Graph Based Communication Analysis for Hardware/Software Codesign

Peter Voigt Knudsen and Jan Madsen
Department of Information Technology, Technical University of Denmark
pvk@it.dtu.dk, jan@it.dtu.dk

Abstract

In this paper we present a coarse grain CDFG (Control/Data Flow Graph) model suitable for hardware/software partitioning of single processes and demonstrate how it is necessary to perform various transformations on the graph structure before partitioning in order to achieve a structure that allows for accurate estimation of communication overhead between nodes mapped to different processors. In particular, we demonstrate how various transformations of *control* structures can lead to a more accurate communication analysis and more efficient implementations. The purpose of the transformations is to obtain a CDFG structure that is sufficiently fine grained as to support a correct communication analysis but not more fine grained than necessary as this will increase partitioning and analysis time.

1 Introduction

In this paper we focus on communication analysis for hardware/software partitioning of control-intensive applications that are specified using hierarchy, functions, conditionals and loops. In particular, we focus on the structures that implement control, i.e. conditionals and loops. These structures are used to direct the flow of data between functional elements according to the values of test variables. As communication overhead is an important factor to consider during hardware/software partitioning [4][5], the mapping of these structures is thus important to analyze and optimize. The presented CDFG model supports the exploration of various implementation alternatives for these structures through conditional and loop transformations which will be demonstrated in the following. Furthermore, it supports communication analysis for cross hierarchy communication through *hierarchical expansion* and for function calls through *virtual function expansion*. Virtual function expansion is only described briefly in this paper. The purpose of the transformations is to obtain a CDFG structure that is sufficiently fine grained as to support a correct communication analysis but not more fine grained than necessary as this will increase partitioning and analysis time.

Name	*Alias*	*Comment*
PURE_DFG	D	A pure dataflow graph
FULL_LOOP*	FL	A whole loop node
LOOP_BODY*	LB	Loop body node
LOOP_ENTRY	LE	Loop entry node
LOOP_EXIT	LX	Loop exit node
FULL_BRANCH*	FB	A full branch node
BRANCH_BODY1*	B1	First branch body
BRANCH_BODY2*	B2	Second branch body
BRANCH_SPLIT	BS	Branch variable split node
BRANCH_MERGE	BM	Branch variable merge node
REPEATER	R	Repeater node
HIER_IN	Hi	Hierarchy input interface node
HIER_OUT	Ho	Hierarchy output interface node
FU_CALL*	F	Function call node
FU_IN	Fi	Function input interface node
FU_OUT	Fo	Function output interface node
NOP	N	NOP (variable duplicator) node
VOID	V	Void node (variable sink)

Table 1: *Elements of NodeType. Hierarchical nodes are marked with an asterisk(*).*

2 CDFG model

This section defines the CDFG model which is used to describe the functionality of a single process. It includes structures for basic arithmetic and logical operations, hierarchy, conditionals, loops and functions and is as such sufficiently expressive as to be able to represent universal computation power [2].

The CDFG can be denoted a *high level CDFG* as nodes represent high level functions rather than simple operations, either in the form of function calls or in the form of data flow graphs (DFGs) containing simple arithmetic and logical operations and no control, and edges represent variable sets that are communicated between the high level functions rather than single variables.

Nodes can have different types, as defined in table 1. The alias column defines short forms of the type names that will be used in figures. Nodes are records that contain a number of parameters, as defined in table 2.

Edges are also records and contain the parameters defined in table 3. Edges can be either data or control edges, as distinguished by the EdgeType parameter.

The usage and meaning of the various node/edge types and fields will be defined as they are used in the following sections[1].

[1] Only the parameters that are relevant for this paper are shown in the table. The type (Variable $\overrightarrow{m}$ Variable) denotes a map (sometimes called dictionary) that maps variables to variables.

Name	*Type*	*Comment*
type	NodeType	The type of a node.
rset	Variable-**set**	The set of variables read by a node.
wset	Variable-**set**	The set of variables written by a node.
cdfg	CDFG	The CDFG this node is a part of.
subcdfg	CDFG	The sub-CDFG of a hierarchical node.
subdfg	DFG	The dataflow graph of a DFG node.
tvar	Variable	The branch, loop and rpt. node test variable.
tpol	Bool	The test variable polarity.
bmap1	Variable $\rightarrow_m$ Variable	Branch variable mapping for branch body 1.
bmap2	Variable $\rightarrow_m$ Variable	Branch variable mapping for branch body 2.
emap	Variable $\rightarrow_m$ Variable	Loop entry node variable mapping.
xmap	Variable $\rightarrow_m$ Variable	Loop exit node variable mapping.
rmap	Variable $\rightarrow_m$ Variable	Repeater node variable mapping.

Table 2: *The parameters associated with a node.*

Name	*Type*	*Comment*
type	EdgeType	The type of an edge (DATA or CONTROL).
src	Node	The node that feeds an edge.
snk	Node	The node that is fed by an edge.
varset	Variable-**set**	The set of variables transferred on a data edge.

Table 3: *The parameters associated with an edge.*

Note that the fact that hierarchical nodes can reference a subgraph via the subcdfg parameter makes the CDFG definition recursive. A CDFG contains nodes but nodes may themselves contain whole CDFGs.

As for operational semantics of the graph, we use token flow semantics as defined in [1]. This means that variables are tokens that flow on edges and that each node executes according to a *firing rule* that defines that output tokens are generated for each output variable when and only when all input tokens are present on the input edges. During this process, the input tokens are absorbed by the node.

As mentioned, edges do not correspond to single variables but to *sets* of variables that are communicated between nodes. Therefore, we denote these edges *hyper edges*.

For a given node, an input hyper edge is created for each node that feeds it and an output hyper edge for each node that it feeds itself.

The individual variables that are communicated on edges can in general be of any type. For simplicity, we only model simple variables like integers and reals and arrays of simple variables. This means that we need only record the bit width and length of each variable (the length of a simple variable is 1). Using the methodology in [5], the communication time for read- and write sets that, according to analysis, need to be transferred between different processors using particular protocols can be estimated with the help of these parameters.

3 Transformations for communication analysis

Traditional CDFG formats like the one described in [1] use a separate edge for each variable that is communicated between nodes in the graph. If for example seven variables are transferred between two nodes, seven edges will be the result. Our coarse grain CDFG format supports fast communication analysis as we use a single hyper edge between each pair of nodes that communicate with each other. This reduces the number of edges and therefore decreases analysis and partitioning time. For implementing conditional and loop structures, we use special control nodes that direct the flow of data according to the values branch or loop test variables, as described in sections 3.2 and 3.3. These nodes correspond to multiplexers/demultiplexers in hardware and to conditional or loop constructs in software. The format in [1] uses one control node for each variable in the system, leading to a very large number of control nodes to consider for partitioning. While this fine grain graph format allows for maximum flexibility with respect to partitioning control structures, it also complicates the graph and therefore increases analysis and partitioning time. Our graph format allows for exploring the whole range from using just two large control nodes for each control construct to using control nodes for each variable. In the following sections we demonstrate how graphs with large control nodes can be transformed into sufficiently fine grained structures that allow for better optimization of communication. These transformations improve both efficiency of the final implementation and accuracy and efficiency of analysis. It is important to note that, while the transformations allow for exploring different implementation alternatives for loops and conditionals, they should only be performed to the extent that the synthesis tools are able to produce similar implementations. If, for instance, the hardware synthesis tool can only produce a coarse grain loop control implementation (i.e. using one controller and single big multiplexers/demultiplexers), the loop control nodes should *not* be transformed in the graph prior to doing partitioning. The graph structure must reflect what is done in synthesis, even if what is done is not efficient. For a further discussion of the relation between the model domain and the implementation domain, please refer to [4].

In the following, we first introduce a basic transformation called *hierarchical expansion* which eases the analysis of cross hierarchy communication and which is a prerequisite for performing the subsequently presented conditional and loop transformations correctly.

3.1 Hierarchical expansion

Hierarchy is introduced by letting hierarchical nodes (those marked with an asterisk in table 1) reference a CDFG. The node H in figure 1A is such a hierarchical node. We use double circles in figures to denote hierarchical nodes.

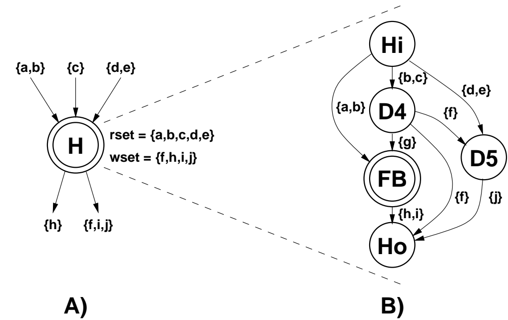

Figure 1: *Structure of a CDFG hierarchy. A) Atomic (node) view. B) Expanded view.*

All subgraphs of hierarchical nodes are polar graphs as shown in figure 1B. D4 and D5 are DFGs and FB is a full branch node whose sub-CDFG is not shown. We see that the hierarchical node H in figure 1A is fed by three nodes and feeds two nodes itself. The hierarchy CDFG of a hierarchical node always contains a hierarchy input node Hi and a hierarchy output node Ho. These nodes act as an interface to the hierarchy and as placeholders for the variables that go

in and out of the hierarchy[2]. The write set of the Hi node equals the set of variables that are read from outer hierarchies. The read set of the Ho node equals the set of variables that are written to outer hierarchies. We assume that every variable that is produced in a CDFG is unique with respect to its name throughout the whole CDFG, i.e. throughout all hierarchy levels of the CDFG.

As mentioned in [7], one of the first steps in the codesign process is to determine the granularity of the functional specification that partitioning operates on. This can be done in a number of ways [3][6][7], the simplest being *hierarchical granularity selection* [6] where we for each hierarchical node determine whether it should be regarded as a granule (i.e. atomic function which is not split across processors) or whether we should replace the hierarchical node with the contents of the hierarchy and thus make the input specification more fine grained. Our graph structure supports communication analysis for both cases. If the hierarchical node H is to be regarded as a granule itself, we simply use the input- and output hyper edges shown in figure 1A for communication analysis for a particular processor mapping of the node H. If the *contents* of the hierarchy is to be regarded

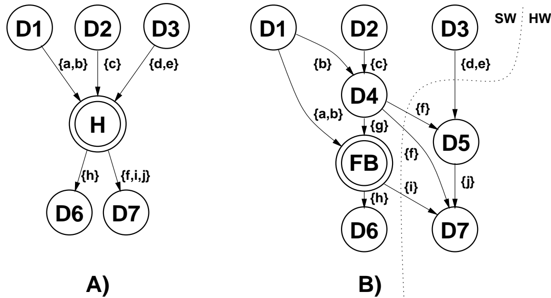

Figure 2: *One-level expansion of a hierarchical node. A) CDFG prior to expansion of H. B) CDFG after expansion.*

as granules, we perform *hierarchical expansion* in order to be able to perform a correct data dependency analysis for a particular mapping of the nodes inside and outside of the hierarchy to different processors. This is shown in figure 2 where the hierarchical node H, corresponding to the hierarchy in figure 1B, is expanded into its surrounding CDFG. The expansion is a one-level expansion as the full branch within the CDFG of node H is not expanded, but of course expansion can be multi level. Note that when performing hierarchical expansion, the Hi and Ho nodes are eliminated and hyper edges are regenerated so that we can analyze the true dependencies between the nodes inside the hierarchy and the nodes outside the hierarchy.

In the example, D5 and D7 are placed in hardware while the rest of the nodes are placed in software. This expansion, for example, allows us to see that even though D7 reads three variables, {f,i,j}, it only needs to have two variables {f,i} transferred across the hardware/software boundary.

Note that it is legal for the same variable to be present on several edges when more than one node reads the variable, as it is the case for the variables b and f in the figure. When several nodes that read such a shared variable are mapped to another processor than the producing node is mapped to, there are several possibilities for scheduling the corresponding edges. If dynamic memory storage on the receiving processor allows it, the variable needs only be transferred once, for the first scheduled node (D5, for the variable f). For subsequent edges that contain the variable (the one from D4 to D7 for f), such an already transferred variable can be removed from variable set of each edge which decreases the communication time of the edges and possibly allows subsequent nodes (D7 for f) to be scheduled earlier. If memory storage on the receiving processor is limited and memory storage on the transmitting processor allows it, the variable can be stored temporarily on the transmitting processor, retransmitted each time it is needed by a receiving node and freed when the last receiving node has been scheduled. Determining the optimal time/space mapping of shared variables can be done by introducing variable duplicator nodes whose mapping and scheduling in effect determine in which time slots the variables are stored on which processors. This is left to future work.

3.2 Branches

Branches or conditional structures are introduced by using full branch, branch body 1, branch body 2, branch split and branch merge nodes. A full branch hierarchical node is used to encapsulate the whole branch. The basic structure of a conditional is shown in figure 3.

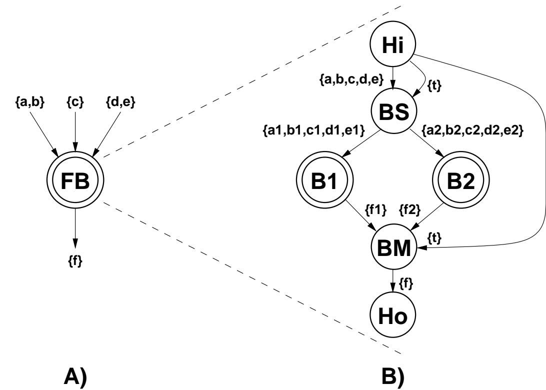

Figure 3: *Basic structure of a full branch. A) Node view. B) Expanded view.*

The BS node is a branch split node that duplicates its input variables and sends them to either B1 or B2, depending on the value of the test variable (t in the figure). The BM node is a branch merge node that selects the output variables from either B1 or B2, also depending on the value of the test variable, and outputs the corresponding branch output variables. The test variable is identified by the tvar field of the BS and BM nodes. A test polarity parameter (tpol) of the BS and BM nodes specifies which of the branches that is taken if the test variable is true. If the test polarity is true, B1 is taken, otherwise B2. In order to keep track of how input variables map to output variables of the BS and BM nodes, we use the bmap1 and bmap2 variable maps which define the mappings for B1 and B2, respectively. In the examples we have used the intuitive mapping that a variable named x outside of a branch maps to the variable x1 in B1 and to x2 in B2.

[2]This makes the hierarchy graphs polar and corresponds to the implementation of hierarchy in the *flow graph model* defined in [2].

3.2.1 Transformation for unshared variables

In figure 4A we see that the (copies of the) variables {a,b} are used solely by B1 and {d,e} solely by B2[3]. If the branch is implemented using only a single BS node, such variables must be led through the BS node which may be very inefficient, depending on the mapping of the BS node. Figure 4B shows a transformation that allows such variables to be communicated directly from their producing node to the branch they are used in.

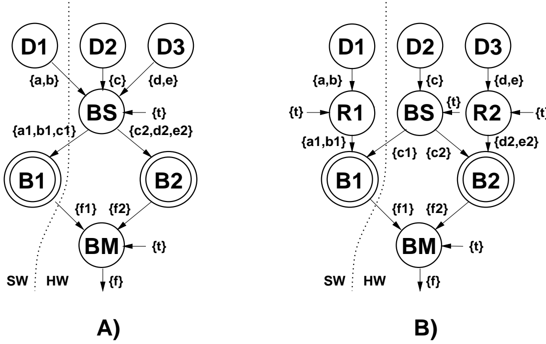

Figure 4: *Transformation for unshared variables. A) Original branch structure. B) Transformed branch structure.*

Here we have expanded the branch into a surrounding hierarchy where D1 supplies the {a,b} variables, D2 the {c} variable and D3 the {d,e} variables. The branch test variable is disregarded in the rest of this section. In figure 4A, we have assumed that the branch has been constructed in such a way that all variables read within the branch are led through the branch split node. In figure 4B, a repeater node is added for each of the source nodes of the branch split node that produces variables that are only read by one of the branches. These repeater nodes are called R1 and R2 in the figure. A repeater node copies its input variables to its output variables (according to the rmap variable map) if the value of the repeater test variable (tvar) is equal to the value of its polarity field (tpol). Otherwise it absorbs its input variables. Repeater nodes for B1 must have the same polarity as the branch split node and repeater nodes for B2 must have opposite polarity[4].

Assume that we know that the left branch B1 is taken so that the BS node does not communicate variables to B2. In the un-transformed case in figure 4A, communication analysis shows that six variables cross the hardware/software boundary because it is not recognized that {a,b} can be communicated directly from D1 to B1. In the transformed case in figure 4B, only two variables cross the hardware/software boundary.

We find that a similar transformation is not needed for the branch merge node because the two branches produce equivalent sets of output variables.

Note that the B1 and B2 nodes are regarded as granules in this example. If granularity selection has determined that they should be expanded, this expansion must be performed *before* the branch optimization so that repeater nodes are generated with respect to the nodes inside the branch hierarchies. In general, we have that hierarchical expansion must be performed before transformation.

[3]The unused variables {d1,e1,a2,b2} are assumed to be absorbed within the BS node. Indeed, assuring this for all unused variables is another transformation that we perform but which is not shown here.

[4]Note that the token flow semantics of the CDFG mean that we can *not* use a simple hyper edge instead of a repeater node. We should only direct variables (tokens) to the active branch, and, for this, a repeater node is needed.

3.2.2 Transformation for shared variables

This section describes a transformation for those variables that are read by (and produced by) *both* branches, like c in figure 4.

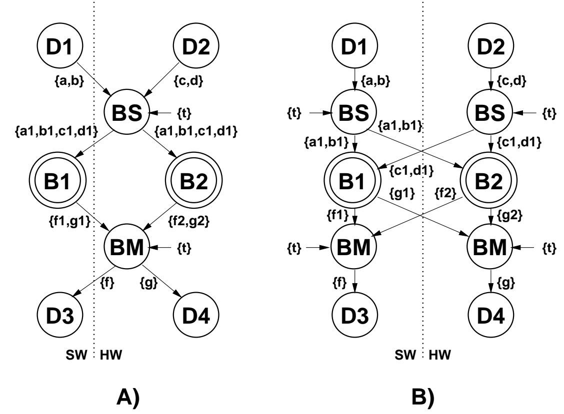

Figure 5: *Transformation for shared variables. A) Original branch structure. B) Transformed branch structure.*

Consider the branch structure in figure 5. Here the variables {a,b,c,d} are read by both branches. With the given structure, it is not recognized that the (copies of the) variables {a,b} can be led directly from D1 to B1 and that the (copies of the) variables {c,d} can be led directly from D2 to B2. If we assume again that the left branch B1 is taken, we see for the structure in figure 5A that 9 variables must be moved across the hardware/software boundary. In figure 5B, the BS and BM nodes have been split and communication analysis now shows that only three variables {c1,d1,g1} have to be moved across the boundary. Notice how the f and g output variables are now led directly to D3 and D4.

Splitting of the BS node must be performed for each of its source nodes that produces at least one variable that is read by both branches. Such a source node may also produce variables that are only read by one of the branches. Such variables are still transferred to the original branch split node or to a repeater node, as described in section 3.2.1.

Splitting of the BM node is currently performed for each of its sink nodes. If, however, several sink nodes share variables in their read sets, this leads to several branch merge nodes that produce the same variable. Either, one of these branch merge nodes must be selected as the sole producer of such a variable, or the produced variables must be renamed, as we do not support two nodes producing the same variable. We use the last strategy.

3.3 Loops

We use the structure shown in figure 6 to represent a full loop. LB is the loop body that also produces the loop test variable t. The loop is a REPEAT UNTIL loop[5] that executes LB until the value of the test variable t is false. LE is a multiplexer that initially, when t is false, directs the input

[5]A REPEAT UNTIL loop can always be transformed into a WHILE loop by enclosing it in a conditional [2], so the graph structure can represent both kinds of loops.

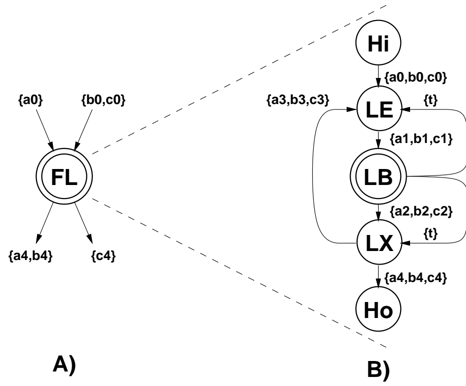

Figure 6: *Basic structure of a full loop. A) Node view. B) Expanded view.*

variables of the full loop, {a0,b0,c0}, to LB. When t becomes true, it directs the output variables from the LX node, {a3,b3,c3}, back into LB. A false token is assumed to have been placed on the t edge of all LE nodes before execution of the graph as to ensure that the loops start when they receive their first input variables. LX is also a multiplexer that directs its input variables {a2,b2,c2} back to LE as long as t is true and out of the loop (to Ho in the figure) when t becomes false.

We perform the single LE/LX node split transformation shown in figure 7 in order to obtain a loop structure that allows us to analyze communication between nodes within the loop more accurately. This transformation is performed with respect to the nodes *within* the loop as these nodes may communicate a large number of times with the LE/LX nodes while nodes outside of the loop only communicate one time with the LE/LX nodes. The splitting is performed by producing one LE node for each of the sink nodes of the original LE node and one LX node for each of the source nodes of the original LX node. It may be the case that several nodes within the loop read the same variable from the original LE node, thus causing several LE nodes that produce the same variable to be generated. This is currently handled the same way as described in section 3.2.2, i.e. by variable renaming.

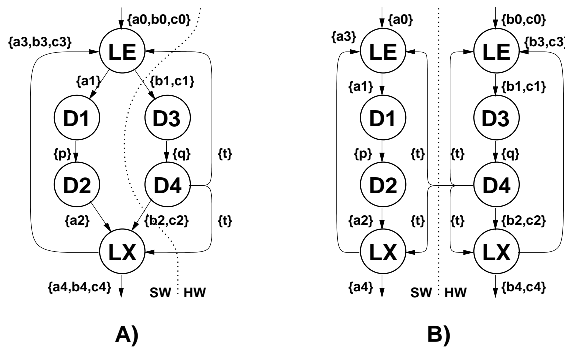

Figure 7: *LE/LX node split transformation. A) Initial loop structure. B) Resulting loop structure.*

Figure 7B shows the resulting loop structure in which it is apparent that only t needs to be transferred across the hardware/software boundary for the given mapping. In figure 7A, we have that five variables must be transferred between hardware and software for each loop iteration.

3.4 Transformation of the full graph

In order to obtain the full CDFG structure on which partitioning and analysis is to be performed, we first perform a recursive hierarchical expansion of all hierarchical nodes that should be expanded according to granularity selection. This expansion includes a CDFG wide regeneration of hyper edges. Thereafter, the branch and loop transformations described in the previous sections are performed for each loop and branch structure. Furthermore, we perform so-called virtual expansion of functions where each function call is fully expanded, i.e. (recursively) replaced with a copy of the function implementation CDFG. During this expansion, formal parameters of the function are recursively replaced with actual parameters (yielding new names for variables on input and output edges of the function graph) and internal edge names of the CDFG made unique (as to avoid collision with other virtually expanded instances of the same function), so that a correct data dependency analysis can be performed with respect to nodes that feed the function call and nodes within the function. Function expansion is denoted *virtual* as it is only performed in order to analyze communication correctly, not for mapping nodes of functions to processors (i.e. we do not assume *inlining* of functions). Mapping of the nodes of a function graph is performed only once, and this mapping is retained for each of the nodes of each virtually expanded instance of the function.

4 Conclusion

We have presented a coarse grain CDFG format that is useful for performing hardware/software partitioning of control intensive processes. We have shown that loop and conditional structures can be specified at different levels of granularity and that it is important to choose the right granularity in order to be able to perform a correct communication analysis and an efficient exploration of implementation alternatives for these structures. We have developed a tool that can translate a VHDL process into this CDFG format and which can perform the transformations described above. Future work includes integrating this with hardware/software partitioning and communication estimation in the LYCOS [6] co-synthesis system.

Acknowledgements

This work is supported by the Danish National Center for IT Research under grant no. CIT 149.

References

[1] G. G. de Jong. Data flow graphs: system specification with the most unrestricted semantics. In *Proc. European DAC*, pages 401 – 405, 1991.

[2] R. K. Gupta. *Co-Synthesis of Hardware and Software for Digital Embedded systems.* Kluwer Academic Publishers, 1995.

[3] J. Henkel and R. Ernst. A Hardware/software Partitioner Using A Dynamically Determined Granularity. In *Proc. 34th DAC*, pages 691 – 696, 1997.

[4] P. V. Knudsen and J. Madsen. Aspects of System Modelling in Hardware/Software Partitioning. In *Proc. 7th RSP Workshop.*, pages 18 – 23, 1996.

[5] P. V. Knudsen and J. Madsen. Integrating Communication Protocol Selection with Partitioning in Hardware/Software Codesign. In *Proc. 11th ISSS*, pages 111 – 116, 1998.

[6] J. Madsen, J. Grode, P. V. Knudsen, M. E. Petersen, and A. Haxthausen. LYCOS: the Lyngby Co-Synthesis System. *Design Automation for Embedded Systems*, 2(2):195 – 235, 1997.

[7] F. Vahid. A Three-Step Approach to the Functional Partitioning of Large Behavioral Processes. In *Proc. 11th ISSS*, pages 152 – 157, 1998.