A front - end for better behavioral synthesis

Gauthier, Lovic
Institute of Systems & Information Technologies/Kyushu

Devroye, Natasha
Institute of Systems & Information Technologies/Kyushu

Tomiyama, Hiroyuki
Institute of Systems & Information Technologies/Kyushu

Murakami, Kazuaki
Institute of Systems & Information Technologies/Kyushu

出版情報：IPSJ Technical Report 2002-SLDM-107-6, pp.31-36, 2002-10. 情報処理学会SLDM研究会
バージョン：

A front-end for better behavioral synthesis

Abstract

By allowing higher-level descriptions, behavioral synthesis has been an important solution to cope with the growing complexity of the chips. However, its efficiency has never met the one of RTL synthesis. Our goal is to define a flow for automatically converting such high-level specifications to ones that can be efficiently handled by synthesis tools. This flow can be seen as a front-end for those tools.

1 Introduction

Progresses in integration allow the design of chips that contain millions of transistors. Therefore it becomes possible to design complete systems within a single chip. These chips, called SoC (System on Chip), are very interesting for embedded systems as the latter have to integrate more and more functionalities and have strong constraints in term of size and power consumption. However, it is impossible to design in a reasonable time such complex chips with classical methodologies.

To aid in the definition of methodologies able to handle the previous challenges, new kinds of hardware/software specification languages have been defined. SpecC[7] is one of the most popular of those new languages featuring: object-oriented concepts, hierarchical description, abstract communication through channels, C-like syntax (well known by a majority of designers), and fast execution. We can also mention SystemC[1] which has almost the same features.

It is important to notice that these languages are similar to behavioral VHDL[2]: the main difference being the data handling (which is performed through signals in the case of VHDL, and through software variables for SpecC). This difference makes SpecC easier to use and faster to simulate. However, behavioral VHDL synthesis tools have never been a complete success[4][6]: they are often quite efficient in some specific cases, but fail in general. Therefore, this shortcoming should be the same for a SpecC synthesis tools.

We propose a flow that aims to solve, or ease, these problems at high-level by an appropriate pre-computing of the "specification". This flow uses various techniques for determining the functionalities or the implementation domain (for instance dataflow oriented), for reusing in-library components, and performing high-level exploration. The "specification" we want to address is very close to the paper specification but is still written in the executable SpecC language. This flow can be used as a front end of existing synthesis flows.

The rest of the paper is organized as follows: the second section described some works related to ours, the third section presents the proposed flow and the fourth section describes its use on the JBIG encoder example. The final section is the conclusion.

2 Related work

SpecC methodology[8] implemented in the SCE tool[11], proposes a several step refinement: architecture refinement which corresponds to architecture exploration (including hardware software partitioning), communication refinement and implementation refinement (which correspond to behavioral synthesis). One of this methodology's strong points is its using of precise and orthogonal semantics. This methodology relies on the user for the behavior coding and for the exploration decisions. Apart from this, it considers that the input code (called "specification") can be directly synthesized, as if it was an implementation. This implies that the user have to write a "synthesis-efficient specification" which is not trivial. Moreover, several specific cases can be efficiently synthesized by specific tools but not by general tools[4].

Some of the techniques our flow uses are based on previous works. Code recognition is one these techniques. A lot of work have already been done about it: in verification[5], in software compiling[9], in logic synthesis[10], etc. However these works focused mainly on recognizing low level optimized implementations for validation whereas our goal is to recognize high-level straight forward specification. In-library component reuse (IP-reuse) is also deeply studied, the main difficulty there being to find the best component[12].

3 A pre high-level synthesis flow

3.1 Hardware specifications

In this paper, we call **paper specification** specification such as the standard norms' documents like the JPEG encoding. For our input specification we choose the SpecC specification level as defined in SpecC methodology[8]. But, contrary of this methodology, we try to be as close as possible to the **paper specification**. For that purpose we simply transcript it into the SpecC formalism, without thinking about its future implementation.

3.2 Presentation of the flow

The input is a high-level description obtained by simply translating the **paper specification** into the SpecC description language. The output is an architecture level description defined in the SpecC methodology. This flow assumes that the hardware/software partitioning has already been decided. It only focuses on static parts of the system: dynamic parts (mainly software) have to be treated in another flow. Before being treated by the flow, the input text specification is converted to a hierarchical set of graphs whose top graph represents the different modules of the initial specification (called **behavior** in SpecC). Each **behavior** may contains other **behaviors** or some code. As for the actual behavior (the function bodies), they are converted to control-data flow graphs.

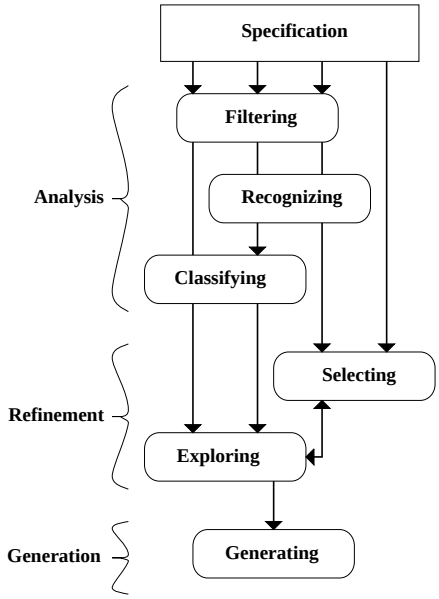

Figure 1: The proposed pre-synthesis flow

Figure 1 presents the proposed flow. This flow can be divided into 3 different parts:

- The analysis part that includes the **filtering** the **recognizing** and the **classifying** steps. The goal of this part is to translate the behavioral description into a set of symbolic objects. The **filtering** quickly annotates each node of the graph with its local properties, the **recognizing** tries to indetify precisely the functionalities (**services**) and the **classifying** tries to regroup specification parts by common kind of implementations (**domains**, e.g. dataflow oriented or memory intensive). **Domains** can be used during the refinement part to decide which behavioral synthesis policy should be used.

- The refinement part tries to find the best component and the best architecture using some performance annotations. This is a recursive step that includes the **selecting** step (that find the component from the library) and the **exploring** step (that choose the best architecture for the components).

- The generation part that will produce the output files (SpecC). This generation is performed through the assembly and the expansion of optimized in-library macros corresponding to the recognized part of the specification. The non recognized parts are also assembled with the macros, but their iner code is kept untouched.

As seen in the figure, there are 4 ways (from left to right) to handle a part of the specification: from a specification part which goes through each step to one that goes directly to the selecting step. This last case occurs when the specification part is a black box annotated with a **service**, or directly a component (**module**) to use. The idea is that the flow is still useful without one or several of its steps.

Note: the **recognizing** step is not interesting for all the cases. For now we have defined 3 grain levels that are interesting to recognize. The first grain level includes the "complex operators", they represent some computation operations that are not usually represented by operators, but that are commonly implemented in hardware computation units. For example the $1/\sqrt{x^2+y^2+z^2}$ function is such an operator. The second grain level includes the generic algorithm patterns, for example all the different kind of loops (unsorted, sorted, dependant on the indexes and so on). The last one includes common behaviors that are often used but hidden and dispatched within a more complex algorithm.

This flow works with a library whose contents is: for the first part, the symbolic object and the way to identify them; for the second part, the components and performance annotations; and for the third part the code elements.

3.3 The objects used in the flow

The flow uses several families of objects. These objects can be stored within the library, or can be linked to the specification (introduced by the user or computed during the flow).

The first family regroups all the plug-in functions for the flow stored in the library. Plug-ins can be used for the **filtering**, the **recognizing** the **classifying** and the **exploring** steps and are respectively called the **filters**, the **recognizers**, the **classifiers** and the **explorers**.

The second family regroups all the symbols used by the flow to represent some high-level concepts. These objects are the **services** and the **domains**. A **service** represents a functionality or a group of functionalities. For example a DFT (Discrete Fourier Transform) or a generic divide and conquer algorithm can be represented by a **service**. **Services** are the output of the **filtering** and the **recognizing** steps. A **domain** represents an information for implementation. For example dataflow oriented can be represented by a **domain**. **Domains** are the output of the **filtering** and the **classifying** steps.

The third family regroups all the **parameters** that can be associated to the specification. They can be user's annota-

tions, or they can be computed during the **filtering** or the **classifying** steps. A **parameter** is more precisely a name associated to a type and a value. They are used by the **exploring** and the **generating** steps.

The fourth family regroups all the **modules**, the **ports** and the **channels**, that is to say the components that implement the **services** and their interactions. For example a DFT can be implementated by a **module**. Several **modules** can implement a same **service**. They can also be hierarchical containing other **modules**, **ports** and **channels** or **services** (a module can provide or require some **services**). Finally a **module** can be partially defined or generic: it is called a **pattern**.

The last family regroups all the material for generation: it is a set of macros that are expanded and assembled in order to obtain the output of the flow, that is to say the behavioral code.

4 The flow applied on the JBIG encoder

In order to validate our methodology, and to estimate the difficulty for designing such tools, we applied it on a real specification: the JBIG image compressing method[3].

4.1 Presentation of the JBIG

The JBIG[3] is a lossless bi-level image[1] compression encoding method. It also has a "progressive" capability, which makes it possible to display low resolution images before the complete image being available (in case of low-band transfer for example). Figure 2 shows the global encoding flow: it repeats D times[2] a resolution reduction and a differential layer encoding (i.e. encoding the differences between two consecutive resolutions of the input image), and finally it applies the lower resolution encoder. The figures 3 and 4 detail the two steps of the flow.

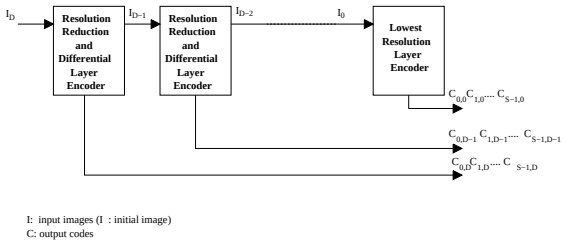

Figure 2: The JBIG encoding flow

In figure 3, the resolution reduction produces a lower resolution image trying to keep the image quality[3]. Typical prediction looks for most probable pixel conformations knowing the lower resolution image, and deterministic prediction looks for the pixels whose value can be deduced from the encoding mechanism. Adaptive template and model template are used to compute the contexts related to the pixels that will be used by the adaptive arithmetic encoder to produce the output code.

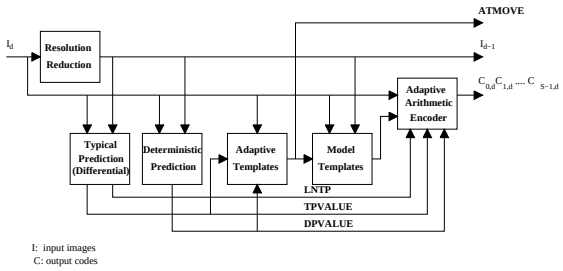

Figure 3: The resolution reduction and differential layer encoder part of the JBIG

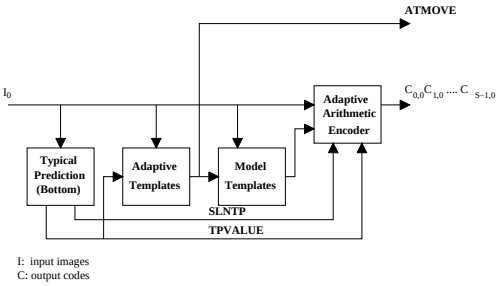

Figure 4: The lower encoder part of the JBIG

In the lower resolution encoding, shown in figure 4, the typical prediction is similar to de previous one, but it uses only one resolution.

4.1.1 Translation of JBIG paper-specification to a SpecC specification

Almost all the encoding process has been translated to the SpecC specification. Only the adaptive templates has not been described because it was not necessary, and let unspecified by the norm.

The translation has been straight forward as almost all the algorithms are given in a pseudo-code manner or are limited to access to some lookup tables. More precisely, a recurrent operation in the specification is to build for each pixel an index from its neighbors. This index is then used to access a lookup table. It is the case for the resolution reduction, the typical prediction, the deterministic prediction, and also the adaptive templates. For this, the **paper specification** gives the lookup table, and a diagram showing the neighbors of a pixel annotated with the index's bit they correspond to. Therefore the translation consisted only in copying the tables and coding the memory accesses.

Regarding the hierarchy, we strictly followed the one given by the **paper specification**.

In total, the translation process took less than one man week, with basic verifications. Of course, more time would be necessary for a complete verification, but as the translation was mainly a simple copy (sometimes automatic), the main errors should be only typing-like ones. What is

[1] Bi-level images, like black-and-white images, have only two colors.

[2] D is the lower resolution reduction rate.

[3] A straight forward resolution reduction, like keeping only one pixel over two, strongly degrades the image quality.

important to notice is that, on purpose, no effort has been spent for producing an "optimized" specification: neither for the behavior nor for the architecture.

In the other sections only the work on the resolution reduction and differential layer encoder will be presented as the one on the lower resolution encoder is very similar.

4.1.2 Filtering step

The filtering step has been applied independently on each behavior of the SpecC specification. It consists in quickly analyzing each variable and each node of the specification graph, and annotate them with the corresponding **services** and **domains**.

Variable analysis gives similar results for the resolution reduction, the typical prediction, the deterministic prediction and the model template. The following kinds of variables have been found: some integer variables used as loop and array index and also used as limit checkers, some 2 dimension bit (or bit vector) matrixes used as input or output memories, some 1 dimension fixed bit or bit vector arrays used as look up tables (not present in the model template and the typical prediction), and some temporary integer/bit vectors (both types are used) used as lookup index (not present in the typical predition). Typical prediction has another output variable: the LNTP bit.

Variable analysis for the arithmetic encoder bloc return the following kinds of variables: some integer variables used as logical, arithmetic and comparison registers (we mean by register that they are not dataflow temporary variables that could be easily removed), some integer variables used as loop indexes, some bit variables used as streamed output, some 1 dimension fixed integer or bit arrays used a lookup tables, and some 1 dimension bit (or bit vector) arrays used as input memory.

After variable analysis, node analysis has been applied.

Using these informations, several groups have been built for recognition. The first kind of group contains all the 2 dimension memory accesses using the loop indexes. The second one contains all the loops that englobe the algorithms. The third one contains the limit checks. The fourth one contains the lookup accesses. The fifth one contains the bit accesses in the lookup indexes. The sixth one contains the arithmetic and logic operations (for the arithmetic encoder). The seventh one contains the mask and shift operations (for the arithmetic encoder). Even if these groups have been given here independantly on the bloc they come from, they are still linked to them in the methodology. For instance, recognizing and classifying will be first applied on each group of each bloc. Another important remark is that these groups sometimes overlap on each other: it is for example the case of the seventh and the sixth group (in fact the seventh group is included in the sixth one).

For this experiment, each of these kinds of groups has been assumed to be potentially recognized. Due to this assumption, they are all sent to the recognizing step.

4.1.3 Recognition step

Consecutive 2 dimension memory accesses (the first filtered group) within loops are very common in image processing algorithms. Recognition step applied on the first group are performed simply analyzing the loop and array indexes (comparing them to some classical 2 dimension memory accesses within loops). They give different results according to the bloc they come from. For resolution reduction, 2 dimension memory accesses are local, centered to the index, and do not recover over the loop iteration. The corresponding service is simply a 9-pixel bloc access to a 2-dimension memory. For the differential typical prediction, two kind of 2-dimension memory accesses are used: a 9-pixel bloc access as previously, and a 9 pixel bloc access with 6 pixels recovered. For deterministic prediction and for model template, 9-pixel bloc access with 6 pixels recovered and and 25-pixel access with 20 pixels recovered. One should notice that the number of pixels within the blocs are not part of the **services**, but are **parameters** added to the corresponding nodes.

For the second group, a specific **recognizer** for loops must be used. It looks for data dependencies inside the loop with the loop indexes with the goal of determining which kind of loop it is. For all the specification parts, the code within the loop uses the loop indexes, but their utilization differs: for instance, for the resolution reduction, in order independant, and same memory accesses are not repeated over iterrations whereas deterministic prediction is order dependant and same memory accesses can be repeated over iterations.

For the third, the fifth and the sixth groups no interesting service has been found, so they will be transferred directly to the classifying step.

For the fourth group, the lookup accesses are considered as random (as they completly depend on the input), therefore the corresponding service is simply lookup access.

For the seventh group, shift and masks are in fact only bit accesses. Therefore the **service** register bit range access is used.

4.1.4 Classifying step

First the input specification graph is first reordered so that node within the same group became the closest possible.

From these results (and the filtering annotations) the splitting were just cutting when the **domain** of the nodes change. The splitting result is shown figure 5. For other input specifications though, splitting can be much more difficult, especially if different **domains** are randomly interleaved. Elaborate techniques have then to be used, like fuzzy computations.

Final annotations put the proportion of different domains of each split part thanks to the classifying table.

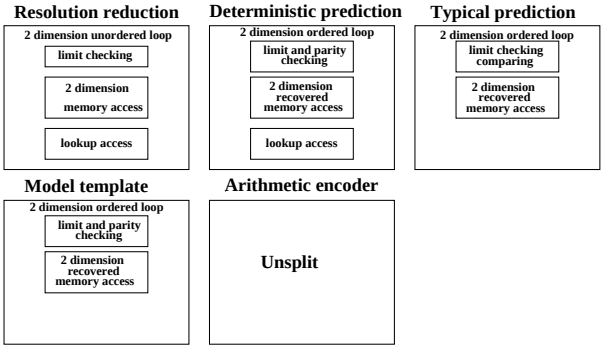

Figure 5: The splitting result

4.1.5 Selecting and exploring steps

As the library is not yet built, exploration and selection steps for this example has been rather limited. In fact, this example brought the opportunity to design the first modules for the library.

For the application, one module for selection is providing a 2-dimension memory access **service**. The figure 6 gives the proposed implementation for this module. It is a generic module whose parameters are the shape of the pixel buffer, the number of pixels that have to be read in each pass and the functions applied on the buffer (in our case, these functions are the building of the lookup indexes, the context computation, and some comparisons). One could argue, that such architecture could have been found with a memory access optimization tool, which may be true. However, the use of such a tool can also be a result of our flow through the classifying step. Moreover, these specific memory accesses are very common so it may be better to have an already made finely tuned component for that. One should also notice that the choice of such a module strongly depends on the kind of memory which is accessed.

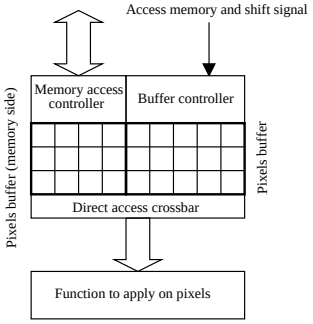

Figure 6: The implementation for the 2-dimension memory access

The rest of the specification can be directly handled by lower level tools. Lookup accesses are quite common, therefore it may be interesting to add in the library some **modules** for them.

Preliminary exploration results are shown in figure 7: The first level of hierarchy groups all the module into a loop modules[4] The second level of hierarchy groups in a first **module** the 2 dimension memory accesses within the loops, in a second **module** all the lookup accesses and tables and in a third **module** the rest. The third level of hierarchy within the first **module** separates the resolution reduction memory accesses from the others (as the first one has no memory recover contrary of the others). Within the third **module**, the next level of hierarchy groups together the comparisons and initializations, and the rest of the arithmetic encoder is left there.

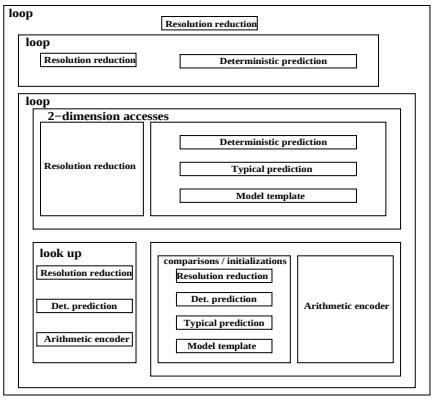

Figure 7: The result of the exploration step

4.1.6 Conclusion about the results

This final result shows that the resulting architecture is completely different from the initial one. Regarding the behavior, all the 2-dimension memory accesses has been replaced by a single library-based implementation. The rest of the behavior has been kept, but its order has been changed to put together parts with similar domains.

For the arithmetic encoder, only the lookup accesses have been treated, the rest remains untouched, and should be treated as is by the lower level synthesis tools. The main reason is that the code of this part does not contain typical patterns. Further work on our flow will be therefore necessary to handle such kind of behavior, especially in the **filtering** and **classifying** steps.

Finally simple simulation have been performed at the RTL level with different memory accesses times. Their results are shown in the table 4.1.6 [5]

In the table, the first column is for the specification translated directly to the RTL level without the initial flow, and the second one for the “optimized” one. The two first lines give the number of read and write accesses to the memories, the last lines give the total number of cycles required for encoding the input image with different memory access times (respectively 10 cycles, 2 cycles and 1 cycle). As seen in the table, even with fast memory accesses, the “optimized” version obtains better performances. The best performance are mainly due to:

[4]Resolution reduction and deterministic prediction have a part outside the x-loop as a complete line have to be computed by them before the rest of the algorithm can be applied.

[5]The algorithm complexity is proportional to the image’s size; in our simulation we used a 80x24 pixel image.

Measure	Initial specification	"Optimized" specification
read	54197	11418
write	7786	3406
cycles (10)	704034	163735
cycles (2)	197674	41263
cycles (1)	704034	26079

Table 1: RTL simulation results

1. the fewer memory accesses resulting from the 2 dimension memory access component;

2. the grouping inside the same loops of the different steps of the JBIG encoding algorithm.

The "optimized" version also requires less memory[6] as the exploring step make it possible to exploit locality of the image processing.

Performance similar to ours can be obtained with classical flow. However, the intial specification have then to be completely rewritten to adapt it to the behavioral synthesis tools, which is time consuming and error-prone.

5 Conclusion

SpecC is a useful language for describing complex SoCs at a high-level of abstraction. However the problems encountered with behavioral synthesis tools have shown that it is difficult to generate efficient hardware from high-level descriptions: either the result is bad, or the input specification is difficult to understand.

Our goal is to convert high-level straight forward SpecC specification to lower level complex ones that can be efficiently synthesized. In this paper we first defined our specification, and then proposed a flow that can be considered as a front-end to existing synthesis flows. The flow contains several steps that treat the input specification following different axes: recognizing, classifying and exploring. The initial filtering prepares the specification (by annotating it) for these steps, and the selecting step allows the use of in-library already designed components.

As preliminary experiments, this flow has been applied on the JBIG encoder. The results shows that, even if the whole specification has not been successfully handled (the arithmetic encoder has been mostly untouched), the resulting synthesis-optimized architecture and behavior strongly differ from the initial ones. Using existing synthesis flow without this front-end would force the user to perform a complex translation of the initial specification to one that could be efficiently synthesized, therefore, when automated, this flow will bring an important gain of productivity. Moreover, with the growing library, the productivity will increase over time.

The results are promising, however there is still a lot of work do to obtain the automated flow. Additional works are also planned to define a library that could use the results of synthesis to enhance its annotations for achieving better selecting and exploring results.

[6] Apart from the input image and the output codes, there are memory needs for only 2 lines.

References

[1] Technical report, SystemC language, available at: http://www.systemc.org.

[2] Behavioral compiler. Technical report, available at: http://www.synopsys.com/products/beh_syn/beh_syn.html.

[3] *IUT-T Recommendation T.82*.

[4] Raul Camposano. Behavioral synthesis. In *DAC*, 1996.

[5] P. Camurati and P. Prinetto. Formal verification of hardware correctness: Introduction and survey of current research. *Computer*, 1988.

[6] Wander Oliveira Cesário, Zoltan Sugar, Imed Moussa, and Ahmed Amine Jerraya. Efficient integration of behavioral synthesis within existing design flows. In *ISSS*, 2000.

[7] SpecC Consortium. Technical report, SpecC language and methodology, available at: http://www.specc.gr.jp/eng/index.htm.

[8] Daniel D. Gajski, Jianwen Zhu, Rainer Dömer, Andreas Gerstlauer, and Shuqing Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, 2000.

[9] Robert Metzger and Zhaofang Wen. *Automatic Algorithm Recognition: A New Approach to Program Optimization*. MIT Press, 2000.

[10] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. 1994.

[11] Center of Embedded Computer Systems. Soc design environment. Technical report, available at: http://www.ics.uci.edu/špecc/research/index.html.

[12] Ting Zhang, Luca Benini, and Giovanni De Micheli. Component selection and matching for ip-based design. In *DATE01*, 2001.