Abstract—The synchronous model of computation (MoC) has been successfully used for the design of embedded systems having a local control like hardware circuits and single-threaded software, while its application to distributed parallel embedded systems is still a challenge. In contrast, other MoCs such as data-flow process networks (DPNs) directly match with these architectures. In this paper, we therefore present a translation of synchronous systems to data-flow process networks, thereby bridging the gap between synchronous and asynchronous MoCs. We use the resulting DPNs to generate CAL code for the OpenDF package, which offers important features for embedded system design.

I. INTRODUCTION

The development of efficient and reliable embedded systems is very challenging. Various methodologies have been proposed over the last decades, among them model-based design, which integrates various design tasks like simulation, verification and synthesis (of both hardware and software) by using a set of models instead of particular implementations. For the description of models, there is a large number of languages and formalisms available, which can be grouped by their underlying models of computation (MoC) [15, 19, 22], i.e. meta-models defining general rules for computation and communication.

The synchronous MoC [5] is such a model of computation, and it forms the basis of synchronous programming languages such as Esterel [6], Lustre [16] or Quartz [25]. These languages divide the system behavior into a sequence of reactions, which immediately gives rise to the definition of steps and clocks. Thereby, synchronous systems are well suited for formal verification and analysis due to their deterministic parallelism. As hardware synthesis from these languages is straightforward and their deterministic parallelism makes it possible to generate very efficient single-threaded software from concurrent programs, synchronous languages are a good choice for the development of embedded systems and other applications requiring precise timing, e.g. streaming applications or audio filter.

However, the synchronous languages also have some disadvantages – in particular, if the target architecture does not match its model of computation. First, computations in a synchronous system implicitly follow the data dependencies [7, 26] so that executing them requires schedules for the actions of a synchronous system. Second, the synchronous MoC usually implements fine-grained parallelism and a global clock, which makes software synthesis for multithreaded or distributed systems very challenging.

In contrast, data-flow process networks (DPN) are an un-timed model of computation, which models the system by a set of nodes that run concurrently and exchange information through point-to-point links. Thereby, data dependencies are explicitly modeled and computations are completely data-driven. As DPNs are inherently parallel and completely abstract from timing and clocks, they can be straightforwardly mapped to parallel and distributed platforms.

The main contribution of this paper is a translation from synchronous systems to data-flow process networks, thereby bridging the gap between two fundamentally different models of computations. This makes it simpler to map synchronous systems onto distributed target architectures. Furthermore, we propose to use DPNs as an abstract intermediate representation before final synthesis to concrete data-flow languages or multithreaded software. On the one hand, this separates concerns, since the conceptual part, the switch of the model of computation, is clearly separated from the technical part, which deals with the concrete APIs or differences on the target platforms. On the other hand, this also gives us the possibility to reuse this translation as a common step for various architectures.

We will show the feasibility of our approach by a translation from DPNs to CAL [14], which is a programming language of the OpenDF package. It provides many useful features, which will be also accessible to our synchronous systems: including the ability to create efficient multithreaded software and VHDL code from CAL programs [8, 18]. Furthermore, the OpenDF package also contains a simulator to investigate the behavior of single nodes and the complete system. A highlight of OpenDF is an approach for HW/SW co-design [24]. In particular, [24] describes how one can create code for heterogenous systems by translating nodes of a system separately to different platforms.

Our approach is clearly different to previous approaches which also targeted the generation of multi-threaded code from synchronous programs. Baudisch et al. showed how independent computations [2] and pipelining [3] can be used to generate multi-threaded programs from synchronous systems. However, their translation immediately created C code based on OpenMP, which imposes tight restrictions on the source code structure. More closer is the approach presented in [1], which partitions a synchronous program into concurrent

Daniel Baudisch, Jens Brandt and Klaus Schneider
Embedded Systems Group
Department of Computer Science
University of Kaiserslautern
http://es.cs.uni-kl.de
components. Although the structure of the partitioned program resembles a DPN, its semantics is different so that the nodes — in particular the designated merge nodes — cannot be translated to DPN nodes without a heavy encoding. Moreover, we do not only translate synchronous systems to DPNs, but we do also translate the resulting DPNs to a data-flow language, showing the applicability of our approach.

The rest of this paper is structured as follows: Section II introduces synchronous guarded actions, which is our representation of synchronous systems in this paper. Section III considers data-flow process networks. Then, Section IV explains the translations from synchronous systems to DPNs, thereby highlighting the switch from one MoC to another one. Section V shows how the resulting code can be used for a concrete target language, which is in our case CAL, the input language of OpenDF. Finally, Section VII concludes with some results and some preliminary conclusions.

II. SYNCHRONOUS GUARDED ACTIONS

Synchronous languages [5, 16] such as Esterel [6], Lustre [16] or Quartz [25] all follow the same general scheme: in each reaction, the system reads all inputs to update its internal state and to compute all outputs. Each of these reactions is usually called a macro step [17], while all the computations done in a macro step are commonly called micro steps. The synchronous model of computation postulates that each variable of the system has a unique value in each macro step, so that all micro steps of a macro step are based on the same mapping of values to variables, i.e. they appear to be executed at the very same time. In practice, this means that the micro steps must be executed according to the data dependencies. Therefore, synchronous systems are always deterministic, since a variable can not have two different values in a single macro step.

In the following, we use a very general representation of synchronous systems, namely synchronous guarded actions. Since all synchronous languages can be translated to guarded actions [9], guarded actions are the perfect starting point for our translation to DPNs. Thereby, we do not depend on any concrete synchronous language.

Synchronous guarded actions are in the spirit of classical guarded commands [12], which are widely used to describe the behavior of concurrent systems [10, 13, 20]. However, in our case, they follow the synchronous model of computation, i.e. their semantics is different compared to the classical counterpart. In general, a guarded action has the form \( \langle \gamma \Rightarrow A \rangle \), where the guard \( \gamma \) is a Boolean condition that determines wether the action \( A \) is activated in the current macro step or not. The action \( A \) corresponds to a micro step as explained above, and in our case this is either an immediate assignment of the form \( x = \tau \) or a delayed assignments of the form \( \text{next}(x) = \tau \). While both types evaluate the right-hand side \( \tau \) in the current macro step, immediate assignments cause \( x \) to have the value of \( \tau \) in the current macro step, while delayed assignments transfer the value in the following macro step. These types of assignments are comparable to wires (immediate assignments) and registers (delayed assignments) in synchronous hardware circuits.

In general, the value of a variable \( x \) depends on a set of guarded actions having \( x \) on the left-hand side of the action. Even if none of these actions is activated in the current step, i.e. if the guards of all of these actions are evaluated to false, \( x \) is given a well-defined value: Depending of the declaration of \( x \), it either keeps its old value (memorized variable) or it is reset to a default value (event variable). However, in the following, we assume that for each variable in each macro step exactly one action will be executed. This is no restriction, since this default action can be also encoded as a guarded action.

Furthermore, we assume that the given set of synchronous guarded actions is consistent. In particular, they are free of write conflicts (several actions assign different values to the same variable in the same macro step) and causality problems [7, 27] (several actions mutually depend on each other). These problems are well studied in the context of synchronous systems, and many analysis procedures have been developed to locate and eliminate these problems. Hence, we can assume that our system does not have these problems.

\[
\begin{align*}
\text{true} & \Rightarrow \text{next}(x) = a & \text{true} & \Rightarrow c = z \cdot z \\
\text{true} & \Rightarrow \text{next}(r) = s & \text{true} & \Rightarrow x = p \\
\text{true} & \Rightarrow \text{next}(o) = a \cdot b & \text{true} & \Rightarrow y = q \\
\text{true} & \Rightarrow a = x + y & \text{true} & \Rightarrow y = o \\
\text{true} & \Rightarrow b = x - y & \text{true} & \Rightarrow m = b + c
\end{align*}
\]

Figure 1. Synchronous Guarded Actions

Figure 1 list an example for synchronous guarded actions, which does not represent a meaningful algorithm, but which will serve as a running example in the rest of this paper. The set is defined over the inputs \( s, p, q \) and \( z \) (which can be only read), the output \( o \) (which is exposed at the system interface), and some other local variables. It contains three delayed actions writing \( x, r \) and \( o \), and seven immediate actions.

III. DATA-FLOW PROCESS NETWORK

Data-flow process networks have a long history, and their roots date back to the 70s. The most known variant are certainly Kahn networks introduced in [21], while many other variants exist [4, 23, 28]. All models have in common that they consist of a set of nodes which process data concurrently and independently from each other. These nodes communicate by sending tokens through unbounded point-to-point FIFO buffers.

As DPNs are a completely untimed model, the firing rules are triggered and not bounded to a specific point of time, i.e. there is no global clock that fires activated firing rules. Hence, the scheduling of all nodes is determined by the data-flow of the network. In particular, DPN nodes run independently from each other and the behavior of each node is controlled by firing rules. Each DPN node consists of a list of firing rules. Each firing rule consists of a guard that defines when the node has to be activated and at least one guarded action. A general behavior for DPN nodes is not defined in detail. Hence, we set its format as follows: the guarded actions are carried over including their semantics such that each single DPN node keeps the synchronous MoC. Changing the parts of the synchronous system does not improve the following
Translation steps, but keeping them as long as possible in their original MoC avoids non-reversible conversions. Definition 1 gives a formal definition of DPN nodes.

**Definition 1 (DPN Node):** A node in a DPN is defined as follows:

$$\text{DPNNode} := \text{list of FiringRule}$$

$$\text{FiringRule} := \text{list of } (\gamma_B \Rightarrow \text{Guarded Action})$$

In the following, we use firing rules to describe the behavior of a single node. In general, a single node may have several firing rules, where a single firing rule consists of (1) a condition stating when it has to be activated, (2) a list of input buffers that have to be read, (3) a function that is applied to the input tokens, and finally (4) the output buffers where the produced tokens are put.

It is possible that firing rules of a node overlap, i.e. more than one is enabled at some point of time. In this case, any enabled firing rule can be chosen, which obviously gives rise to a nondeterministic behavior of the whole network. For instance, consider the node given on the left-hand side of Figure 2: if both input buffers are filled, both firing rules are activated. Furthermore, according to the Kahn principle [21], nondeterministic behavior of the network is even possible if rules do not overlap. The example on the right-hand side of Figure 2 illustrates this effect: the ordering of the values sent through $\alpha$ depends on the arrival time of inputs on $i_0$ and $i_1$.

**IV. Translating Synchronous Systems to DPNs**

The starting point of our translation is a synchronous system represented by synchronous guarded actions as described in Section II. In the following, we assume that we have a set of guarded actions $G$, which are defined over a set of variables $V$. The goal of our translation is the creation of a DPN, which will work as follows: each node of the DPN will be activated if all input buffers contain at least one token. If an activated DPN fires, it reads from all input buffers one token, processes them and places exactly one token in every output buffer.

Our translation is divided into several steps, which are described in the following. Section IV-A describes some initial preprocessing steps on the guarded actions, before Section IV-B shows how DPN nodes are created for the guarded actions. Section IV-C focuses on the dependencies between these nodes, and Section IV-D discusses the creation of firing rules. The final step is the partitioning of the resulting DPN, which we describe in Section IV-E.

**A. Preprocessing the Synchronous System**

As already stated above, we start with a synchronous system represented by a set of guarded actions $G$ defined over a set of variables $V$. In general, several guarded actions write a variable $x \in V$, and this set of actions may contain immediate and delayed actions. Since this would complicate the translation presented in the following (because write accesses address different macro steps), we first normalize the set of guarded actions so that each variable is either written by immediate or delayed actions.

This can be accomplished by introducing an auxiliary variable $x'$ (sometimes called carrier variable) for each $x$ which is written by immediate and delayed actions. While the immediate actions still write the original variable $x$, all delayed actions are redirected to the new variable $x'$.

As we need to know whether a delayed action has written some value to $x'$, we additionally need to track the guards in $x'_{\text{guard}}$ of all delayed actions writing $x'$ so that we know when to transfer the value of $x'$ to $x$. Figure 3 formalizes these considerations.

As explained at the beginning of this section, a DPN node will always read a token from each incoming buffer whether or not the value is required for the calculations. Hence, we have to ensure that for every variable in every step a value has to be set by a guarded action. This must be also done for the auxiliary variables, which is accomplished by Lines 19 and 20 in function SeparateDanIA.

If we apply the function SeparateDanIA to our running example given in Figure 1, we get the guarded actions shown in Figure 4. Obviously, only variable $x$ is written by immediate and delayed actions. Hence, only for that variable auxiliary variables $x'$ and $x'_{\text{guard}}$ are inserted. The delayed action which writes $x$ is redirected to $x'$, and we set $x'_{\text{guard}}$ to true if and only if the delayed write takes place. Finally, an action is added to copy the result of a delayed write if it has taken place. Note that the system behavior has not changed, and the modified system cannot be distinguished from the original one by the environment.
B. Creating DPN Nodes

The next step of our translation will transform the synchronous system to a DPN. Thereby we basically map the guarded actions \( G \) to nodes of the DPN, and the variables \( V \) to the FIFO buffers in between. In the following, we first consider the nodes, before the next subsection describes the creation of the buffers in between.

As the buffers of a DPN are point-to-point connections, they all have a single writer and a single reader for each variable. However, even after the preprocessing of the previous section, each variable still may be written by several actions.

To handle several readers of a variable, the buffer representing a variable \( x \in V \) needs to be split up in buffer receiving new tokens for \( x \), several other buffers providing the new values and a duplicator node in between. In the following, for the sake of simplicity, we assume that the DPN has implicit duplicators, i.e. we allow several nodes to read the same variable.

In contrast, as several writers of a variable cannot be handled by the introduction of a naive merge node, we have to group the guarded actions before the transformation to a DPN node so that each buffer is written by a single node. Hence, the crucial step is to group all actions that write to the same variable (see Figure 5). To this end, we first create a DPN node with a single empty firing rule (thus containing no actions) for each writable variable and subsequently add all actions that write to a variable to the firing rule of the corresponding DPN node. The guard \( \gamma \) for this firing rule is preliminarily set to true but it will be improved in the rest of the translation. Recall that the previous separation of immediate and delayed actions to the same variable will lead to a creation of two separate nodes for a single program variable \( x \).

C. Creating DPN Buffers

After the nodes of the DPN have been created, we have to set up the communication infrastructure by adding FIFO buffers to the network. These buffers basically need to be placed at two positions: (1) between nodes exchanging data and (2) at the system interface to pass the inputs and outputs of the system.

In our representation of DPNs we describe the sets of buffers by their characteristic functions. To this end, we define two functions \( \text{hasImEdge}(N_1, N_2) \) and \( \text{hasDeEdge}(N_1, N_2) \), which return true if the DPN contains a buffer, which obtains data from node \( N_0 \) and makes it available to node \( N_1 \). Similarly, we define a function \( \text{imOutbounds}(N) \) to describe the buffers read by a node and two functions \( \text{imOutbounds}(N) \) and \( \text{deOutbounds}(N) \) to describe the buffers written by a node. These definitions will later help us to define the buffers at the system interface.

For the internal and the output buffers, we distinguish buffers that are due to immediate and delayed actions. While both variants indicate that a buffer will be placed between the nodes, we have to distinguish them since they need to be initialized differently: the delayed variant must initially contain a token representing the default value for that variable so that its tokens are assigned to the right step.

In order to find the places where internal buffers must be placed, we have to analyze the data dependencies between the guarded actions. Basically, whenever a variable \( x \) is written in node \( N_1 \) and read by another node \( N_2 \), an edge has to be created from \( N_1 \) to \( N_2 \). Thus, we first define read and write dependencies of guarded actions. Thereby, we distinguish between variables that are written by immediate and delayed actions.

**Definition 2 (Action Dependencies):** Let \( FV(\tau) \) denote the free variables occurring in the expression \( \tau \). Then, the dependencies from actions to variables are defined as follows:

- \( \text{rdVars}(\gamma \Rightarrow \gamma) := FV(\gamma) \cup FV(\gamma) \)
- \( \text{rdVars}(\gamma \Rightarrow \text{next}(x) = \gamma) := FV(\tau) \cup FV(\gamma) \)
- \( \text{wrImVars}(\gamma \Rightarrow \gamma) := \{x\} \)
- \( \text{wrImVars}(\gamma \Rightarrow \text{next}(x) = \gamma) := \{x\} \)
- \( \text{wrDeVars}(\gamma \Rightarrow \gamma) := \{x\} \)
- \( \text{wrDeVars}(\gamma \Rightarrow \text{next}(x) = \gamma) := \{x\} \)

We now lift Definition 2 to DPN nodes, i.e. we define the variables that are read or written by the firing rules of a node \( N \). Similar to the guarded actions, we distinguish immediate and delayed write accesses so that we define two functions \( \text{wrImVarsDPN}(N) \) and \( \text{wrVarsDPN}(N) \).

**Definition 3 (Node Dependencies):** Let \( \mathcal{G}(N) \subseteq \mathcal{G} \) be all guarded actions of a single DPN node \( N \). Then, the dependencies from the DPN node \( N \) to variables are defined as follows:

- \( \text{rdVarsDPN}(N) := \bigcup_{\gamma \in \mathcal{G}(N)} \text{rdVars}(\gamma) \)
- \( \text{wrImVarsDPN}(N) := \bigcup_{\gamma \in \mathcal{G}(N)} \text{wrImVars}(\gamma) \)
- \( \text{wrDeVarsDPN}(N) := \bigcup_{\gamma \in \mathcal{G}(N)} \text{wrDeVars}(\gamma) \)
- \( \text{wrVarsDPN}(N) := \bigcup_{\gamma \in \mathcal{G}(N)} \text{wrImVars}(\gamma) \cup \text{wrDeVars}(\gamma) \)

Now we are in the position to define the functions mentioned above (see Figure 6), which are almost straightforward with the previous definitions. However, the clause \( N_0 \neq N_1 \) in the first line may not be obvious: To understand this restriction, consider a DPN node which reads \( x \) and also contains a delayed action writing to the same variable \( x \). It requires an intermediate variable to store the result of a delayed write, such that the value of the current macro step is not overwritten and

\[
\begin{align*}
s & \Rightarrow \text{next}(x') = a & \text{true} & \Rightarrow a = x + y \\
\neg s & \Rightarrow \text{next}(x') = 0 & \text{true} & \Rightarrow b = x - y \\
s & \Rightarrow \text{next}(x'_{\text{guard}}) = \text{true} & \text{true} & \Rightarrow c = x \cdot z \\
\neg s & \Rightarrow \text{next}(x'_{\text{guard}}) = \text{false} & \text{true} & \Rightarrow p = m = b + c \\
x'_{\text{guard}} & \Rightarrow x = x' & \text{false} & \Rightarrow y = q \\
\text{true} & \Rightarrow \text{next}(r) = s & \neg s & \Rightarrow y = o \\
\text{true} & \Rightarrow \text{next}(o) = a \cdot b & \text{true} & \Rightarrow m = b + c 
\end{align*}
\]

Figure 4. Example: Synchronous Guarded Actions after Preprocessing

\[
\text{function } \text{CreateDPNNodes}(V, G) \\
\text{\quad } N := \{ \} \\
\text{\quad } \text{for all } x \in V \\
\text{\quad } \text{if } (\text{wrImActs}(x) \neq \{\}) \text{ then} \\
\text{\quad } \quad N := N \cup \text{new DPNNode}(N) \\
\text{\quad } \text{else } (\text{wrDeActs}(x) \neq \{\}) \text{ then} \\
\text{\quad } \quad N := N \cup \text{new DPNNode}(N) \\
\text{\quad } \text{return } N
\]

Figure 5. Creating DPN Nodes
DetermineGuardOfFR

\[ \forall v \in \text{present}(N) \quad \gamma_B := \bigwedge_{v \in \text{rdVarsDPN}(N)} \text{present}(v) \quad N := (\gamma_B \Rightarrow \mathcal{G}(N)) \]

Figure 6. Creating DPN Buffers

\[ \text{hasImEdge}(N_0, N_1) := N_0 \neq N_1 \land (\text{wrImVarsDPN}(N_0) \cap \text{rdVarsDPN}(N_1)) \neq \emptyset \]

\[ \text{hasDeEdge}(N_0, N_1) := (\text{wrDeVarsDPN}(N_0) \cap \text{rdVarsDPN}(N_1)) \neq \emptyset \]

\[ \text{inbounds}(N) := \{N_S \in \mathcal{N'} : \text{hasDeEdge}(N_S, N)\} \]

\[ \text{hasOutbounds}(N, N_T) := \{N_T \in \mathcal{N'} : \text{hasImEdge}(N, N_T)\} \]

\[ \text{deOutbounds}(N) := \{N_T \in \mathcal{N'} : \text{hasDeEdge}(N, N_T)\} \]

Figure 8. Function to group actions writing to the same variable to a single DPN node.

With the definitions of the nodes in the previous section and the buffers in this section, we can now give the DPN of our running example. Figure 7 shows the resulting network.

D. Determining the Guards of Firing Rules

After the creation of the DPN and determining the dependencies between the nodes, we can now determine the guards of the firing rules. As mentioned at the beginning of this section, each DPN node has exactly one firing rule that has to be activated as soon as all input buffers contain at least one token. In our implementation, we use the flag present(v) to check whether v contains a token or not. Figure 8 shows the code for assigning the guards to the firing rules. Note that a guard of a firing rule is not restricted to a conjunction of these flags, but can be any Boolean expression.

E. Post-Processing the DPN

At that point, we already have a DPN for a given synchronous system. Hence, we can use the created model to feed it into other tools, e.g. we could generate CAL (see Section V) and synthesize C or HDL code. However, we could optimize the DPN as generated by the previous sections in a final step of our translation. Obviously, this post-processing does not change its behavior.

A post-processing of the DPN makes sense if subsequent synthesis steps require coarse-grained parallelism, i.e. nodes with a larger number of actions. As an example, we recall that creating multithreaded code requires each thread to contain a minimum amount of computational effort such that the synchronization between threads is amortized. A higher computational effort can be reached by putting more actions into a thread. Another motivation comes from the fact that sometimes the number of available resources is known in advance so that a static clustering of nodes reduces the runtime overhead.

Due to this reason, it is interesting to partition a given DPN and to merge nodes of each partition to a single DPN node. In the following, we will not discuss strategies how to gain a good partitioning since the choice of the best algorithm heavily depends on the final architecture and its requirements. This is out of the scope of this paper, and there is a large number of papers which exactly addresses this problems, e.g. to map a DPN to a MPI cluster, one could use the Ford-Fulkerson’s algorithm [11] to get the min-cut/max-flow of the DPN to minimize communication between the resulting nodes.

In the following, we define what a valid partitioning is, which gives rise to the correctness of a network post-processing.

1) Each partition consists of a set of DPN nodes that can be merged to a single DPN node. This condition guarantees that the result of the partitioning step is a new DPN. In particular, merging of DPN nodes \( N_1, N_2, \ldots, N_k \) in this context is done in four steps: 1) create a new DPN node \( N \) and add it to the DPN; 2) \( \gamma_B(N) = \bigwedge_{i=1,\ldots,k} \gamma_B(N_i) \); 3) \( \mathcal{G}(N) := \bigcup_{i=1,\ldots,k} \mathcal{G}(N_i) \); 4) remove \( N_1, \ldots, N_k \) from the DPN and delete them. Note that due to the implicit definition of edges by the DPN nodes, there is no need for redirecting or creating edges.

2) A partition creates a set of groups and a bijective mapping of the created groups to DPN nodes. In particular, the sets of DPN nodes represented by the groups must be disjunctive. Furthermore, DPN nodes must not be duplicated or removed.

3) After merging all DPN nodes of each group to a single DPN node, the resulting graph must not contain loops that contain immediate edges. Loops containing at least one delayed edge are allowed because delayed write accesses refer to the next macro step, and therefore, these loops do not lead to a deadlock of the DPN.

These conditions are formally defined as the term legal partition in Definition 4.

Definition 4 (Legal Partition of a DPN): A partition \( \pi \) of a DPN is a mapping from DPN nodes to groups \( \pi \in \Pi \). Let \( \text{partition}(A) \) denote the group of a DPN node \( N \in \mathcal{N} \), and let \( \text{nodes}(\pi) \) denote all the DPN nodes occurring in group \( \pi \). A partition is legal iff all of the following points are fulfilled:

1) \( \forall \pi_0, \pi_1 \in \Pi \pi_0 \neq \pi_1 \rightarrow \text{nodes}(\pi_0) \cap \text{nodes}(\pi_1) = \emptyset \)

2) Let \( \sqsubseteq \) be the reflexive and transitive closure of the following relation \( \mathcal{R} \subseteq \mathcal{N} \times \mathcal{N} : (N_0, N_1) \in \mathcal{R} \iff \text{wrImVars}(N_0) \cap \text{rdVars}(N_1) \neq \emptyset \), then \( \sqsubseteq \) is a partial order.
V. TRANSLATING DPNs TO CAL

Section IV explained how synchronous systems can be translated to DPNs. In this section, we illustrate with the help of an example how the resulting networks can be finally implemented. As an example target language, we choose CAL, the input language for the OpenDF package\(^1\). The OpenDF tools can translate data-flow graphs to C and VHDL. Furthermore, the OpenDF toolset contains a simulator with arbitrary precision arithmetic for investigating the behavior of a given DPN or even a single node. In the following, we will start with a brief introduction to CAL and explain the creation of CAL code by the running example.

A CAL application, usually called CAL network, consists of a set of actors and a net-list. An actor corresponds to a node in the DPN. The net-list instantiates these actors and connects their input and output queues with each other and the environment, i.e. it corresponds to the DPN buffers.

An example for an actor is given in Figure 9. The CAL code represents a DPN node containing the guarded actions occurring in the guard are known, the guard is evaluated. If it is not satisfied, a roll-back of the execution including the consumption of the tokens is performed. Figure 10 shows a semantically equivalent actor using guards instead of if-statements. Note that all actions of Actor\(Y\)' still read all input queues, although the actions only use some part of it. This is necessary since the producers will always create a token when an action for their actor is executed.

After the translation of all DPN nodes to CAL, the actors have to be connected according to the dependencies in the DPN. As already stated above, in CAL this is accomplished by a net-list description. Figure 11 shows the net-list for our running example. The example shows, the net-list consists of a header, the entities and the structure. The header of the net-list is similar to the one of the actor: it defines the net-list’s name and the interface to the environment. In the next step, the entities section instantiates all nodes of the data-flow graph. This part is straightforward by generating a line of code for each node in the DPN, as shown in Figure 11. Finally, the structure section completes the net-list description: it describes how to connect the actors with each other and the environment.

The function ConnectActors in Figure 12 creates for a given list of nodes the code for the structure section. Thereby, function PrintConnection(\(A,x,B,y\)) is meant to be a printing function that prints CAL code to connect the output \(x\) of actor \(A\) with input \(y\) of actor \(B\). If one of the actors is omitted, the corresponding variable is an input (\(x\)) or output (\(y\)) of the net-list, i.e. an input or an output of the system.

VI. EXPERIMENTAL RESULTS

We implemented the translation from synchronous system to data-flow process networks and the subsequent step to

---

\(1\)http://www.opendf.org
function ConnectActors(N)
for all NT \in N
for all NS \in inbounds(N_T)
  V := rdVarsDPN(N_T) \cap wrVarsDPN(NS)
for all v \in V
  PrintConnection(NS.v, N_T.v)
for all v \in (rdVarsDPN(N) \cap inputs)
  PrintConnection(v, N_T.v)
for all v \in (wrDPNNode(N) \cap outputs)
  PrintConnection(N_T.v, v)

Figure 12. Function to connect actors in a net-list.

actor ActorY'() bool S, int Q, int O => int Y:
  action S : [s], Q : [q], O : [o] => Y : [y]
  guard s do
    y := q;
  end
  action S : [s], Q : [q], O : [o] => Y : [y]
  guard not s do
    y := o;
  end

Figure 10. Actor of Figure 9 using guards

net-list RunningExample()

for all M, int P, int Q, int Z, bool S => P : M
  entities
    inst_ActorA = ActorA();
    ...
    inst_ActorY = ActorY();
  structure
    inst_ActorA.X_in = inst_ActorX.X_out;
    ...
    inst_ActorY.X_in = inst_ActorY.X_out;

Figure 11. Example: CAL net-list

CAL within our Averest system as described in the previous sections. As a starting point, we use synchronous programs written in Quartz, which are translated by a compiler to AIF (Averest Intermediate Format) files, which based on synchronous guarded actions.

Figure 13 shows experimental results that we obtained from applying the translation to a subset of synchronous programs from our Averest benchmark set. Thereby, the examples IntEqu, IntLes, Int2SD, NatEqu, NatSub, SD_EQU, SD_ADD, and SD_SUB all implement arithmetic functions for arbitrary precision integers. MatrixMultComb2 is a combinational implementation of a 6x6 matrix multiplication and WarshallA is an implementation of Warshall’s algorithm for computing the transitive closure of a graph. RSFLipFlop implements a flip flop constructed with NOR-gates. Furthermore, we have an implementation for calculating the edit distance in EditDistance. ConvArray0/N computes the discrete convolution of an input stream with respect to given weights w[0..5] for N = 1, 2, 3 and w[0..6] for N = 4. While SortCell is a sorter for exactly two variables, SortingNetworks checks whether a given permutation matrix P represents a correct sorting network.

The second group of columns in the table show some properties of the translated DPN, including the number of nodes and edges. The more nodes a DPN has, the better the network can be distributed. The number of edges correlates to the amount of communication that is required for running a system. Additionally, we list the length of the longest path and the maximal width of the graph. These numbers are a measure for the parallelism of the graph and thereby the theoretical workload supported by the DPN. In general, the latency of a system, i.e., the delay between sending inputs and receiving outputs of the system, will be proportional to the length of the longest path. If the length of the longest path l_{max} is short compared to the number of nodes in the DPN, the number of nodes that can be executed in parallel for a single input set will be higher. The maximum number of nodes that can be executed is already given by the theoretical maximum width w_{max} and will usually only be a peak value. Nevertheless, the closer the product l_{max} \cdot w_{max} is to the number of nodes in the DPN, the more homogeneous and better the utilization will be.

The last set of columns shows experimental data obtained from the translation to CAL, namely the simulation time of the CAL simulator and the number of steps that have been necessary to simulate 100 input sets with random values.

The experimental results show the feasibility of our approach. The parallelism of the original synchronous programs is still available in the DPNs, while internal synchronization could be reduced to a minimum.

VII. SUMMARY

Synchronous languages are a good choice for applications requiring precise timing of actions and architectures having a local control to enable these actions. These properties make a compilation to target architectures without local control like multicore processors or workstation clusters challenging.

We presented a translation to bridge the gap between these two worlds. It compiles synchronous systems to data-flow process networks (DPN), thereby translating the synchronous model of computation to an asynchronous one. We illustrated the feasibility of our approach by a translation of DPNs to CAL, and experimental results are very promising that the resulting DPNs can be used for an efficient synthesis of these architectures.

REFERENCES


<table>
<thead>
<tr>
<th>Example</th>
<th>data-flow process network</th>
<th>OpenDF simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int2SD</td>
<td>#nodes: 13, #edges: 23, (l_{\text{max}}: 2), (w_{\text{max}}: 7)</td>
<td>#steps: 3045, time: 1313</td>
</tr>
<tr>
<td>IntEqu</td>
<td>#nodes: 2, #edges: 1, (l_{\text{max}}: 1), (w_{\text{max}}: 2)</td>
<td>#steps: 609, time: 2490</td>
</tr>
<tr>
<td>IntLes</td>
<td>#nodes: 13, #edges: 26, (l_{\text{max}}: 6), (w_{\text{max}}: 3)</td>
<td>#steps: 2842, time: 2241</td>
</tr>
<tr>
<td>NatEqu</td>
<td>#nodes: 2, #edges: 1, (l_{\text{max}}: 1), (w_{\text{max}}: 2)</td>
<td>#steps: 609, time: 2484</td>
</tr>
<tr>
<td>NatSub</td>
<td>#nodes: 18, #edges: 33, (l_{\text{max}}: 12), (w_{\text{max}}: 7)</td>
<td>#steps: 3251, time: 2136</td>
</tr>
<tr>
<td>SD_ADD</td>
<td>#nodes: 23, #edges: 43, (l_{\text{max}}: 2), (w_{\text{max}}: 12)</td>
<td>#steps: 5074, time: 3323</td>
</tr>
<tr>
<td>SD_EQU</td>
<td>#nodes: 25, #edges: 57, (l_{\text{max}}: 3), (w_{\text{max}}: 3)</td>
<td>#steps: 5480, time: 3299</td>
</tr>
<tr>
<td>SD_SUB</td>
<td>#nodes: 13, #edges: 23, (l_{\text{max}}: 2), (w_{\text{max}}: 7)</td>
<td>#steps: 3044, time: 2313</td>
</tr>
<tr>
<td>MatrixMultComb2</td>
<td>#nodes: 41, #edges: 44, (l_{\text{max}}: 2), (w_{\text{max}}: 39)</td>
<td>#steps: 1185, time: 8605</td>
</tr>
<tr>
<td>WarshallA</td>
<td>#nodes: 29, #edges: 70, (l_{\text{max}}: 3), (w_{\text{max}}: 11)</td>
<td>#steps: 6393, time: 1957</td>
</tr>
<tr>
<td>RSFlipFlop</td>
<td>#nodes: 4, #edges: 7, (l_{\text{max}}: 1), (w_{\text{max}}: 4)</td>
<td>#steps: 1812, time: 1034</td>
</tr>
<tr>
<td>EditDistance</td>
<td>#nodes: 174, #edges: 466, (l_{\text{max}}: 22), (w_{\text{max}}: 35)</td>
<td>#steps: 46337, time: 977</td>
</tr>
<tr>
<td>ConvArray01</td>
<td>#nodes: 13, #edges: 37, (l_{\text{max}}: 1), (w_{\text{max}}: 13)</td>
<td>#steps: 5953, time: 943</td>
</tr>
<tr>
<td>ConvArray02</td>
<td>#nodes: 18, #edges: 56, (l_{\text{max}}: 2), (w_{\text{max}}: 13)</td>
<td>#steps: 9943, time: 925</td>
</tr>
<tr>
<td>ConvArray03</td>
<td>#nodes: 18, #edges: 56, (l_{\text{max}}: 1), (w_{\text{max}}: 18)</td>
<td>#steps: 10604, time: 1027</td>
</tr>
<tr>
<td>ConvArray04</td>
<td>#nodes: 27, #edges: 85, (l_{\text{max}}: 1), (w_{\text{max}}: 27)</td>
<td>#steps: 17498, time: 933</td>
</tr>
<tr>
<td>SortCell</td>
<td>#nodes: 7, #edges: 13, (l_{\text{max}}: 3), (w_{\text{max}}: 4)</td>
<td>#steps: 2018, time: 1089</td>
</tr>
<tr>
<td>SortingNetworks</td>
<td>#nodes: 179, #edges: 694, (l_{\text{max}}: 8), (w_{\text{max}}: 3)</td>
<td>#steps: 30110, time: 2188</td>
</tr>
</tbody>
</table>

**Figure 13.** Experimental results


