From High Level Application Specification to System-level Architecture Definition: Exploration, Design and Compilation

Vladimir D. Živković
Ed Deprettere
LIACS - Leiden University
The Netherlands
{lale,edd}@liacs.nl

Pieter van der Wolf
Erwin de Kock
Philips Research Laboratories
The Netherlands
{pieter.van.der.wolf,erwin.de.kock}@philips.com

Abstract

In this paper we present a survey of a recently developed approach to specifying, modelling, exploring and designing parallel media-processing and wireless communications platforms and platform instances. More specifically, we take a closer look at the notions of Y-chart and Abstraction Pyramid. The methodology and approach that is reviewed in this paper uses these notions and focuses on the system level of abstraction on which the application and the architecture are co-existing in terms of a Model of Computation and a Model of Architecture, respectively. The relation between the two models is expressed in terms of a mapping layer with transformation methods that take a particular representation of (parts of) the application model and bind it to a particular representation of (parts of) the architecture model, and all this in such a way that: 1) modifications in either the application, or the architecture, or the mapping can be easily dealt with (for exploration purposes), 2) the evaluation of the models yields sufficient accurate data to help steer design decisions, and 3) the subsequent design of one or more physical instances of a platform can be done along compilation ways that are not too distant from the transformations that were explored to arrive at feasible platform instances. We argue that the so-called Trace Driven co-simulation using linearly ordered Symbolic Instruction representations of the application model has proved to be conceptually successful but practically limited by nature. We then show how an extension of this approach to the so-called Symbolic Program co-simulation using Symbolic Program representations of the application model is a better and more powerful approach satisfying the criteria given above.

1. Introduction

Today's embedded systems are to be designed for more complex and more compute-intensive applications than they were a decade ago. This is due to advances in the semiconductor industry and the development of advanced applications. The large amount of processing resources that can be integrated in a single chip has challenged system-level design approaches. Emerging design-paradigms and a long-term accumulated knowledge and expertise of the parallel-computing community may help to accomplish the design tasks. At the same time, design paradigms that are evolving from and built on top of Object-Oriented Technologies, are maturing to support system-level exploration and design methodologies. And, likewise important, modelling applications and architectures, and mapping concurrent tasks into parallel architectures are boosted by sound and rich methods that have been researched and proposed by the parallel-computing community over the past two decades. Thus, parallel computing is finding its way out from the scientific computing application domain into other application domains, among which are the multimedia and (wireless) communications domains.

In these domains, the parallel computing system is currently called the platform which is a heterogeneous system consisting of communicating processing units of different types and mostly distributed memory units. A platform is more dedicated than - say - a cluster computer, yet it can not be so dedicated that it can only perform a single task well.

However, even though the potentials for designing complex and sophisticated embedded systems platforms are large, a satisfactory methodology has not yet emerged that can meet market constraints. That is, the specification, exploration and design of application domain-specific system platforms from user requirements is still a painstaking process that takes too long and is too costly.

For example, SystemC [12] provides primitives to build models at higher levels of abstraction, yet it does not offer a methodology for system-level design in which models can be refined systematically.

As another example, although SpecC [13] offers more of a methodology it does not tend to support complex multi-processor platforms (examples shown in [13] suggest that the target platform is relatively simple).

As a last example, the Compaan System [19] offers a sound methodology and tools to parallelise imperative code and to map the parallel models into some hardware platforms, yet is restricted to static nested-loop algorithms which constitute only a portion of the algorithms encountered in media and communications applications.

We embody in this paper an application domain-specific platform design methodology that we are developing and experimenting with. Our methodology is underpinned by the notions of *Abstraction Pyramid* and *Y-chart Approach* [1] [2]. The abstraction pyramid visualises at a glance the reachability and operability of a design space from a particular level of abstraction. The Y-chart approach is, mutatis mutandis, applicable at each and every level of abstraction and advocates the application - and separation - of Models of Application, Models of Architecture, and Methods of Mapping applications into architectures. The models are assumed to match for the mapping methods to be feasible: binding representations of (parts of) application models with representations of (parts of) architecture models through representation transformations, all this in such a way that 1) modifications in either the applications, or the architecture, or the mapping can be easily dealt with - for exploration purposes, 2) the evaluation of the models yields sufficient accurate data to help steer design decisions, and 3) the subsequent design of platform instances can be undertaken along compilation paths that are not too distant from the transformations that were applied in the exploration phase to arrive at feasible platform instances.

The rest of the paper is organized as follows. In Section 2 we briefly review the notions of *Abstraction Pyramid* and *Y-chart Approach*. The Abstraction Pyramid represents a top down view of stepwise exploration and refinements step on intra-levels of abstractions and inter-levels of abstraction, respectively, from a highest specification level, down to a lowest synthesis level. Next, in Section 3 we review the models we use to model applications and architectures. These are heterogeneous Models of Computation (MoC), called Models of Applications and Models of Architecture, respectively. Then, in Section 4 we present and contrast three different representations of application process components and architecture processor components in these models as well as the corresponding mapping methods that transform process representations into processor representations. Subsection 4.1 deals with the classical Control Data Flow Model and Compilation methods which are typically used by designers. Subsection 4.2 goes into an exploration oriented mapping approach in which (parts of) the application model are represented by *Traces of Symbolic Instructions* that are mapped onto components in the architecture model, after having gone through a number of symbolic instruction transformations. The method is simple and fast but, at the same time, limited and not satisfactory from the point of view that exploration and design should be inter-related. Finally, we propose in Subsection 4.3 a more generic approach in which we replace the symbolic instructions representation with a symbolic program representation. This *Symbolic Program* approach is richer than the symbolic instruction approach and, at the same time, reminiscent of the Control Data Flow Graph model that is typically a designer's model. This approach brings explorers and designers in a more unified framework. The Symbolic Program approach has emerged as our most recent and most preferred approach. For this reason, we elaborate on this approach in Section 5.

2. Abstraction Pyramid and Y-chart

Most of the embedded systems, however large, are designed by large teams under the supervision of an expert chief architect, even today. However, the commercial tools to do so have reached their problem-complexity limits beyond which they are no longer reliable. Moreover, recurrent engineering costs are skyrocketing when it comes to designing complex embedded systems for reactive compute-intensive applications such as multimedia and wireless communications. Sooner or later, these hindrances must lead to a dead end unless a paradigm shift paves a new way. Such a paradigm shift is glooming already as witnessed by the emerging of new terms such as *platform*, *platform instance*, *system-level*, *re-use*, and *intellectual property*. As far as we know, for the term *platform* there is no definition that is generally agreed upon. Roughly speaking, then, a platform is a collection of feasible constellations of interconnected components. The components as well as the platform are typical for a particular application domain. Alternatively, a platform is a parametrised architecture template, in which the parameters may range from *type of interconnection structure* to *number of components of a certain type* to *type and size of storage unit*. A platform instance is a platform together with a vector of values of the parameters. A platform is said to be specified at system-level when the components are, themselves, specified in terms of parameters, such as *latency* and *throughput*. Library components do have, of course, physical counterparts that are typically designed as re-usable intellectual properties (IP). The current trend is that platform designer and IP designer are no longer one and the same person (or team). In fact, platform designers are working on *higher levels of abstraction* whilst IP designers are working on *lower levels of abstraction*. To make this distinction and its consequences explicit, a so-called *Abstraction Pyramid* was introduced in [6], and reproduced here in Figure 1 for convenience.

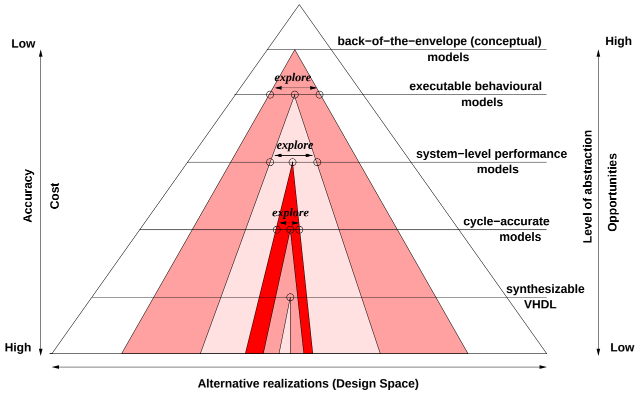

Figure 1. The abstraction pyramid

The base of the pyramid represents the complete design space for the application domain. This space is, at least in principle, reachable from user requirements and constraints that are represented by the top of the pyramid. Specification, exploration and design is then proceeding at discrete levels of abstraction as represented by the parallel cuts. At each and every level, level-specific models are used to exploring platform instances with levels of confidence that are within pre-defined bounds. Selected instances narrow down the reachable design space as is illustrated by the inner pyramids in Figure 1. Transition from one level of abstraction to the next one down comprises a number of refinements of both the parameters and the accuracy measures. The cost of model construction and evaluation is higher at the more detailed levels of abstraction whilst the opportunities to explore alternatives is significant lower at the higher levels of abstraction. Exploration and design at higher levels of abstraction is called system-level exploration and design. At system-level, parameters and parallelism are typically coarse grained, and performance/cost measures are simple metrics. For example, a processor unit is characterised by a latency and a throughput, parallelism is at the level of tasks, and performance and cost are measured in terms of, say, throughput and number of processing units, respectively.

The models that are used on a particular level of abstraction are crucial. The models we rely on are formal models on which we can reason. Thus, we specify both the applications and the architecture for the execution of the applications in terms of so-called *Models of Computation* (MoC) [15]. We call them *Models of Application* and *Models of Architecture*. A MoC essentially deals with communication and synchronisation of computations that take place in actors or processes. In our approach, both models are strictly separated and methods to mapping platform independent behaviour (application models) into platform depending behaviour (architecture models) are an essential part in the process of exploration and design. This approach was proposed in [3] and called the *Y-chart approach*. The name stems from the pictorial view of the approach as shown in Figure 2.

The Y-chart approach permits multiple target applications to be mapped one after another onto a variety of of candidate architectures to evaluate performance and cost of the resulting systems. Performance and cost numbers that are collected in a quantitative performance/cost analysis step are used, in one way or the other, to tune parameters in the architecture and/or the application models and/or to select options in the set of mapping methods. In the framework in which the Y-chart approach is implemented, the top three boxes appear as applications layer, mapping layer and architecture layer, respectively. The mapping layer translates *representations* of components of the application model into *representations* of components of the architecture model. For example, an application's actors communication protocol may be transformed (or refined) to an architecture's synchronisation and data transfer protocol. The mapping of applications into an architecture can only be done systematically if the models that are used to specify them do match. For example an application that is specified in an imperative MoC cannot really be mapped on an architecture that is specified in terms of, say, Co-design Finite State Machines [14]. Similarly, the mapping methods can only consist of property preserving transformations in a sound translator when the representations they operate upon and the representations they produce are well chosen and mutually convertible.

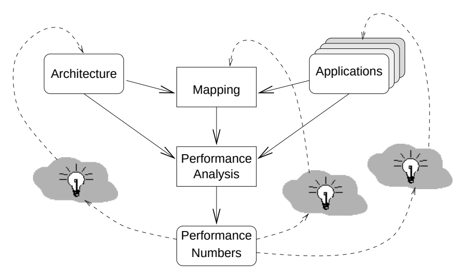

Figure 2. The Y-chart

In the past five years we have experimented with two Y-chart prototype systems: the `SPADE` system [6] and the `Archer` system [7]. Both have multimedia as their application domain, and both use the Kahn Process Network (KPN) MoC [4] to specify the domain's typical streaming applications. In `SPADE`, the processes are represented by traces of *Symbolic Instructions*. These *application traces*, T^{ap}, are then transformed to *architecture traces*, T^{ar}, that are interpreted by the architecture processing units. In `Archer`, the processes are represented by *Symbolic Programs* that are reminiscent of the well-known Control Data Flow Graphs. These *application programs*, P^{ap}, are translated in the mapping layer to *architecture programs*, P^{ar}, in a way that is reminiscent of classical compilations.

In the next section we review, as far as our needs go here, the models that are used in `SPADE` and `Archer`. Then, in Section 4, we discuss the various system-level representations and translation methods that we have been dealing with in our two prototype systems.

3. Models of Application and Architecture

As we said already, we specify applications in our domain as Kahn Process Networks (KPN). These are natural models to specifying streaming applications with a high degree of concurrency. A KPN models *task-level parallelism* as processes that communicate point to point over unbounded FIFO channels, using blocking read synchronisation. The processes or tasks are autonomous: their is neither a global scheduler, nor a global memory. Processes are, by definition, sequential programs written in a host language. A property of a KPN is that it models deterministic applications, that is, the input-output relation is independent of the order in which processes are scheduled [1]. A process that has to *read* from a channel will block whenever the FIFO channel is empty. If not empty, one or more channel tokens are read, some execution is initiated, and results that have to be send to another process are written unconditionally into the appropriate FIFO channel. In addition to the applications, and separated from them, we specify our domain platform or architecture template as heterogeneous compositions of MoCs. The components in the architecture are *Processing Units*, *Memory Units*, *Interconnection Structures*, and *Interfaces*. Communication protocols, synchronisations and data transfers are in general different from the applications MoC and may differ from component to component. Components may or may not have functional behaviour. If not, they are essentially exposing timing behaviour in which case the functional behaviour is captured in the application model. In the latter case, which is a typical exploration setup, application models and architecture model must be related through binding. This is the task of the mapping layer and its mapping methods. For example, when mapping a KPN model into a shared-memory architecture, synchronisation and data transfers must be separated by using finer grained primitives than the Kahn `Read` and `Write` primitives. Or, dependencies between primitives may have to be undone or re-ordered to expose intra-task parallelism if necessary to improve performance, reduce cost, or increase the confidence levels of the measures. The order in which such transformations are applied is important as this may have a substantial impact on the results. The mapping approach in `SPADE` and `Archer` are different. In fact, the latter is more general and more appealing, especially from the point of view that exploration and synthesis are following similar paths. The `SPADE` approach is simpler and more restrictive. It has its own merits and some of them have been preserved in the `Archer` approach. The two approaches are dealt with in the next section.

[1] Most applications in the domain are not strictly deterministic. However, the natural match between KPN models and streaming applications is too appealing that a choice of a different MoC would be favourable. In our prototype systems, in particular the `Archer` system, we will extend the KPN MoC with a *select function* to model a non-determinate merge. However, we do not go into this aspect in our current survey.

4. System-level Translation

As we have already argued, the mapping translator applies transformations on representations of the applications model components and returns representations that can drive the architecture components that are, then, interpreted by the same in a quantitative performance and cost analysis step. Different representations capture different aspects of the components and, hence, determine what are the possible transformations that can be applied. In the course of our experimentations, we have identified three types of representations and corresponding mapping approaches [5],

1. The Control Data Flow Graph (CDFG) representation and Retargetable Compilation (RC) approach,
2. The Symbolic Instructions (SI) representation and Trace Driven (TD) approach, and
3. The Symbolic Program (SP) representation and Retargetable Translation (RT) approach.

The main differences among these representations are with respect to their functionality (executability) and the extend to which they capture control constructs and dependencies. The CDFG-RC approach is the standard design/synthesis approach. The CDFG is fully functional (executable). It does not comply with the separation of applications and architecture concerns because, among other reasons, both functional and timing behaviour are fully captured by the architecture. Moreover, the CDFG-RC approach, although well suited for synthesis purposes, is too low-level and too detailed for fast exploration purposes. The SI-TD approach is a simple and appealing approach, at least from the point of view of the explorer: the TD mapping approach is by no means reminiscent of the methods used in a subsequent synthesis step. Finally, the SP-RT approach combines the pros of the two other approaches without suffering from their cons. It is reminiscent of the CDFG-RC approach in that it uses *Symbolic CDFGs* that can be transformed by a retargetable translator much as standard CDFGs are by a retargetable compiler. Also, it resembles the

SI-TD approach in that 1) the outcomes of conditional constructs (CC) in the application model are still collected in traces and 2) the SP is converted, with the help of the CC-trace, to a trace of symbolic instructions. Notice, that in the SI-TD approach as well as in the SP-RT approach, the performance/cost analysis is valid for only a single data set. Once again, in these two approaches, the functional behaviour is captured in the application model whilst the timing behaviour is captured in the architecture model. This is not so for the CDFG-RC approach. The three distinct representations and corresponding approaches as contrasted in Figure 3.

The Symbolic Instructions Trace Driven Architecture approach is depicted at the left of this figure. The processes in the application model are represented by traces of symbolic instructions `R` (Read), `E()` (Execute), and `W` (Write). These SIs are generated by the (annotated) processes upon execution of the KPN on a single data-set. These application traces, T^{ap}, are transformed in the mapping layer to architecture traces, T^{ar} that drive the architecture model in a quantitative performance/cost analysis step.

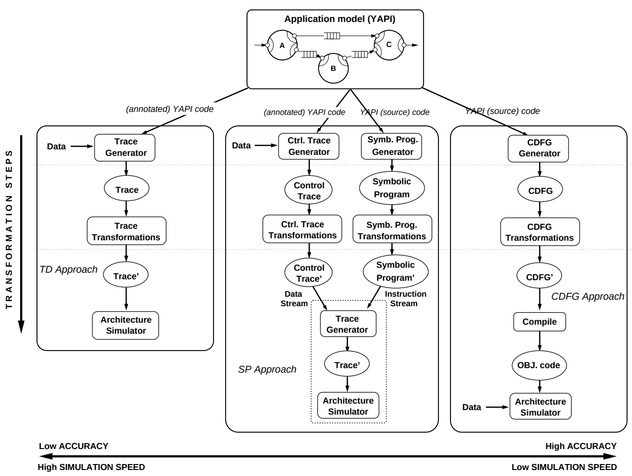

Figure 3. The positioning of the mapping approach

The traditional CDFG approach is depicted at the right in Figure 3. Here, the source code of the application model processes is parsed and converted into a structure preserving CDFG. The application CDFGs are then transformed to CDFGs which suit better the target architecture. Such architecture-level CDFGs are compiled into object code, which is then executed by a CDFG architecture simulator. A data-set is processed in this execution step, as opposed to the TD approach in which the data-set is processed in the application layer.

Finally, the Symbolic Program approach is depicted in the centre box of Figure 3. The Symbolic Program approach has emerged in an attempt to bring the TD approach closer to the traditional designer's approach without drifting too far away from the appealing property of the TD approach that allows fast exploration iterations [5]. Thus, a symbolic program is obtained in a similar way as a CDFG in the CDFG approach - by parsing the application sources. On the other hand, control information is collected in Control Traces that are generated by the application model processes upon execution of a single data-set. In contrast with the Symbolic Program, the Control Traces are only valid for a single data-set. Both the CTs and SPs are transformed in the mapping layer to architecture CTs and SPs. These are, then, combined to generate condition-free Symbolic Instruction traces that drive the architecture in the quantitative performance/cost analysis step.

In the next two subsections, we explain and illustrate CDFG and SI-TD approaches in more detail. The SP approach is described in the separate section.

4.1. Control Data Flow Graph Mapping Approach

The Control Data Flow Graph (CDFG) mapping approach is not a system-level exploration approach, but rather a system-level synthesis approach [9],[8]. In it, the application model is captured in a set of representations that preserve the executable functionality of all the application tasks, including control constructs. These representations take the form of Control Data Flow Graphs [21]. CDFGs express both mutual exclusivity and parallelism. The mapping layer, then, transforms these application layer CDFGs to architecture layer CDFGs where they are executed in RTL models of computation.

As mentioned, CDFGs may express both mutual exclusivity (conditional expressions) and parallelism (fork-join pairs). Thus, these representations are detailed, and, hence, very powerful. They are obtained by parsing task source code files. Then, the mapping layer takes as input the application processes CDFGs and produces transformed architecture processor CDFGs. The procedure is illustrated in Figure 4.

The upper part in Figure 4 shows a series of (possible) transformations that are applied to a CDFG representation of a process in an application model. Some of these transformations are architecture independent, some are architecture dependent. In the figure, the first transformation makes explicit the *intra-process* (or intra-task) parallelism that may exist between internal dataflow graphs of the process CDFG. The second transformation applies loop unrolling to reduce the number of control points in the CDFG. The next transformation is an architecture dependent refinement transformation separating that is applied to separating synchronisations and data transfers, for example as illustrated in [18]. Successive transformations and refinements introduce expansions and details, alternative primitives, and

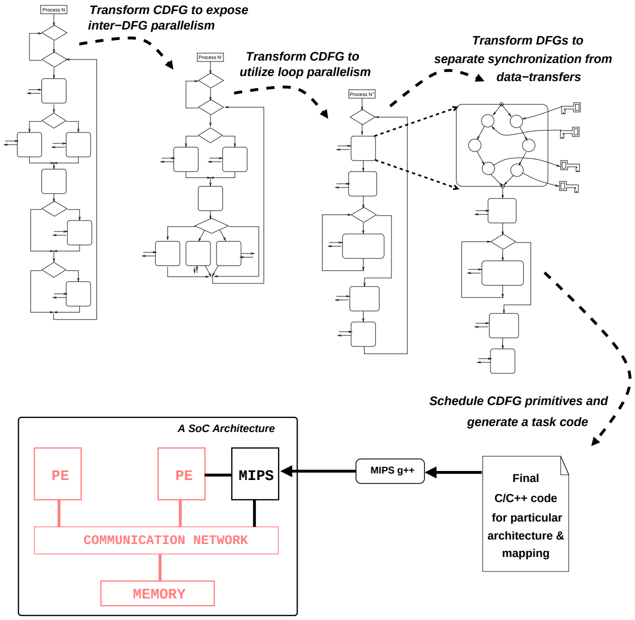

Figure 4. Control Data Flow Graph mapping approach

degrees of parallelism that can execute in the given architecture components. The lower part in Figure 4 shows the architecture model and the code that is generated from the architecture CDFG. This code is compiled and loaded into a MIPS kernel in the architecture.

During the execution of the architecture model, run-time scheduling may also be performed. Since functionality has been preserved in the mapping process, actual data sets can now be processed during architecture simulations. This may enhance the accuracy of these simulations. However, Control Data Flow Graphs are fine grained and, as a consequence, the corresponding quantitative performance and cost analysis is detailed, time consuming and hard to adapt for exploration purposes.

4.2. Symbolic Instruction Trace Mapping Approach

In Symbolic Instructions Trace model and co-simulation approach [6], an application model is executed on a single data set. Process behaviors are abstracted by annotating channel-read, channel-write, and function executions in the process. These become symbolic instructions `R`, `W`, and `E`, respectively, that are collected in application traces T^{ap} in the order in which they are generated. The application trace may reveal some information about available intra-task level parallelism to the extent that it can be made explicit in architecture independent trace transformations. These transformations are part of the mapping layer which translates or transforms T^{ap}s to architecture traces, T^{ar}s.

The architecture traces are driving the (non-functional) architecture model which interprets the transformed symbolic instruction in terms of performance and cost numbers.

As mentioned before, in this approach mapping is described as a set of transformations which transform the application into the architecture traces. In the case when there is a total ordering relation of symbolic instructions within a single trace, these transformations are relatively simple [18]. This is illustrated in Figure 5. Note that the processing unit in the architecture, as a consequence of the total order relation among the symbolic instructions in the trace, remains a simple sequentially executing unit, much as the application process unit. In fact, the architecture model is real close to the architecture model.

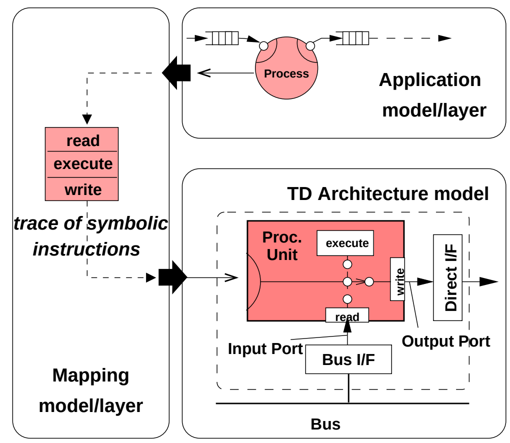

Figure 5. Symbolic Trace Instruction mapping approach

Logically, when more information is captured and when the relations induce a partial order, transformations become complex. Nevertheless, in both cases (with or without partial ordered information), symbolic instructions can not be moved around (*scheduled*) dynamically. This is due to the fact that traces do not contain the outcome of control dependencies which exist in the original application code. Thus, the expressiveness of the Trace-Driven approach is limited, and therefore, can in general not guarantee sufficiently accurate performance [20].

It may be still possible to use this exploration approach but it would require that the underlying architecture uses an extensive data-flow synchronization model, such as the *virtual processor* proposed in [23]. However, even in this case, the method fails in case non-static applications are to be mapped. This is so because capturing control data in traces leads to a possibly unbounded set of feasible application traces (or equivalent flow-graphs representing the virtual processors).

4.3. Symbolic Program Mapping Approach

Whereas the CDFG approach is a typical designer's approach that is not well suited for (fast) system-level exploration, the SI-TD approach is - conversely - a typical explorer's approach that is not attractive to designers. This is so because the models and the transformations that are involved in these 2 approaches are too distinct to allow easy conversion between the two. Thus, the limitations of the SI-TD approach are not only in terms of the its restricted expressiveness (as experienced by the explorer), but also in terms of its restricted usefulness (as experienced by the designer). The next representation and transformation approach tries to overcome both drawbacks of the SI-TD approach.

A symbolic program is an abstraction of the behaviour which was earlier given either as a Control Data Flow Graph or as a source-code description. It uses ideas of the *abstract execution* described in [11]. On one hand, the symbolic program syntax must be rich enough to be able to expose most important Control Data Flow Graph related information which a compiler (or a designer) may need while searching for both *intra-task* and *inter-task* parallelism. On the other hand, the syntax should not support fine-grain operations which are accustomed in CDFG representations (e.g., arithmetic & logic expressions, assignments, etc.), as this would complicate transformations and lead to slower simulations. Thus, what is needed is to find a minimal set of expressions that is sufficient enough to support both requirements.

A symbolic program is an abstract Control Data Flow Graph representation of an application process where vertices of a graph represent either (1) symbolic instructions, or (2) conditional control expressions, and edges of a graph represent the precedence relation between (1) and (2). Note that symbolic instructions are the already mentioned coarse read, write, execute communication and computation events defined when we introduced application traces.

A purely symbolic instruction trace misses (control) information. The conclusion was that this information has to be inserted somehow if we want a higher flexibility (and consequently a higher accuracy). Additionally, a major property of the Symbolic Instruction Trace approaches is that data is not processed in the architecture model. And, again, symbolic programs are a kind of a *generalisation* of symbolic instruction traces. Therefore, symbolic instructions cannot simply be replaced with symbolic programs, since control in symbolic programs needs to be resolved upon architecture simulation. However, the combination of control construct outcomes for a particular data-set (*trace of control outcomes*) can be provided by an execution of the application model so that with this information traces can again be generated from the symbolic program.

5. Symbolic Program Transformations and Architecture Driving

The Symbolic Program approach has emerged from an attempt to unify arguments that are in favor of fast exploration, and those that are in favor of design and synthesis practice. The SP approach is, thus, a hybrid approach. Applications processes are abstracted by means of 1) generic non-executable CDFG-like representation a process and 2) a symbolic instruction trace that captures conditional construct outcomes that are the result of the execution of a single data-set. Architecture model components are still driven by SI traces that are obtained by interpreting transformed application SPs knowing the control outcomes from the control traces. In the next two subsections we take a closer look at the transformation methods in the mapping layer, and the way the SPs drive the architecture model. We do this by example cases.

5.1. Symbolic Program Transformations

Symbolic programs use loop and selection descriptors and series-parallel sequences to express both algorithmic structure of a process and parallelism that is internally available in a process. Additionally, since the code is abstracted, transforming SPs is less costly than transforming CDFGs. This is illustrated in the example given in Figure 4, where the focus of an explore/designer is on a portion of the complete process functionality. As a consequence, it is easier to select transformations that are appropriate to apply. Moreover, automating these transformations is (in general) much simpler than in the case of the fully-functional CDFG. Apparently, the exploration based on this representation is faster and cheaper in a sense that a designer may see what will be a consequence of some of his choices even before he did the real job. As a consequence, he may want to choose different transformations, or even to go back and repartition the code.

We have already mentioned earlier that SPs come with actual control information in the form of control traces; one per application process (or task). Thus, when transformations, such as in Figure 6, are applied to an SP, corresponding control traces have to be transformed too. This can be automated since an refined control construct in a transformed SP can be easily expressed in terms of logical expressions over original control constructs. For example, in the case of the transformations SP1-to-SP2 and SP2-to-SP3, which are shown in Figure 6, the corresponding transformations of control traces are shown in Figure 7. Concretely, the goal of the SP1-to-SP2 transformation is to detect *variants* which divide the iteration space of the loops in mutually exclusive regions. Then, in each region possible parallel symbolic instructions are grouped into series-parallel se-

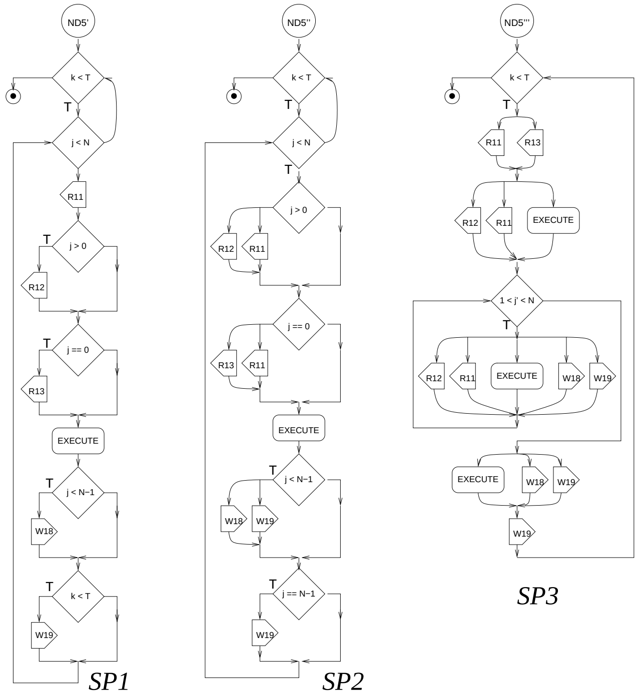

Figure 6. The CDFGs of a symbolic program of process ND5 (SP1) before and (SP2, SP3) after transformations being applied

ical circuit in Figure 7. Furthermore, when transformation SP2-to-SP3 is applied, selection statements are lost, so the trace events that correspond to these selections, namely c_3, c_4, c_5, and c_6, are simply ignored. Moreover, because software pipelining is applied (prolog and epilog code sections are generated at the expense of the number of inner loop iterations), the first two c_2 events in $ND5''$ are also ignored. The resulting $ND5'''$ control trace is shown in Figure 7.

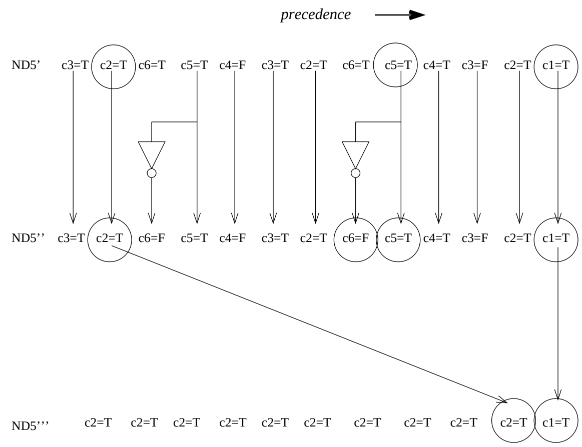

Figure 7. Transformations of control traces with regards to transformations of symbolic program of process ND5 in Figure 6: the topmost corresponds to SP1, the middle one to SP2, and the remaining one to SP3

quences. The transformation SP2-to-SP3 is obviously *software pipelining* [17]. It is possible only if there is not an "inter-task" cycle[2]. If possible, then it 1) reduces the number of control points (from 6 to 2 in the case illustrated in Figure 6), and 2) allows parallel execution of instructions in a loop-body (since they are independent in this case). Note that each conditional control point in $ND5'$ is enumerated as a c_i, where i is unique in the scope of $ND5'$. And similar for both $ND5''$ and $ND5'''$. In other words, c_1 stands for the outer loop condition $k < T$, c_2 stands for the inner loop condition $n < N$, and so on.

When the transformation SP1-to-SP2 is applied, the control trace of the process $ND5'$ needs to be transformed as well. This new trace is enumerated as $ND5''$ in Figure 7. Since the transformation did not change the CDFG structure from the point of view of loops, loop-related trace events are unchanged. However, because two selection statements at the end of the inner loop are merged, corresponding trace events, namely c_5 and c_6, in the control traces $ND5'$ and $ND5''$ they have a different meaning and, therefore, transformation is required. This is illustrated as a negation log-

[2]If $R11$, $R12$, and/or $R13$ instructions are "externally dependent" on $W18$ and/or $W19$, then $R11$, $R12$, and/or $R13$ will block since $W18$ and/or $W19$ are delayed by software pipelining technique.

5.2. Architecture Model Driven by Symbolic Programs

Having clarified the motivation, the usefulness, and the internals of the SP approach, we now describe briefly the architecture model components modules support this approach. For more detailed information, please consult [20].

Currently, there are three basic generic architecture component module types: 1) Symbolic Program Units (SPU), 2) Read-or-Write Interfaces, and 3) First-In-First-Out (FIFO) buffers. The idea is to obtain a small set of generic library components which will allow easy modeling of streaming heterogeneous multiprocessor architectures. While the purpose of SPUs is to model the intra-task level parallelism[3], the purpose of the other two modules is to allow modeling of Task Level Parallelism (TLP).

The TLP is utilised by execution of different tasks on different SPUs in parallel. For example, a process network (modeled as a Kahn Process Network) shown in Figure 8 can be easily mapped onto an architecture shown in Figure 9 in order to exploit TLP; tasks $ND3$ and $ND4$

[3]Actually, SPUs model high-level manifestations of ILP.

are computation-intensive, so the underlying SPU modules may be *application specific processors*, while $ND1$ and $ND2$ are just source-nodes, so corresponding SPU modules may be *general-purpose processors*. Thus, SPUs must be generic enough to allow the construction of these two distinct processor models.

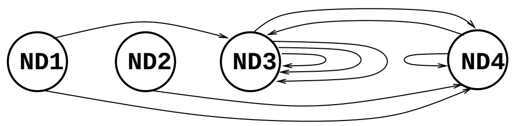

Figure 8. An application process network

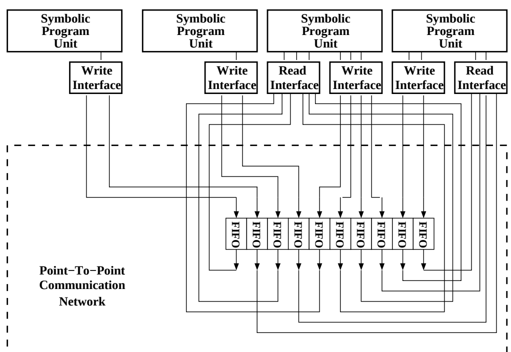

Figure 9. A SP-based architecture instance related to the process network shown in Figure 8

Additionally, it may be required that models exploit the intra-task level parallelism (i.e., series-parallel symbolic instruction sequences in symbolic programs). As a consequence, the purely sequential processor model shown in Figure 5, which can run only a single symbolic trace instruction at a time, cannot exploit this parallelism. On the contrary, SPU modules can exploit intra-task parallelism. It allows a flexible deployment of the potentially parallel load originating from the application process and, therefore, it allows designers to model both static and dynamic instruction scheduling [16] at a high-level of abstraction.

The SP approach is shown in Figure 10. After a parallel (process network) specification is obtained either automatically [19] or by hand-coding, each process is parsed and SP-abstractions are applied. That is, symbolic programs, described using a predefined syntax, are translated to the corresponding data structure. This data structure is often known as a parse tree. In the same time, control traces which correspond to processes are obtained by running the process network. These inputs are forwarded to each SPU on top of which a process from the process network is mapped. Naturally, in order to instantiate SPUs an

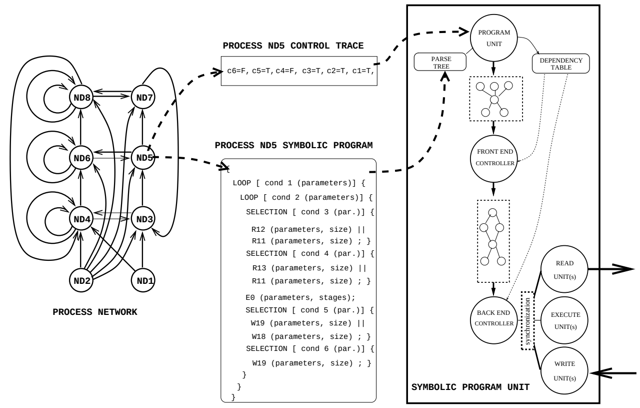

Figure 10. Symbolic Program Unit and the corresponding SP flow

architecture specification is also required. After a simulation is finished one can choose to perform transformations on SPs in order to see how different application-architecture combinations will behave when evaluated in in a subsequent simulations.

6. Conclusion

In [20] we performed an example case study based on the application described in [10], where we showed that the symbolic program approach, as an exploration approach, is superior in comparison to both symbolic instruction trace and control data flow graph approaches. This stems from a few key issues concerning system-level design-space exploration: (1) ease of modeling streaming multiprocessor systems, (2) accuracy of simulating these systems, and (3) speed of quantitative performance/cost analysis (simulations). The accuracy is determined by modeling capabilities. Modeling restrictions (as in the SI trace-driven approach) may hamper correct modeling and lead to inaccurate results as shown in [20]. Conversely, unrestricted, detailed and inflexible modeling may over-specify models and lead to inflexible and costly explorations with levels of accuracy that are not really relevant. The SP-approach, in contrast, incorporates the important characteristics of both extreme approaches (namely, the fast trace-based simulations, and the CDFG control constructs), and consequently improves the weak aspects of both extreme approaches (namely, the inaccuracy of the symbolic instruction trace approach, and the inefficiency of the CDFG approach). This yields a design-space exploration approach that is both accurate and efficient, while it connects well to more detailed design/synthesis trajectories.

7. Acknowledgements

This paper contains the work performed in part in the `Archer` project, funded by Philips Semiconductors. We want to thank Kees Vissers and Bart Kienhuis for their contributions concerning the Abstraction Pyramid and the Y-chart. We also want to thank Paul Lieverse for his work that provided the Trace-Driven co-simulation prototype system `Spade`, as well as his Trace Transformations theoretical work. We are grateful to Andy Pimentel for his sharing with us his recent experiences with dataflow modeling of trace transformations. Finally, we want to thank Todor Stefanov for his valuable suggestions and remarks concerning the SP approach and the writing this paper.

References

[1] B. Kienhuis, et al., "An Approach for Quantitative Analysis of Application-specific Dataflow Architectures," in *Proc. of ASAP'97*, July, 1997.

[2] B. Kienhuis, et al., "A Methodology to Design Programmable Embedded Systems," in the LNCS series of Springer-Verlag (c), Volume 2268, SAMOS: Systems, Architectures, Modeling, and Simulation, November 2001.

[3] B. Kienhuis, "Design Space Exploration of Stream-based Dataflow Architectures: Methods and Tools", *PhD thesis*, Delft University of Technology, the Netherlands, January 1999.

[4] G. Kahn, "The semantics of a simple language for parallel programming," *Information processing 74* - North-Holland Publishing Company, 1974.

[5] V. Živković et al., "Design Space Exploration of Streaming Multiprocessor Architectures," at *SiPS'02*, USA,Oct.,2002.

[6] P. Lieverse et al., "A methodology for architecture exploration of heterogeneous signal processing systems,"in *Proc. SiPS'99*, Taiwan, Oct. 1999.

[7] Archer - ARCHitecture ExploRation of heterogeneous embedded systems at system-level of abstraction, http://www.liacs.nl/ cserc/archer.html

[8] F. Thoen et al., "Multi-Thread Graph: A System Model for Real-Time Embedded Software Synthesis," Proceedings of the 1997 European Design and Test Conference, pp. 476-481, 1997.

[9] N. Cossement et al., "DF*: An extension of synchronous dataflow with data dependency and non-determinism," in *FDL'00*, Germany, Sep. 2000.

[10] T. Harriss et al., "Compilation from Matlab to Process Networks Realized in FPGA," in *the 35th Asilomar Conf. on Signals, Systems, and Computers*, US, Nov. 2001.

[11] J. Larus, "Abstract execution: A technique for efficiently tracing programs," *Soft. Practice & Experience*, Dec.,1990.

[12] "SystemC Version 2.0 User's Guide," *Synopsys, Inc., CoWare Inc., Frontier Design, Inc.*, http://www.systemc.org/

[13] D. Gajski et al., *System-level Abstraction Semantics*, ISSS'02, Japan, Oct. 2002,

[14] F. Balarin, et al., "Hardware/Software Co-Design Of Embedded Systems - The POLIS Approach," Kluwer Academic Publishers, 2nd printing, 1999.

[15] E. A. Lee et al., "A Framework for Comparing Models of Computation," *IEEE Trans. on Compuetr-Aided Design of Design of Integrated Circuits and Systems*, Wol. 17, No. 12, December 1998.

[16] J. Hennessy et al., *Computer Architecture - A Quantitive Approach*, Morgan Kaufmann Publishers, 1996.

[17] M. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," in *Proc. of SIGPLAN'88*, USA, June, 1988.

[18] P. Lieverse et al., "A Trace Transformation Technique for Communication Refinement," in *Proc. CODES'01*, Denmark, Apr. 2001.

[19] A. Turjan et al., "The Compaan Tool Chain: Converting Matlab into Process Networks," in *DATE'02*, France, 2002.

[20] V. Živković et al., "Fast and Accurate Multiprocessor Architecture Exploration with Symbolic Programs," submitted for *DATE'03*, Munich, Mar.,2003.

[21] W. Wolf, "Computers as Components - Principles of Embedded Computing System Design," Morgan Kaufmann Publishers, 2001.

[22] A. D. Pimentel, et al. "Modeling of Intra-task Parallelism in Sesame", in the Proc. of the 2nd Int. Workshop on Systems, Architectures, MOdeling, and Simulation (SAMOS 2002), Samos, Greece, July, 2002.

[23] A. D. Pimentel et al., "An IDF-based Trace Transformation Method for Communication Refinement," Internal Report, University of Amsterdam, http://carol.wins.uva.nl/ andy/publications.html