pair $S1S3$ and $S2S3$ we build the systems of linear inequalities (5) which is depicted in Table II. For pair $S1S3$ all operations of statement $S1$ are known and thus, a parameter is not introduced. However, for pair $S2S3$ we introduce parameters α_i and α_j which for read operation $\langle S2, (i_3, j_3)\rangle$ designate source iteration and are unknown at compile-time.

$\mathbf{Q}_{S1S3}((i_3, j_3))$	$\mathbf{Q}_{S2S3}((i_3, j_3), (\alpha_i, \alpha_j))$	
$1 \le k \le 4$	$1 \le i_2 \le 4 \wedge i_2 \le j_2 \le 4 \wedge$ $i_2 = \alpha_i \wedge j_2 = \alpha_j$	(c1)
$k = j_3$	$j_2 = j_3$	(c2)
`true`	$\langle S2, (i_2, j_2)\rangle \prec \langle S3, (i_3, j_3)\rangle$	(c3)

TABLE II: Examples of system (5) for S1S3 and S2S3 statements.

After combining approximate sources found in S1S3 and S2S3 pairs, the source operation for statement S3: $\sigma(\langle S3, (i_3, j_3)\rangle, (\alpha_i, \alpha_j))$ is:

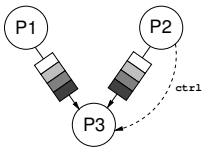
$$\begin{array}{llll} \textbf{if} & i_3 \ge \alpha_i \wedge j_3 == \alpha_j & \textbf{then} & \langle S2, (\alpha_i, \alpha_j)\rangle \\ & & \textbf{else} & \langle S1, (j_3)\rangle. \end{array} \tag{8}$$

From Equation 8 we see that for any reading operation $\langle S3, (i_3, j_3)\rangle$ the source of the data can be from two different locations. The source is in $S1$ when for given j_3 none of the previous evaluations of the condition at line C was `true`. Otherwise, the source is in $S2$. Parameters α_i and α_j are not

known at compile-time, therefore, the exact source is unknown at compile-time either.

```
1:  for j = 1 to 4,
      ctrl[j] = (5,5)
3:  end
4:  for k = 1 to 4,
S1:  y_1[k] = F1()
    end
7:  for i = 1 to 4,
      for j = i to 4,
C:     if y_1[j]<=2,
S2:      y_2[i,j] = F2()
11:      ctrl[j] = (i,j)
       end
```

```
13:   c1 = ctrl[j].i
14:   c2 = ctrl[j].j
15:   if i >= c1 &&
          j == c2,
       in_0 = y_2[c1, c2]
      else
       in_0 = y_1[j]
      end
S3:   [] = F3(in_0)
     end
    end
```



(a) dSAC code (b) PN from WDP

Fig. 6: An example of a dSAC and corresponding PN.

B. Communication Model Identification based on dSAC form

In this section we present the dynamic SAC (dSAC) form which we derive from FADA analysis, and our novel communication model identification procedure based on dSAC.

Because of the approximated data dependencies our dSAC notion is different from the classical single assignment code (SAC) form. In the SAC form, every variable is written only once, whereas dSAC has a property that every variable is written *at most once*. This property implies that some of the variables may not be written at all. This is because of the data-dependent control structures in a WDP, i.e., outcome of the conditions is unknown at compile-time.

The dSAC form of `ProgramD` derived from Equation 8 is depicted in Figure 6(a). It is in dSAC form because if we consider line1 0, we do not know at compile-time at which iteration the elements of array `y_2` will be written. The only thing known is that they will be written at most once. The other array `y_1` has the same property as an array in SAC form: every element is written exactly once.

Another new feature of the dSAC form is the presence of parameters that originate from FADA analysis. For example, in Figure 6(a) the parameters are `c1` and `c2` which correspond to parameters α_i and α_j in Equation 8. In order to make dSAC to be functionally equivalent to the initial WDP program, the values of these parameters have to be changed at run-time.

Parameters are changed with the help of control variables that store the correct value of the parameters for every iteration. For example, dynamic change of the values of `c1` and `c2` is accomplished by lines 13 and 14. The control variable `ctrl` at line 11 stores the iterations for which the data-dependent condition at line C is true. The control variables must be initialized with values that are greater than the maximum value of the corresponding parameters. Therefore, the control variable `ctrl` is initialized with value `(5,5)` at lines 1–3 in Figure 6(a). Writing to the control variables is performed just after the functional statement $F2()$, see line 11 in Figure 6(a). This guarantees that when the function is executed, the current iteration is stored in the control variables. The values of the control variables are propagated and assigned to the parameters `c1` and `c2` at lines 13 and 14. These parameters are used to evaluate the conditions at lines 15 and 16 which determine the source of the data for function $F3()$.

From the dSAC we can build the topology of the PN depicted in Figure 6(b). Every functional statement becomes a process, and every variable or array becomes a channel. For example, lines 4-6 form process P1; lines 1-3 and 7-12 form process P2; and, finally, lines 7-8, 13-21 form process P3. Processes P2 and P3 are connected with two channels: the first one which originate from array `y_2` for transferring data, and the second channel which originate from control variable `ctrl`, to communicate the outcome of the condition at line C.

From the dSAC form and Equation 7 we derive parameterized mapping functions which are functions of the Consumer iteration point and vectors of parameters: $f(y, \alpha)$. Vector of parameters α is used to uniformly specify a set of unique mapping functions which exist for every instance of a dynamic program, thus capturing the unknown information at compile-time. Values of vector α will be determined at run-time during the execution of a PN. The derivation of parameterized mapping functions is our novel contribution.

For example, for the P1/P3 pair, the mapping function is:

$$f_{P1P3} : \mathbb{Z}^4 \rightarrow \mathbb{Z} : k = \begin{pmatrix} 0 & 1 & 0 & 0 \end{pmatrix} (i_3, j_3, \alpha_i, \alpha_j)^{\mathbf{t}}.$$

For the P2/P3 pair, the mapping function is:

$$f_{P2P3} : \mathbb{Z}^4 \rightarrow \mathbb{Z}^2 : (i_2, j_2)^{\mathbf{t}} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} (i_3, j_3, \alpha_i, \alpha_j)^{\mathbf{t}}.$$

The domains of the mapping functions derived from Equation 8 are: $\mathbf{D}(f_{P2P3}) = \{i_3 \geq \alpha_i \wedge j_3 = \alpha_j\}$, $\mathbf{D}(f_{P1P3}) = \{i_3 < \alpha_i \vee j_3 \neq \alpha_j\}$, where variables $i_3, \alpha_i \in [1, 4]$, and $j_3, \alpha_j \in [i_3, 4]$.

Finally, we use the parameterized mapping functions in a novel procedure for communication model identification in PNs derived from WDPs. To identify the communication model of an arbitrary P/C pair, we specify the *Reordering Problem* (RP) and the *Multiplicity Problem* (MP) shown in Figure 7 which correspond to Definitions III.2 and III.3, respectively.

The meanings of all constraints in Figure 7 are the same as in Figure 5. A major difference is that *parameterized* mapping functions are used. The whole innovation in our communication model identification procedure in WDP programs is to model unknown information at compile-time in parameterized mapping functions.

$$\begin{cases} y^1, y^2 \in \mathrm{LmP}(\mathbf{D}(f)), \\ y^1 \prec y^2, \\ f(y^1, \alpha^1) \succ f(y^2, \alpha^2). \end{cases} \qquad \begin{cases} y^1, y^2 \in \mathbf{D}(f) & (c1) \\ y^1 \neq y^2, & (c2) \\ f(y^1, \alpha^1) = f(y^2, \alpha^2). & (c3) \end{cases}$$

(a) Reordering Problem (RP) (b) Multiplicity Problem (MP)

Fig. 7: Reordering and Multiplicity Problems for WDP programs.

The definition of the LmP set used in Figure 7(a) is given in Figure 8(a). It is an integer linear problem similar to the problem shown in Figure 3. The differences between the two formulations of LmP problems are that in the problem

shown in Figure 8(a) parameterized mapping function is used and this problem finds lexicographically minimal Consumer's iteration points *and* parameters α. For example, consider the P2/P3 pair formed by statements S2 and S3 from the dSAC shown in Figure 6(a). The LmP problem for the P2/P3 pair is illustrated in Figure 8(b). The solution of this problem is $\mathrm{LmP}(\mathbf{D}(f_{P2P3})) = \{(i_2, j_2, i_2, j_2) \in C(N) \mid 1 \leq i_2 \leq j_2 \leq 4\}$.

$$\textbf{objective}: \quad (y_m, \alpha_m) = min_{\mathrm{lex}}\{y(x), \alpha(x)\},$$

$$\textbf{subject to}: \quad \begin{cases} y \in C(N), \\ x = f(y, \alpha). \end{cases}$$

(a) LmP ILP problem for WDP programs

$$\begin{cases} i_3 \geq \alpha_i, j_3 = \alpha_j, \\ i_2 = \alpha_i, j_2 = \alpha_j, \\ 1 \leq i_3 \leq j_3 \leq 4, \\ 1 \leq i_2 \leq j_2 \leq 4. \end{cases}$$

(b) An example of LmP problem

Fig. 8: Statement of the LmP problem with an example.

Examples of applying RP and MP problems to our running example `ProgramD` depicted in Figure 2(a) are shown in Figure 9. The RP and MP problems are formulated for the P2/P3 pair formed by statements S2 and S3 from the Figure 6(a), respectively. Clearly, the RP problem shown in Figure 9(a) does not have an integer solution, because constraints $(c3)$ and $(c4)$ contradict each other. Therefore, the communication model of P2/P2 pair is **in-order**. The MP problem illustrated in Figure 9(b) has an integer solution: for some $i_3^1 \neq i_3^2$, there can be $\alpha_i^1 = \alpha_j^2$ which satisfy $(c4)$, and, thus, the communication model of the channel has a **multiplicity**.

$$\begin{cases} i_3^1 = \alpha_i^1, j_3^1 = \alpha_j^1, \\ i_3^2 = \alpha_i^2, j_3^2 = \alpha_j^2, \\ (i_3^1, j_3^1) \prec (i_3^2, j_3^2), \\ (\alpha_i^1, \alpha_j^1) \succ (\alpha_i^2, \alpha_j^2). \end{cases}$$

(a) RP for P2/P3 pair

$$\begin{cases} i_3^1 \geq \alpha_i^1, j_3^1 = \alpha_j^1, & (c1) \\ i_3^2 \geq \alpha_i^2, j_3^2 = \alpha_j^2, & (c2) \\ (i_3^1, j_3^1) \neq (i_3^2, j_3^2), & (c3) \\ (\alpha_i^1, \alpha_j^1) = (\alpha_i^2, \alpha_j^2). & (c4) \end{cases}$$

(b) MP for P2/P3 pair

Fig. 9: Examples of RP and MP problems for P2/P3 pair.

V. Conclusion

In this paper we presented a novel procedure for communication models identification in PN derived from Weakly Dynamic Programs. We used Fuzzy Array Dataflow Analysis to analyze original WDP program and as a consequence, we were able to specify the program in dynamic Single Assignment Code (dSAC) which is our extension of classical SAC. Based on dSAC, we derived parameterized mapping functions in P/C pairs of a PN. Parameters are used to uniformly specify a set of unique mapping functions which exist for every instance of dynamic program, thus, capturing unknown information at compile-time. We showed how values of parameters are being changed at run-time during evaluation of a PN. For the communication model identification, we formulated a new *Reordering Problem* which uses the *Lexicographically minimal Preimage*. This approach allows us to realize communication models more efficiently. Our novel contributions are the derivation of parameterized mapping functions and communication model identification procedure while translating WDPs into equivalent PNs.

References

[1] W. Wolf, A. A. Jerraya, and G. Martin, "Miltiprocessor System-on-chip (mpsoc) Technology," *IEEE TCAD*, vol. 27, no. 10, 2008.

[2] G. Martin, "Overview of the mpsoc design challenge," in *DAC '06*. New York, NY, USA: ACM, 2006, pp. 274–279.

[3] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," in *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974.

[4] A. Turjan, B. Kienhuis, and E. Deprettere, "Translating Affine Nested-loop Programs to Process Networks," in *Proc. CASES'04*, Washington D.C., USA, Sep. 23-25 2004.

[5] T. Stefanov, "Converting Weakly Dynamic Programs to Equivalent Process Network Specifications," 2004, PhD thesis, Leiden University, The Netherlands, ISBN: 90-9018629-8.

[6] P. Feautrier, "Dataflow Analysis of Scalar and Array References," *Journal of Parallel Programming*, vol. 20, no. 1, pp. 23–53, 1991.

[7] A. Turjan, "Compiling nested loop programs to process networks," 2007, PhD thesis, Leiden University, The Netherlands.

[8] P. Feautrier and J.-F. Collard, "Fuzzy Array Dataflow Analysis," Ecole Normale Superieure de Lyon, Tech. Rep., 1994, eNS-Lyon/LIP N^o 94-21.