Deriving Process Networks from Weakly Dynamic Applications in System-Level Design

Todor Stefanov
Leiden Embedded Research Center
Leiden Institute of Advanced Computer Science
Leiden University, The Netherlands
stefanov@liacs.nl

Ed Deprettere
Leiden Embedded Research Center
Leiden Institute of Advanced Computer Science
Leiden University, The Netherlands
edd@liacs.nl

ABSTRACT

We present an approach to the automatic derivation of executable Process Network specifications from Weakly Dynamic Applications. We introduce the notions of Dynamic Single Assignment Code, Approximated Dependence Graph, and Linearly Bounded Sets to model and capture weakly dynamic (data-dependent) behavior of applications at the task-level of abstraction. Process Networks are simple parallel processing models that match the emerging multi-processor architectures in the sense that the mapping of Process Network specifications of applications onto multi-processor architectures can be done in a systematic and transparent way.

Categories and Subject Descriptors

I.6.3 [**Simulation and Modeling**]: Applications; J.6 [**Computer-aided Engineering**]: Computer-aided design (CAD)

General Terms

Algorithms, Design

Keywords

System-Level Design, Heterogeneous Embedded Systems, Kahn Process Networks, Weakly Dynamic Applications

1. INTRODUCTION

Single-chip embedded systems are increasingly becoming *heterogeneous systems*, i.e., systems composed of fully programmable components (microprocessors), reconfigurable components (FPGAs), and dedicated hardware blocks. Typically, these components are linked via some kind of communication structure (high-speed bus, multiple buses or programmable network) forming a multi-processor architecture. Mapping applications onto a multi-processor architecture is a key issue in the emerging system-level and platform-based design methodologies. Today, system designers experience significant difficulties because the way an application is specified by the application developer does not match the way multi-processor architectures operate. The applications are typically specified using an imperative model of computation, i.e., programming languages like C/C++. Although the imperative model of computation is a natural model for application specification it does not reveal parallelism due to its inherent sequential nature. This fact makes the mapping of an application onto a parallel multi-processor architecture very difficult. On the other hand, if the application is specified using a parallel model of computation (MoC) then the mapping will be done in a systematic and transparent way using a disciplined approach [12], but specifying an application using a parallel MoC is difficult, not well understood by application developers, and a time consuming process. That is why application developers still prefer to specify an application using the imperative model of computation, which is well understood, regardless the fact that this model is not suitable for mapping an application onto a parallel multi-processor architecture.

The facts, given above, suggest that there is a gap between the way applications are currently specified (as sequential programs in C/C++ or Matlab) and the way they should be specified (using a parallel model of computation) [12] in order to allow a systematic and transparent mapping onto parallel multi-processor architectures. Although, many parallel models of computation exist [9][10], in this paper we focus on the Process Network model of computation [7] because its operational semantics are simple, yet general enough, to specify conveniently *stream-oriented* data processing that fits nicely with the application domain we are interested in - multimedia and signal processing applications. Moreover, for this application domain a prior work described in [2][16][11] reports that indeed the Process Network (PN) model is very suitable for specifying and mapping systematically and efficiently applications onto multi-processor architectures. The PN model expresses an application in terms of distributed control (no global scheduler is present) and distributed memory that are key requirements to take advantage of the parallel resources available in multi-processor architectures.

Our previous work presented in [8] provides some techniques to bridge the gap mentioned above but these techniques are limited to automatic derivation of Process Networks only from applications described as *static* affine nested loop programs. Such programs are important in Scientific, Matrix Computation and Adaptive Signal Processing applications. In this paper, however, we focus on techniques that support automatic derivation of Process Networks from *Weakly Dynamic Applications* (WDA). We define a WDA as a **task-level** sequential program where:
A) the control structures in the program are: *for-loops* with upper and lower bounds as affine functions of iterators of other loops and parameters; *if-then-else* constructs with **no restrictions on the**

CODES+ISSS'03, October 1–3, 2003, Newport Beach, California, USA.

condition - the condition of the *if* may be an arbitrary function of loop iterators and/or data variables.
B) The indexing of data variables (arrays) must be an affine function of *for-loop* iterators and possible parameters.

Notice that if we constrain the condition of *if-then-else* constructs to be an affine function of loop iterators and parameters then our WDA reduces to a static affine nested loop application. Therefore, the techniques we present in this paper extend significantly the class of applications that can be handled by [8]. For example, our techniques can handle not only Scientific, Matrix Computation and Adaptive Signal Processing applications but also media applications with dynamic (data dependent) behavior such as the JPEG codecs, the MPEG codecs, etc.

As a simple example, consider the WDA shown in Figure 1. This

```
1  %parameter N 8 16;          7  for i = 1:1:N,
2                              8   if t(i) <= 0,
3  for i = 1:1:N,              9    [x(i)] = F2( x(i) );
4   [x(i), t(i)] = F1(...);    10  end
5  end                         11  [...] = F3( x(i) );
6                              12 end
```

Figure 1: Pseudo code of simple Weakly Dynamic Application.

application consists of three function calls[1] $F1$, $F2$ and $F3$. These function calls execute tasks that communicate data via the array $x(i)$. Every element of the array can be anything from a scalar value to very complicated data structures. The execution of $F2$ depends on the condition at line 8. This condition is data dependent because of the variable $t(i)$. The values of this variable are not known at compile time, i.e., they are not known before the actual execution of the program. This fact makes the program to have dynamic behavior, unpredictable at compile time. A challenging problem is how to analyze and transform this kind of dynamic programs at compile time in order to derive automatically executable Process Network specifications. In this paper we address this problem and present a systematic solution approach.

1.1 Paper Contributions

We present a novel systematic and step-wise approach that allows automatic derivation of executable Process Network specifications from Weakly Dynamic Applications. New notions like Dynamic Single Assignment Code, Approximated Dependence Graph, and Linearly Bounded Sets have been introduced in our approach in order to capture and model weakly dynamic behaviors of an application. Most of the steps in our approach have been implemented in a prototype software framework called COMPAANPRO. The approach has been applied and validated successfully on a real-life application - Motion-JPEG encoder.

1.2 Related Work

Previous work on automatic derivation of Process Networks has been presented in [13][8]. This work focuses on deriving Kahn Process Network (KPN) specifications from applications described as static parameterized affine nested loop programs. In contrast, the work presented in this paper deals with a more general class of applications, i.e., weakly dynamic applications from which KPN specifications are derived automatically.

Kahn Process Networks are supported by the Ptolemy II framework [10] and the YAPI environment [3] for concurrent modeling and design of applications and systems. The designer has to specify manually the application as a Kahn Process Network and to give this network as an input to the Ptolemy II or YAPI simulation and verification engines. In many cases specifying manually application as a Kahn Process Network is a very time consuming and error prone process. Our work, presented in this paper, can be used as a front-end tool by Ptolemy II or YAPI. This will speedup significantly the modeling effort when Kahn Process Networks are used as well as modeling errors will be avoided because our techniques guarantee correct-by-construction generation of Kahn Process Networks.

The work presented in [2][16][11] uses Kahn Process Networks to model applications and to explore the mapping of these applications onto multi-processor architectures. This work clearly indicates that the application modeling is done manually starting from a sequential C code as well as that significant amount of time (a few weeks) is spent by designers on transforming correctly the sequential C code into Kahn Process Networks. This fact slows down the design space exploration process. The work presented in this paper gives a solution for fast automatic derivation of Kahn Process Networks from sequential C code that will contribute for faster design space exploration.

1.3 Paper Organization

In the next section, we give an overview of our step-wise approach to derive Process Networks from Weakly Dynamic Applications. In Section 3, every step in the approach is explained in detail. Section 4 presents some results we have obtained by applying our approach on a real-life application, i.e., Motion-JPEG encoder. Finally, we draw some conclusions in Section 5.

2. SOLUTION APPROACH: OVERVIEW

In this section, we present an overview of our approach to solve the problem of deriving process networks from weakly dynamic applications (WDA) as stated in Section 1. Our approach consists of three main steps shown in Figure 2.

We start with a WDA specified in Matlab or C and transform it in a *dynamic Single Assignment Code* (dSAC) representation - STEP 1 in Figure 2. There is a difference between our dSAC and the classical single assignment code (SAC) used in the compiler community and systolic array community. The classical SAC is defined as a program in which every variable is written only once whereas our dSAC has the property that: 1) every variable in the code is written *at most* once because of the dynamics in the application; 2) for some variables, it is not known whether or not they will be written or read before the actual execution of the code. The dSAC reveals all possible data-dependencies between the functions in the WDA. Some of the data dependencies in the dSAC are not exactly defined, i.e., they depend on variables having values that are not known at compile time. In Section 3.1, we give more details about the dSAC and we show a simple example.

The second step in our approach is to convert the dSAC into another representation that is a formal model. This model consists of two annotated graph structures, namely *Approximated Dependence Graph* (ADG) and *Schedule Tree* (STree). The ADG and the STree capture all the information that is present in the dSAC in a formal way. As a consequence, formal operations can be easily defined and applied on the ADG and the STree instead of the dSAC.

The ADG contains all the information that is related to the data dependencies between the functions in the dSAC. The data dependencies are approximated, i.e., the exact data dependencies are not known at compile time. Because of this, our ADG model is more general than the reduced dependence graph models that are used to represent static programs. If the data dependencies are known at compile time then the ADG is actually a polyhedral reduced de-

[1] At some places in this paper we use the word *function* instead of *function call* for the sake of brevity.

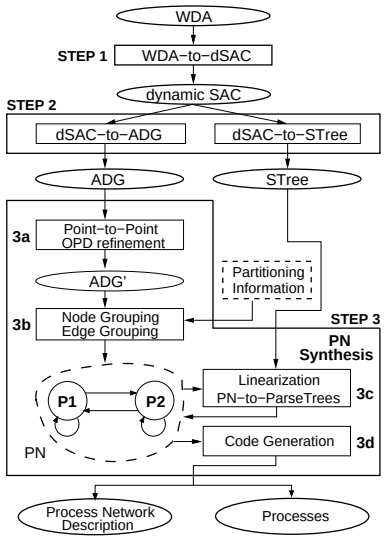

Figure 2: An approach to derive Process Networks from Weakly Dynamic Applications.

pendence graph (PRDG) [13]. In Section 3.2, we define the ADG model. Also, we briefly describe our procedure to derive the ADG from a dSAC and we give an example.

The STree contains all the information about the execution order between the functions in the dSAC. The STree represents one valid schedule between all these functions that we call global schedule. From the STree a local schedule between any arbitrary set of the functions in the dSAC can be obtained by pruning operations on the STree. In Section 3.3, we define the STree and give an example.

In the final step (STEP 3 in Figure 2) of our approach a Kahn Process Network (KPN) [7] is synthesized. A KPN consists of concurrent processes that communicate with each other over unbounded FIFO channels. Every process is specified as a sequential program. The synthesis of this program is based on information derived from the ADG and the STree. The synchronization between the processes is accomplished by blocking reads. The computational and communication workloads are distributed over the processes and the channels in accordance with some partitioning information. Such information can be given manually or delivered by the *partitioning information* box shown in Figure 2. This box implements some design space exploration procedures and/or some optimization procedures. If partitioning information is not available then we use the following partitioning by default: 1) for every node in the ADG a process is generated; 2) for every edge in the ADG a channel is generated. In Section 3.4 the process network synthesis[2] is described in details.

3. SOLUTION APPROACH: DETAILS

In this section, we explain in more details some of the models and techniques used in the approach presented in Section 2. Also, some examples are given for the sake of clarity.

3.1 Dynamic Single Assignment Code

The Dynamic Single Assignment Code (dSAC) derived from a weakly dynamic application (WDA) is a program in which every variable is written at most once. This property implies that some of the variables may be not written at all. This is because of the dynamic control structures in the application where the conditions are data-dependent, i.e., the outcome of the conditions is not known at compile time.

In order to derive a dSAC from a WDA we have to find all possible data dependencies between the functions in the WDA. Because of the dynamic control structures in the WDA an exact array dataflow analysis [5] can not be performed to find the data dependencies. The approach we follow to find the data dependencies in case of WDA is based on parametric integer programming (PIP) [4]. We use the same technique as in the exact array dataflow analysis for building a PIP system but we add to this system constraints with parameters for the dynamic control structures in the WDA. By introducing additional constraints with parameters in a PIP system we "mask" the information that is not known at compile time. Because of this, we find approximated data dependencies. Our approach to find the approximated data dependencies is based on the approach known in the literature as *Fuzzy Array Dataflow Analysis* (FADA) [6].

Using approximated data dependencies we have found a procedure to generate a dSAC that is out of the scope of this paper. As an example, in Figure 3 we show the output of this procedure for the simple program shown in Figure 1.

```
1  %parameter N 8 16;                16   [out_0] = F2(in_0);
2                                    17   [x_2(i)] = opd(out_0);
3  for i = 1:1:N,                    18   [ ctrl(i) ] = opd( i );
4    ctrl(i) = N+1;                  19  end
5  end                               20
6  for i = 1:1:N,                    21   C = ipd( ctrl(i) );
7   [out_0, out_1] = F1(...);        22  if i = C,
8   [x_1(i)] = opd(out_0);           23   [in_0] = ipd(x_2(C));
9   [t_1(i)] = opd(out_1);           24  else
10 end                               25   [in_0] = ipd(x_1(i));
11                                   26  end
12 for i = 1:1:N,                    27
13  [t_1(i)] = ipd(t_1(i));          28  [out_0] = F3(in_0);
14  if t_1(i) <= 0,                  29  [...] = opd(out_0);
15   [in_0] = ipd(x_1(i));           30 end
```

Figure 3: Example of Dynamic Single Assignment Code.

We call the code in Figure 3 dynamic SAC because if we consider, for example, line 17 we do not know at compile time at which iteration the elements of the array $x\_2(i)$ will be written. The only thing known is that they will be written at most once. Moreover, for every execution of function $F3$ in line 28, its input is not known at compile time. The input has to be determined at run time by the code lines 22-26. Both cases described above never occur in the classical SAC cases.

Another new feature of the dSAC is the presence of parameters that originate form the data dependent control constructs in a weakly dynamic application (WDA). In order to keep the functionality of the dSAC equivalent to the functionality of the original WDA, the values of these parameters have to be changed dynamically. Our approach to accomplish the dynamic change is to introduce, for every such parameter, a control variable that stores the correct value of the parameter for every iteration. For example, C in the dSAC shown in Figure 3 is a parameter emerging form the *if*-statement in line 8 of the original program shown in Figure 1. This *if*-statement also appears in the dSAC in line 14. The dynamic change of the value of C is accomplished by the lines 18 and 21 in Figure 3. The control variable $ctrl(i)$ in line 18 stores the iterations for which the data dependent condition that introduces C is true. Also, the variable $ctrl(i)$ is used in line 21 to assign the correct value to C for the current iteration.

The dSAC we generate contains functions called **ipd** and **opd** -

[2]The process network synthesis is not limited to the generation of Kahn Process Networks only. With small modifications other process networks that have different interprocess communication and synchronization mechanisms can be generated.

see Figure 3. These functions just propagate the value of its input to its output and they play a role in the conversion of a dSAC to an approximated dependence graph presented in the next section.

3.2 Approximated Dependence Graph

In this section we give a formal definition of our Approximated Dependence Graph (ADG) model followed by a procedure to derive it from a dSAC, and an example.

Definition 3.1 (approximated dependence graph)
An Approximated Dependence Graph (ADG) is given by a tuple $ADG = (Nodes, Edges)$ where: $Nodes = \{N_{i=1..M}\}$ is a set of nodes. $Edges = \{E_{j=1..P}\}$ is a set of edges.

Definition 3.2 (node)
A node in the ADG is given by a tuple $N = (I_N, O_N, F_N, ND_N)$ where: $I_N = \{p_{k=1..K}\}$ is a set of input ports. $O_N = \{q_{l=1..L}\}$ is a set of output ports. F_N is a tuple $F_N = (F, in, out)$, where in and out are sets of variables and $F : in \rightarrow out$ is a function. ND_N is the node domain of N defined by a linearly bounded set (LBS - Definition 3.6).

Definition 3.3 (input port)
An input port is given by a tuple $p = (V_p, A_p, IPD_p)$ where: V_p is an n-dimensional variable associated with the port. A_p is a variable binding the port to the function in the node to which the port belongs if $A_p \in in \wedge A_p \neq V_p$. If $A_p = V_p$ then A_p is a variable binding the port to the node domain. A_p is a variable binding the port to IPDs of other ports if $A_p \notin in \wedge A_p \neq V_p$. IPD_p is the input port domain of p defined by a LBS (Definition 3.6).

Definition 3.4 (output port)
An output port is given by a tuple $q = (V_q, A_q, OPD_q)$ where: V_q is an m-dimensional variable associated with the port. $A_q \in out$ is a variable binding the port to the function in the node to which the port belongs. OPD_q is the output port domain of q defined by a LBS (Definition 3.6).

Definition 3.5 (edge)
An edge in the ADG is a triple $E = (q, p, M)$ where:
$q = (V_q, A_q, OPD_q)$ is an output port. $p = (V_p, A_p, IPD_p)$ is an input port. $V_p = V_q$, i.e., variables V_p and V_q have the same names and equal dimensions. $M : i_p \rightarrow i_q$ is an affine mapping where $i_p \in IPD_p$ and $i_q \in OPD_q$.

Definition 3.6 (linearly bounded set)
Let be given four sets of functions
$S1 = \{f_x^1(i) \mid x = 1..|S1|,\ i \in Z^n\}$, $S2 = \{f_x^2(i) \mid x = 1..|S2|,\ i \in Z^n\}$, $S3 = \{f_x^3(i) \mid x = 1..|S3|,\ i \in Z^n\}$, $S4 = \{f_x^4(i) \mid x = 1..|S4|,\ i \in Z^n\}$, an integral $m \times n$ matrix A and an integral n-vector b. A *linearly bounded set* (LBS) is a set of points $LBS = \{\ i \in Z^n \mid A.i \geq b,$

$$if\ S1 \not\equiv \emptyset \Rightarrow \forall_{x=1..|S1|},\ f_x^1(i) \geq 0,$$
$$if\ S2 \not\equiv \emptyset \Rightarrow \forall_{x=1..|S2|},\ f_x^2(i) \leq 0,$$
$$if\ S3 \not\equiv \emptyset \Rightarrow \forall_{x=1..|S3|},\ f_x^3(i) > 0,$$
$$if\ S4 \not\equiv \emptyset \Rightarrow \forall_{x=1..|S4|},\ f_x^4(i) < 0\ \}.$$

The set of points $B = \{\ i \in Z^n \mid A.i \geq b\ \}$ is called *linear bound* of the LBS and the set $S = S1 \cup S2 \cup S3 \cup S4$ is called *filtering set*. Every $f_x^j(i) \in S$ can be an arbitrary function of i.

Our procedure to convert a dSAC into the ADG model defined above consists of two steps: 1) the dSAC is converted to a syntax tree [1]; 2) the syntax tree is parsed and all the elements of the ADG are created and specified in accordance with the relations given below:
1) for every function $[\ arg\ ] = \mathbf{ipd}(\ var\ )$ in the dSAC there exists a corresponding input port $p = (V_p, A_p, IPD_p)$ in the AGD, where $V_p = var$, $A_p = arg$ and IPD_p is the set of iterations in which the function **ipd** is executed.
2) for every function $[\ var\ ] = \mathbf{opd}(\ arg\ )$ in the dSAC there exists a corresponding output port $q = (V_q, A_q, IPD_q)$ in the AGD, where $V_q = var$, $A_q = arg$ and IPD_q is the set of iterations in which the function **opd** is executed.
3) for every pair $[\ arg_i\ ] = \mathbf{ipd}(\ var_i\ )$ and $[\ var_o\ ] = \mathbf{opd}(\ arg_o\ )$ in the dSAC there exists an edge $E = (q, p, M)$ in the ADG if var_i and var_o have the same name and dimension. q is the output port in the ADG corresponding to **opd** and p is the input port in the ADG corresponding to **ipd**. M is an affine mapping that gives for every iteration in which the **ipd** consumes a value, the corresponding iteration in which this value is produced by **opd**;
4) for every function $[\ oArg\ ] = \langle\mathbf{name}\rangle(\ iArg\ )$ in the dSAC with $\langle\mathbf{name}\rangle$ neither **ipd** nor **opd** there exists a corresponding node $N = (I_N, O_N, F_N, ND_N)$ in the ADG. For every function $[\ arg\ ] = \mathbf{ipd}(\ var\ )$ in the dSAC: if $arg \in iArg$ then the corresponding input port p in the ADG belongs to I_N. For every function $[\ var\ ] = \mathbf{opd}(\ arg\ )$ in the dSAC: if $arg \in oArg$ then the corresponding output port q in the ADG belongs to O_N. The elements of F_N are related to the dSAC as $F_N.F = \langle\mathbf{name}\rangle$, $F_N.in \equiv iArg$ and $F_N.out \equiv oArg$; ND_N is the set of iterations in which the function $\langle\mathbf{name}\rangle$ is executed.

As an example, consider the dSAC shown in Figure 3. Applying the procedure described above on this dSAC we obtain the ADG shown in Figure 4-a). The ADG consists of three nodes and five edges. Nodes $N1$, $N2$, and $N3$ correspond to functions $F1$, $F2$, and $F3$ of the dSAC, respectively. Edges $ED1$, $ED2$, $ED3$, $ED4$, and $ED5$ correspond to possible data dependencies between $F1$, $F2$, and $F3$ through the variables $t_1(i)$, $x_1(i)$, $x_2(i)$, and $ctrl(i)$ of the dSAC. According to Definition 3.2 node $N2$ is the tuple $N2 = (I_{N2}, O_{N2}, F_{N2}, ND_{N2})$ where: $I_{N2} = \{p1, p2\}$, $O_{N2} = \{q1, q2\}$, $F_{N2} = (F2, \{in\_0\}, \{out\_0\})$, with $F2 : \{in\_0\} \rightarrow \{out\_0\}$. ND_{N2} represents the iterations i at which function $F2$ is executed in the dSAC. The exact iterations i are not known at compile time because of the dynamic condition at line 14 in the dSAC (Figure 3). That is why we introduce the notion of linearly bounded set (LBS-Definition 3.6) by which we approximate the unknown iterations i. So, ND_{N2} is the following LBS: $ND_{N2} = \{i \in Z \mid 1 \leq i \leq N \wedge 8 \leq N \leq 16,\ t\_1(i) \leq 0\}$. The linear bound of this LBS is the polytope $B = \{1 \leq i \leq N \wedge 8 \leq N \leq 16\}$ that captures the information that we know at compile time about the bounds of the iterations i. The variable $t\_(i)$ is interpreted as an unknown function of i called filtering function whose output is determined at run time. Introducing the LBS notion in our ADG model to capture the dynamic behavior of the dSAC is to the best of our knowledge a novel approach.

Because of the semantics of the LBS, described above, we call the graph in Figure 4-a) approximated dependence graph (ADG). Another reason to call it that way is because the probability that some data dependencies exist can not be decided 100% at compile time. For example, the edge $ED3$ suggests that there might be a data dependency between $F1$ and $F3$ through the variable $x_1(i)$ but this depends on the dynamic condition at line 14 of the dSAC in Figure 3. If this condition is always true at run time the data dependency between $F1$ and $F3$ does not exist.

3.3 Schedule Tree

Definition 3.7 (schedule tree)
Let be given a syntax tree [1] $Tree = (N, E)$ derived from a dSAC. A Schedule Tree (STree) is a syntax tree $STree = (N_S, E_S)$, where node set $N_S \subset N$ and edge set $E_S \subset E$. The topology of the $STree$ represents control structure of a program that executes

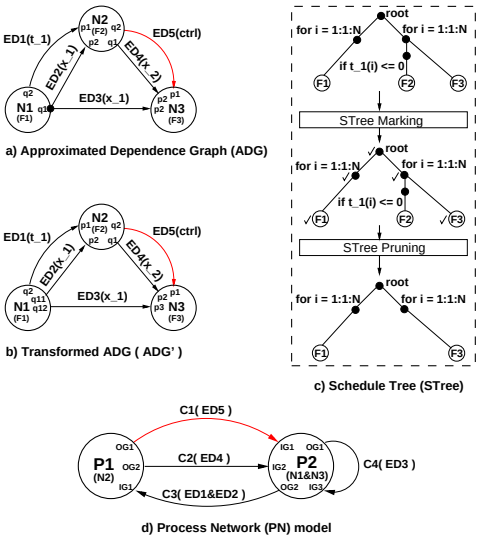

Figure 4: Examples of a) Approximated Dependence Graph (ADG) model; b) Transformed ADG; c) Schedule Tree and Transformations; d) Process Network model.

the functions [ ] = ⟨**name**⟩( ) of the dSAC in a correct order. ⟨**name**⟩ is different from **ipd** and **opd**.

Consider the dSAC shown in Figure 3. The corresponding schedule tree (STree) is depicted at the top part of Figure 4-c). If we parse this tree top-down from left to right a program can be generated that gives a valid execution order (global schedule) among the functions $F1$, $F2$ and $F3$ which is the original order given by the dSAC. The procedure to obtain the STree from the dSAC is done in two steps: 1) the dSAC is converted to a syntax tree using a standard syntax parser [1]; 2) the STree is extracted from the syntax tree by removing all the nodes and edges that are not related to nodes $F1$, $F2$ and $F3$.

3.4 Process Network Synthesis

In this section, we present the final step of our approach in which we synthesize a process network - see STEP 3 in Figure 2. Our synthesis approach is mainly a translation of the ADG model and STree model into a process network (PN) model. The structure of our PN model is defined in Section 3.4.1. Figure 2 shows that STEP 3 consists of four sub-steps. In sub-step **3a** we apply two transformations on the ADG, namely *Point-to-Point* and *OPD refinement*. This transformations have to be done because of two reasons:
1) On the one hand, our synthesis approach translates an ADG model into a PN model, where there is one-to-one correspondence between the input ports of the ADG and the input ports of the PN. The same is true for the output ports. The ADG may have several input ports connected to a single output port. On the other hand, we focus on the synthesis of a special class of process networks - Kahn Process Networks where every input port has to be connected to only one unique output port. This fact requires that we have to change the topology of the ADG such that every input port gets connected to only one unique output port - *Point-to-Point* connection. The change of the topology is always possible without changing the semantics of the ADG model. As an example, consider the ADG depicted in Figure 4-a). Two edges start from port $q1$ of node $N1$. By applying the point-to-point transformation we get the ADG shown in Figure 4-b). Now, in the transformed ADG (ADG'), node $N1$ has output ports $q11$ and $q12$ instead of port $q1$. Ports $q11$ and $q12$ are copies of port $q1$. This means that all the elements (Definition 3.4) of the ports $q11$ and $q12$ are identical with the elements of port $q1$.
2) Every output port in the PN has to send a token via the corresponding channel if the probability that the token will be needed in the process that reads this channel is not zero. This is accomplished by the transformation *OPD refinement*. This transformation is applied on every OPD that is associated with an output port in the ADG before the ADG model is translated to the PN model. The transformations *Point-to-Point* and *OPD refinement* are presented in Section 3.4.2;

In sub-steps **3b** and **3c** (Figure 2), the process network model (PN) is created gradually by creating the topology of the PN - sub-step 3b, followed by creating the behavior of the PN - sub-step 3c. The topology of the PN is created by grouping nodes and edges of the ADG into processes and channels in the PN. The grouping is based on the partitioning information delivered by the dashed box in Figure 2. In general, there is no limitation of grouping ADG nodes into PN processes. Any arbitrary grouping is possible. Grouping ADG edges into PN channels has to be performed after the PN processes are defined by the node grouping. Also, the edge grouping has to obey the following rule: All the edges that we want to group in a channel have to start form the same process, say P_i, and have to end at the same process, say P_j. An example of creating a topology of a PN by node grouping and edge grouping is given in Figure 4-d). The PN is created by grouping nodes $N1$ and $N3$ of the ADG in Figure 4-b) into process $P2$, as well as node $N2$ into process $P1$. The group of nodes of $P1$ consists of only one node $N2$. After that, edge grouping is performed where $ED1$ and $ED2$ form the channel $C3$. The channels $C1$, $C2$, and $C4$ are assigned one edge each, i.e., $ED5$, $ED4$, and $ED3$, respectively.

In sub-step **3c** in Figure 2 the behavior of the PN is created. A procedure called *Linearization* deals with the communication behavior of every process in the PN. This procedure is built depending on the target class of process networks under synthesis - in our case Kahn Process Networks (KPN) where the processes communicate with each other over 1-dimensional unbounded FIFO channels. In Section 3.4.3 we give more details about the linearization.

Internally, a process in a KPN has, by definition [7], a sequential behavior meaning that the functions that have to be executed inside the process are executed in sequential order. The procedure *PN-to-ParseTree* (Figure 2), derives this order such that the PN execution is deadlock free and expresses it as a parse tree for every process in the PN. This procedure operates on the PN model and uses the information encoded in the STree defined in Section 3.3. We illustrate part of the PN-to-ParseTree procedure by a simple example.

Consider process $P2$ of the PN shown in Figure 4-d). This process is constructed by grouping nodes $N1$ and $N3$ of the ADG shown in Figure 4-b). This means that process $P2$ has to execute in sequential order functions $F1$ and $F3$ associated with nodes $N1$ and $N3$, respectively. To find the order we use the schedule tree (STree) shown at the top in Figure 4-c). First, an operation called STree Marking finds the $F1$ and $F3$ leafs in the STree and parses the tree from these leafs to the tree root, marking all the nodes in the path - see Figure 4-c). Second, an operation called STree Pruning prunes the marked STree by removing all the unmarked nodes of this tree. The resultant tree shown at the bottom in Figure 4-c) can be converted to a program by traversing it top-down from left to right. This program gives a valid sequential order (schedule) between $F1$ and $F3$ for process $P2$ that guarantees deadlock free execution of the process network to which $P2$ belongs. Finally, the procedure *PN-to-ParseTree* adds to the tree shown at the bottom in Figure 4-c) other nodes that correspond to control structures in the program mentioned above. These control structures specify from

which ports the functions $F1$ and $F3$ associated with nodes $N1$ and $N2$ get input data and to which ports they put the output data for every iteration i. For lack of space the complete tree generated by *PN-to-ParseTree* is not depicted in Figure 4-c).

The last sub-step of the PN synthesis (Figure 2-**3d**) is called *Code Generation*. In this sub-step an executable code of a Kahn process network is generated from the PN model. We use a software engineering technique called *Visitor* to visit the PN model structure and to generate the executable code. This code can be expressed in any programming language on top of which an environment to execute Kahn process networks is built. For example, the YAPI environment [3] in C++, SystemC, or the Ptolemy II framework [10] in Java.

3.4.1 Process Network Model

Definition 3.8 (process network)
A process network (PN) is given by a tuple $PN = (P, C)$ where: $P = \{P_{i=1..|P|}\}$ is a set of processes. $C = \{C_{j=1..|C|}\}$ is a set of channels.

Definition 3.9 (process)
A process in the PN is given by a tuple $P = (NP, IP, OP, ST)$ where: $NP = \{N_{m=1..M}\}$ is a set of nodes, with N_m as given in Definition 3.2. $IP = \{IG_{k=1..K}\}$ is a set of input gates. $OP = \{OG_{l=1..L}\}$ is a set of output gates. ST is a schedule tree that gives a valid execution order between the functions F associated with every N_m.

Definition 3.10 (input gate)
An input gate is given by a tuple $IG = (I_{IG}, IK_{IG})$ where: $I_{IG} = \{p_{k=1..|I_{IG}|}\}$ is a set of ports, with p_k as given in Definition 3.3. $IK_{IG} = \{InKey_{p_{k=1..|I_{IG}|}}\}$ is a set of functions, where for every $p_k \in I_{IG}$ a function $InKey_{p_k} \in IK_{IG}$ is associated, $InKey_{p_k} : IPD_{p_k} \rightarrow Z^1$ is a one-to-one mapping.

Definition 3.11 (output gate)
An output gate is given by a tuple $OG = (O_{OG}, OK_{IG})$ where: $O_{OG} = \{q_{k=1..|O_{OG}|}\}$ is a set of ports, with q_k as given in Definition 3.4. $OK_{OG} = \{OutKey_{q_{k=1..|O_{OG}|}}\}$ is a set of functions, where for every $q_k \in O_{OG}$ a function $OutKey_{q_k} \in OK_{OG}$ is associated, $OutKey_{q_k} : OPD_{q_k} \rightarrow Z^1$ is a one-to-one mapping.

Definition 3.12 (channel)
A channel is given by a tuple $C = (OG, IG, E, CM)$ where: $OG = (O_{OG}, OK_{IG})$ is an output gate. $IG = (I_{IG}, IK_{IG})$ is an input gate. $E = \{E_{m=1..|E|}\}$ is a set of edges, where E_m is given by Definition 3.5. $CM \in \{1, 2, 3, 4\}$ is the communication mode of the channel. $CM = 1$ is out-of-order communication with coloring of tokens. $CM = 2$ is out-of-order communication without coloring of tokens. $CM = 3$ is in-order communication with coloring of tokens. $CM = 4$ is in-order communication without coloring of tokens.

3.4.2 ADG transformations

Point-to-Point transformation: Let an $ADG = (Nodes, Edges)$ be given. For every edge $E_i = (q_l, p_k, M) \in Edges$ create a new port $q_{l\_i} = q_l$, add this port to the node that contains q_l and modify E_i such that $E_i = (q_{l\_i}, p_k, M)$. Finally, for every node $N \in Nodes$ remove the original output ports q_l from O_N.

OPD refinement transformation: Let an $ADG = (Nodes, Edges)$ be given. According to Definition 3.5 every edge $E \in Edges$ is given by $E = ((V_q, A_q, OPD_q), (V_p, A_p, IPD_p), M)$, where OPD_q and OPD_p are linearly bounded sets with linear bounds $B_q = \{i_q \in Z^n | A_q.i_q \geq b_q\}$ and $B_p = \{i_p \in Z^m | A_p.i_p \geq b_p\}$, respectively. The transformation *OPD refinement* derives for every $E \in Edges$ a new linearly bounded set $OPD_q^{'} = OPD_q \cap M(B_p)$, where $M(B_p)$ is the image of B_p in Z^n defined by the affine mapping M.

3.4.3 Creating the PN behavior

Formally, in a Kahn Process Network (KPN) the functions executed inside a process communicate data with functions inside other processes over channels that are 1-dimensional arrays with FIFO access. However, our approach has to derive KPNs from weakly dynamic applications (WDA) in which all these functions communicate data between each other via variables that are N - dimensional arrays with random access. In order to keep the functionality of the KPN the same as the functionality of the WDA, we apply a procedure called *Linearization* that adds to the PN model information that is necessary to realize models of communication with 1-dimensional FIFO access arrays equivalent to the communication with N-dimensional random access arrays. This procedure extends the work presented in [14]. Our PN model supports four models of communication with 1-dimensional FIFO access arrays:
1) out-of-order communication with coloring of tokens - this model is used when the order of the tokens (data) written in the FIFO channel is different than the order the tokens have to be read. Also, the number of tokens that will be written or read to/from the channel is not known at compile time. In order to keep the correct behavior of the KPN, every token is tagged by a unique number (color) that is used to reorder the tokens while reading them form the channel;
2) out-of-order communication without coloring of tokens - this model is used when the order of the tokens (data) written in the FIFO channel is different than the order the tokens have to be read. The number of tokens that will be written or read to/from the channel is known at compile time. Coloring of tokens is not necessary to do reordering;
3) in-order communication with coloring of tokens - this model is used when the order of the tokens (data) written in the FIFO channel is the same as the order the tokens have to be read. Also, the number of tokens that will be written or read to/from the channel is not known at compile time. Because of this more tokens can be written in the channel than needed at run time. Every token is tagged by a unique number (color) that is used to remove the tokens that are not needed while reading them form the channel;
4) in-order communication without coloring of tokens - this model is used when the order of the tokens (data) written in the FIFO channel is the same as the order the tokens have to be read. The number of tokens that will be written or read to/from the channel is known at compile time.

The reordering of tokens in models 1) and 2) is done by a special controller and a reordering memory located in the process that reads the tokens form the channel. The coloring of tokens in models 1) and 3) is based on the functions $InKey_{p_k}$ and $OutKey_{q_k}$ - see Definition 3.10 and Definition 3.11. The four communication models, described above, have a different cost of implementation in terms of required reordering memory and complexity of the control. The *out-of-order communication with coloring of tokens* model is the most expensive model but it is the most general model. This model can be used for every channel and the correct behavior of the PN is guaranteed. However, depending on the edges that belong to a particular channel, in some cases another less expensive communication model can be selected that still guarantees the correct behavior of the PN. We extended the work presented in [15] to find a procedure that detects at compile time the most optimal communication model for a given channel. In case the coloring of tokens is needed we derive the coloring functions $InKey_{p_k}$ and $OutKey_{q_k}$ at compile time as well.

4. RESULTS

Most of the steps in our approach, presented in Section 2 and Section 3, have been implemented in a prototype software framework called COMPAANPRO. The approach has been applied and validated successfully on a real-life application - MotionJPEG encoder (MJPEG). We started with publicly available sequential C code of the MJPEG. The code was structured by hand such that it meets the definition of a WDA given in Section 1. This took four days. After this preparation work which is a *one-time* effort only, our COMPAANPRO tool derived automatically an executable Kahn Process Network (KPN) specification from the structured MJPEG code in 30 seconds. For comparison, a KPN specification of an MJPEG encoder was derived by hand in [11] that took four weeks. The facts above show that the automated approach presented in this paper reduces significantly the time required to derive a KPN from real-life application.

COMPAANPRO generated the MJPEG network as C++ code in YAPI [3] format which allowed us to run the network and to verify its functional correctness. Without providing any partitioning information to the COMPAANPRO tool, the tool applies the partitioning strategy described at the end of Section 2. Using this strategy for the MJPEG, the computational workload was partitioned into 10 concurrent processes and the communication workload was distributed over 56 FIFO channels. We analyzed the network and concluded that the computational workload was very well balanced over the 10 concurrent processes. However, the distribution of the communication workload was not optimal. We found that the number of communication FIFO channels can be reduced from 56 to 30 FIFOs by merging some FIFOs without obstructing the exploited parallelism. The techniques presented in this paper support FIFO merging by grouping edges as discussed in Section 3.4. However, the selection of channels to be merged is currently still the designers choice.

5. CONCLUSIONS

In this paper, we presented a novel systematic approach that allows automatic derivation of executable Kahn Process Network (KPN) specifications from Weakly Dynamic Applications (WDA). By introducing new notions like Dynamic Single Assignment Code, Approximated Dependence Graph, and Linearly Bounded Sets in our approach, we extend the range of applications where KPNs can be derived automatically.

Our experiments with the MJPEG application show that the proposed approach, implemented in the COMPAANPRO tool, produces an important reduction of the time required to generate executable KPN specifications from real-life applications.

The presented approach includes only basic techniques that have to be used in order to derive automatically KPNs from WDAs. The results, we have obtained for the MJPEG application, indicated that some optimization techniques have to be added to the approach that will help with improving the quality of the generated KPNs in terms of optimal partitioning of the computation and communication workloads.

6. ACKNOWLEDGMENTS

This research is supported by PROGRESS, the embedded systems and software research program of the Dutch organization of Scientific Research NWO, the Dutch Ministry of Economic Affairs and the Dutch Technology Foundation STW.

7. REFERENCES

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[2] E. de Kock. Multiprocessor Mapping of Process Networks: A JPEG Decoding Case Study. In *Proc. 15th Int. Symposium on System Synthesis (ISSS'2002)*, pages 68–73, Kyoto, Japan, Oct. 2-4 2002.

[3] E. de Kock et al. YAPI: Application modeling for signal processing systems. In *Proc. 37th Design Automation Conference (DAC'2000)*, pages 402–405, Los Angeles, CA, June 5-9 2000.

[4] P. Feautrier. Parametric Integer Programming. Operations Research, 22(3):243-268, 1988.

[5] P. Feautrier. Dataflow Analysis of Scalar and Array References. *Int. Journal of Parallel Programming*, 20(1):23–53, 1991.

[6] P. Feautrier and J.-F. Collard. Fuzzy Array Dataflow Analysis. Technical report, Ecole Normale Superieure de Lyon, 1994. ENS-Lyon/LIP N^o 94-21.

[7] G. Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974.

[8] B. Kienhuis, E. Rijpkema, and E. F. Deprettere. Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures. In *Proc. 8th International Workshop on Hardware/Software Codesign (CODES'2000)*, San Diego, CA, USA, May 3-5 2000.

[9] E. Lee and A. Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 17(12):1217–1229, 1998.

[10] E. Lee et al. PtolemyII: Heterogeneous Concurrent Modeling and Design in Java. Technical report, University of California at Berkeley, 1999. UCB/ERL M99/40.

[11] P. Lieverse, T. Stefanov, P. van der Wolf, and E. Deprettere. System Level Design with SPADE: an M-JPEG Case Study. In *Proc. Int. Conference on Computer Aided Design (ICCAD'01)*, pages 31–38, San Jose CA, USA, Nov. 4-8 2001.

[12] A. Mihal and K. Keutzer. *Mapping Concurrent Applications onto Architectural Platforms*. in Networks on Chip (Chapter 3), Editors Axel Jantsch and Hannu Tenhunen, Kluwer Academic Publishers, 2003.

[13] E. Rijpkema. Modeling Task Level Parallelism in Piece-wise Regular Programs, 2002. PhD thesis, Leiden University, The Netherlands.

[14] A. Turjan, B. Kienhuis, and E. Deprettere. *Realizations of the Extended Linearization Model*. in Domain-Specific Embedded Multiprocessors (Chapter 9), Marcel Dekker, Inc., 2003.

[15] A. Turjan, B. Kienhuis, and E. Deprettere. A Technique to Determine Inter-process Communication in the Polyhedral Model. In *Proc. Int. Workshop on Compilers for Parallel Computers (CPC'03)*, Amsterdam, The Netherlands, Jan. 8-10 2003.

[16] P. van der Wolf, P. Lieverse, M. Goel, D. La Hei, and K. Vissers. An MPEG-2 Decoder Case Study as a Driver for a System Level Design Methodology. In *Proc. 7th Int. Workshop on Hardware/Software Codesign (CODES'99)*, Rome, Italy, May 3-5 1999.