A DESIGN REPRESENTATION FOR HIGH LEVEL SYNTHESIS

Mikael R.K. Patel*

Computer Aided Design Laboratory
Department of Computer and Information Science
Linköping University
S-581 83 LINKÖPING, SWEDEN
E-mail: mip@da.liu.se

ABSTRACT
TAO is a hierarchical graph representation of behaviour for High Level Synthesis of Hardware Structures. Typically a High Level Synthesis System takes a behavioural description and a set of constraints as input and generates a structural description of a hardware realization as output. One of the main questions when realizing an internal design representation is what data structures should be used to reduce time complexity of algorithms applied and how to organize these data structures. When using the TAO graph representation one must consider several types of graphs operations such as node merge and distribution. These require different representations to reduce the overall computational complexity of the procedure at hand. In this paper data structures for the three levels of TAO, Task, Algorithm, and Operation graphs, are selected, defined, and discussed with examples of typical graph operations performed during the synthesis process from a behavioural towards a structural description.

Keywords: Design Automation, High Level Synthesis, Design Representation, Data Structures.

1. INTRODUCTION
The area of High Level Synthesis is a relatively new discipline within Design Automation of Electronic System [17]. The main research goal of High Level Synthesis is to achieve theories, methods, and tools that will allow automatic generation of hardware structures (VLSI) from high level specification. Typical input to a synthesis system is a set of concurrent asynchronous tasks described in a high level programming or a hardware description language, together with a set of constraints such as the maximum cost of the design, maximum delay time of response, and design libraries to use. This paper presents the data structures used for a hierarchical design representation called TAO, which is designed especially for supporting the synthesis process.

TAO is a three level hierarchy of graphs describing the concurrent Task, the Algorithm, and Operation level of system behaviour. The task level corresponds to a global data flow model of a set of communicating sequential processes [2]. The tasks interact without time or clocking dependencies through ports on channels. The interaction may be realized as either synchronous or asynchronous depending on if the ports are associated with buffers or not.

The behaviour of each task is described by a graph on the next level; an algorithm graph. This graph expresses the overall control structure of the task. Each node corresponds to a sequential block of operations and the arcs possible branching directions depending on conditions after the evaluation of the associated block. The block of operations is described on the lowest level; the operation graph. This graph defines the partial ordering of the operations. The hierarchy of graphs is mainly motivated by the need to reduce the size of the problem due to the time complexity of most graph algorithms, and the need for capturing the different types of parallelism on separate levels.

The three levels in TAO capture different types of parallelism [9], and allow different sets of transformations to be applied on the way towards a hardware realization. The lowest level allows functional parallelism to be exploited. Several operations may be scheduled within the same major clock cycle in the synthesized architecture if they do not violate the design constraints, e.g., partial ordering and hardware resources. By transforming the algorithmic level the size of the operation graphs may be increased and thus the level of available parallelism. A number of such transformations are well known from compilers for vector machines and Very Large Instruction Width (VLIW) architectures [4]. Also classical basic block transformation such as common sub-expression elimination, copy propagation, dead-code elimination, code motion, etc., are applicable. The highest level, the task level, allows partitioning into physical devices by evaluating feasible clusters of the tasks to form physical unit, e.g., chip. Also if two task have a high level of interaction and only a small portion of the two tasks exploit asynchronous parallelism then the two tasks may share the same controller and data path. This may be achieved by merging the two tasks, i.e., composing their algorithm level graphs, or time multiplexing the two tasks on the same controller.

Within the area of High Level Synthesis a number of different design representation have been presented and used [5, 6, 7, 12, 15]. These fall mainly into three major categories; data flow graphs, compiler oriented data structures, and system state graphs. The data flow graphs are very similar to the operation level of TAO but with the extension that the whole computation is expressed as one uni-level graph. Thus to be able to handle description of conditional execution fork, join and selection nodes are added to the data flow graphs. Iteration is often considered to be handled separately as these are problematic to

* This research is supported by the Swedish National Board for Technical Development (STU).

2024/90/0000/0374@01.00©1990 IEEE
handle with preserved program semantics and high levels of parallelism as shown by Dataflow Architecture research [29].

Extensive work has been carried out at Carnegie-Mellon University (CMU) in the area of High Level Synthesis using a data flow graph like representation called Value Trace (VT). Two major projects, the Design Automation Assistant (DAA) and the System Architect’s Workbench, used ISPS behavioural description [3] which are then transformed into VT-graphs [26]. All transformations, optimizations and partitioning are then performed on the VT-graph.

Data flow graphs have also be used by the Advanced Design Automation Systems (ADAM) project at University of Southern California [19]. N. Park and A.C. Parker have also shown how data flow graphs may be used effectively to synthesize pipelined data paths [18]. But as the CMU research group the data flow graphs are restricted to only one outer loop and any additional iterations have to be handled externally. The data flow graphs are also only one level and thus many graph algorithms experience long execution times when the size of the graphs grow.

Compiler oriented data structures have been used by H. Trickey in his Silicon Compiler for Pascal: Flamel [27]. H. Trickey calls the internal data structures dacon. These capture the traditional block level analysis, and the computation within a block (DAG). The block level building of a transformation tree, and selection of a set of transformations on block level to increase the level of functional parallelism are the major contributions. The dacon is a two level hierarchy of graphs. Both temporal and asynchronous parallelism are not addressed by this research.

The system state graphs such as the Extended Timed Petri-Nets (ETPN) proposed by Zebo Peng [23] allows system behaviour to be expressed as a Petri-Net together with a graph representation of the data path. The Petri-Net captures concurrent threads of action without defining these as explicit tasks [24], and the data path graph the current structure of the design. The CAMAD system, based on the ETPN design representation, uses a transformational approach to synthesis. Through a set of semantic preserving transformations the initial ETPN, with data path, is modified towards a structural realization within the given design constraints; time and cost. The major drawbacks of this design representation is the lack of hierarchy, and that the concurrent tasks are not directly visible and available when attempting to partition the design into chips or other packages. Also parallel paths in the ETPN give exponential explosion of the number of states need to realize the control as a finite state machine.

The following sections will present and discuss the attributes needed to express the hierarchical graph structure of TAO, and how these may be represented as data structures. To illustrate and motivate the power of the selected data structures some of the basic procedures applied during the synthesis process are also presented.

2. OPERATION GRAPH
The operation graph level of the TAO design representation defines the partial ordering of basic operations. In a hardware design these represent operations performed on functional units in a data path. The operation level graph may be compared to a data flow graph with a single assignment assumption. The arcs in the graphs describe the data dependencies between nodes and are not attributed with conditions etc. Initially the graph only contains the operations to be performed. But during the synthesis process these operations are assigned resources and time slot, i.e., scheduled. Registers and interconnections are initially abstracted as the scheduling procedure is only concerned with the operations.

For each node in the operation graph the following set of attributes are needed in the initial synthesis process;

1. The operation to perform by the operation node
2. The set of variables and operation which are used
3. The set of variables which are defined
4. And last, the set of operations that follow

The operation attribute allows mapping between the abstract function from the source program to the physical device implementing the operation, i.e., Module Binding. This operation may later be bound to the actual bit pattern that should be fed to the device to select the appropriate function. This is often the case with Arithmetic Logic Units (ALU) [8].

An example, a plus operation, "x + y"; from a statement in a source program has to be mapped to a physical hardware unit that can perform the operation. The unit to select depends on a number of circumstances such as the size of the operands, i.e., the data type, short, long, etc, and the available resources in the data path. This mapping between the operation graph node and the functional unit has to be established during the synthesis process. But during the initial steps of the synthesis process, i.e., scheduling, a full mapping does not have to be accomplished. It is enough to know that there exists a mapping. This mapping has only to be achieved within a scheduled set of operation nodes and not for the whole operation graph at once.

![Fig. 2.1: Operation Graph for "z = y - z; x = (x + y) + z"](image)

The set of variables and operations which are used by the operation node define both the input domain of each node, and together the input domain of the whole operation graph. Each set is ordered from left to right according to the operation. The expression, "y - z", uses the variables, y and z. These are ordered in the used set from left to right, [y, z], so that commutative and association attributes of operations may be used during the transformations of the operation graph. For leaf nodes the used set will only contain variables.

The next set, defined, contains the set of variables that are assigned by the operation node. Normally, only the root nodes have a non-empty defined set. A variable is only allowed to be defined once in the graph; single assignment assumption. After the scheduling of an operation graph, internal nodes that prior to the scheduling lacked defined variables, may then have been assigned temporary variables.

Last of the basic attributes for an operation node is the set of nodes that follow. This set defines both the control and data
flow dependency of the node. This set is empty for root nodes. Two basic recognizers are defined, leaf, and root of nodes:

Data Structure Definition 2.1: The operation graph is represented as an array of nodes each with the above fields;

array [node] (operation, used, defined, follow)
1. node ∈ N: integer in the range 0..[N]−1
2. used(node) ⊆ N ∪ V:
   ordered collection of nodes or variables
3. defined(node) ⊆ V: collection of variables
4. follow(node) ⊆ N: collection of nodes
5. variable ∈ V, V ⊂ identifiers:
   identifier in the set of variable identifiers
6. leaf(node): V n ∈ used(node): variable(n)
7. root(node): follow(node) = Ø
8. precede(node) ⊆ N: {n ∈ used(node) | node(n)}
   Set of preceding nodes.
9. V v ∈ V: |{n ∈ N | v ∈ defined(n)}| ≤ 1:
   The single assignment assumption.

There may exist several leaf and root nodes in an operation graph. The graph must be a directed acyclic graph (DAG) of operations. During the different stages of the synthesis process additional attributes are needed. The above set of attributes per node are only the basic set. Typically, an attribute for graph colouring is required for list-scheduling, and links into the source code to give information about the correspondence between the nodes and the initial source code lines. Below follows a list of some of the optimizations that may be performed on the operation graph:

1. Common Sub-expression elimination,
2. Algebraic identities,
3. Reduction in strength,
5. Height reduction [13]

Common sub-expression elimination can be achieved when building the operation graph from the program source statements. Height reduction requires analysis of the graphs structure and the types of operations in the graph to apply the laws of associativity, commutativity, and distributivity. When the time to perform the individual operations in the graph have high variation, clustering of sub-graphs may be performed to build a graph with similar time requirements.

A number of scheduling methods and algorithms [25] have been presented. The most general method is an exhaustive search of schedules within the design constraints. But when the number of nodes in the graph becomes too large this is not so attractive, because of the exponential properties of such algorithms, and thus some heuristic must be applied. One method of scheduling is to colour the graph according to the level from the root. When selecting among a set of nodes to schedule one may then apply the simple heuristic of trying to schedule those nodes that are furthest from the root nodes first [17]. An other heuristic is to calculate the strength that each operation has on its neighbours and use this to guide the selection [22].

Let us now continue to the next level in the hierarchy of graphs in TAO, which is the algorithm level.

3. ALGORITHM GRAPH
The algorithm graph of the TAO hierarchical design representation determines the overall behaviour of a task, i.e., the control structure. Each node in the algorithm graph "holds" an operations graph and branching relations to other nodes. The computation model used in TAO is 1) execute the operation graph associated with the current node in the algorithm graph, and 2) evaluate the branching to the next node. The branching is, in principle, the evaluation of all arcs from the current node to find one that is true, and moving to this node. The requirement on the branching set is that only one, and exactly one, of the branching arcs must be true after the evaluation of the operation graph. The boolean expressions on the set of arcs from a node must thus be disjoint. The algorithm level may be viewed as a finite state machine description [11].

On this level of behaviour description some of the interesting optimizing transformations are 1) reduction of execution time, 3) merging and distribution of nodes, and 4) unrolling of loops. Allocating hardware resources for a time critical section of the algorithm graph may allow an efficient schedule of the associated operation graphs. Applying a graph transformation such as the merging of two nodes will require the composition of the associated operation graphs. This will grow the size of the operation graph and result in a higher levels of parallelism. For all of these transformations the cost-time factor are quite high. Distribution of nodes will increase the number of control steps in the synthesizer controller and higher levels of parallelism will increase the cost of the data path. A node in the algorithm graph should contain the following set of basic attributes;

1. The operation graph associated with the node
2. The execution frequency of the node
3. The execution time of the node
4. The branch set with their boolean expressions
5. The schedule of the operation graph

The frequency attribute is derived from typical executions of the algorithm. This information together with the time attribute may be used to guide the overall synthesis process when looking for possible candidate nodes to apply transformations to. The time attribute is calculated from the length of the schedule of the operation graphs and is initially equal to the size of the operation graph. The execution time of the whole algorithm is, thus, the sum of the products of each nodes frequency and time. The global effect of a transformation on the TAO graphs may thus be calculated as the speedup gained.

The branch set contains both the possible follow nodes and their boolean expression attached to each arc. The boolean expressions are given as canonical product of min- or max terms. The expressions must be disjoint so that no ambiguous branching situations occur. This requirement is of no major problem when generating TAO graphs from programs written in traditional programming languages. Constructs with non-deterministic choice such as the Ada™ [28] select statement or Dijkstra's guarded command [2] may give other branching structures.

The schedule is an ordering of the operation graph nodes into sets such that the resource constraints and the partial
ordering of operations are not violated as discussed in the previous section. This set assumes that all operation may be performed within the same major clock cycle in the target sequential machine. The algorithm graph is defined as the following data structure:

**Data Structure Definition 3.1:** The algorithm graph is represented as an array of nodes each with the above fields:

- `array [node]`
- `operation(node): operation graph`
- `frequency(node): integer`
- `time(node): integer`
- `branch(node): collection of boolean expression and nodes pairs`
- `schedule: ordered collection of operation nodes`
- `boolean expression: canonical product of min or max terms`
- `precede(node) ⊆ N: (n ∈ N | node ∈ branch(n))` defines the set of preceding nodes
- `start ∈ N: defines the start node`
- `end ⊆ N: (n ∈ N | branch(n) = 0)` defines the set of possible terminal nodes

The schedule are initially empty. The frequency and time measurements allows calculation of both the critical sections and paths through the algorithm graph. These may then be used to guide an optimization algorithm on selection of locations for transformations. Typical optimizing transformations on this graph level look for ways of reducing the execution frequency and time of the nodes.

![Fig. 3.1: Algorithm Graph for "repeat A if c1 then B else C endif D until c2 E".](image)

One method of reducing the time factor for a node is by allocating hardware resource so that several of the operation in the operation graph may be scheduled in parallel. To gain higher levels of parallelism one must apply transformations which will increase the size of the operation graphs. Often this will effect the size of the controller as these transformations are based on distribution and duplication of nodes, e.g., sequential and conditional merge, and unrolling of iterations.

![Fig. 3.2: Distribution of D onto B and C from figure (3.1)](image)

To give an example of these transformation we will now consider distributing the node (3) onto its preceding nodes, (1) and (2) in figure 3.1. Doing this will affect the size of the operation graphs as graph D is now copied, and merged with graphs B and C. The gain is parallelism on the operation level to the cost of extra functional units to realize this parallelism and extra control steps. The grow of the controller, with regard to steps, is maximum the number of nodes in the distributed nodes operation graph. The branching area of the controller may also grow with the number of arcs.

![Fig. 3.3: Code movement from B to A giving A*B' and B'.](image)

Compilers for Very Large Instruction Width (VLIW) architectures use a method called trace scheduling which is applicable [4]. An example, consider trying to gain higher levels of parallelism along the critical path through node (0), (1), and (3) in figure 3.1. Now if node (1) and (2) define the same variables this section of the operation graph B may be moved forwards and merged with the operation graph A. This requires the variables are not used by the operation graph C, in which
case the values of the variables may become corrupted. Moving the section forward may give higher parallelism in the operation graph A and thus reduce the overall execution time along the critical path.

The above conditions for this type of code movement is of course very restricted but this method may be generalized towards the merging of the two conditional nodes, (1) and (2), with the node (0). This type of code movement will require the evaluation of the graphs A, B, and C in parallel according to some schedule and then the selection of right variable assignment. This method is fully described in [27].

The main motivation for these types of transformations is the low amount of functional parallelism available in programs written in traditional programming. Algorithm or Block level transformations are necessary to increase the level of parallelism. We will now continue to the next level of abstraction in TAO, the task level graph.

4. TASK GRAPH
To describe the behaviour of large systems tasks and modules are valuable abstractions. The task model allows partitioning of an overall algorithm into cooperating sequential processes. A number of modern programming and hardware description languages allow this possibility. Examples of such languages are VHDL [14] and AdaTM [28] which are very similar in syntax. As TAO is targeted for handling programming and hardware description languages that allow asynchronous concurrency this must be handled in an appropriate manner. In TAO this is achieved on a separate level of design representation as these system level concurrent activities are used to determine partitioning of the system into packages. To do so information is needed about the dynamic prolife of the interacting tasks, interaction frequency and message sizes, to derive which tasks should be realized in the same package, e.g., on the same chip. In general, interacting tasks with high band-width requirements should reside on the same chip.

Each task is viewed as a separate controller and data path which is time and clock independent from the other tasks in the system. The only dependency between tasks is interaction through communication channels. Two major communication styles may be used. The task level graph is described by the following basic attributes;

1. The algorithm performed by the task node
2. Communication channel set with frequency and size of interaction
3. The hardware resources allocated for the task

An algorithm graph describes the behaviour of the task. During the synthesis process the set of resources are allocated and the result is the data path. To further motivation the task level transformations similar to the merging of operation graphs may be performed on the algorithm graphs. This would mean that two or more task may be merged together to form one unit, i.e., use the same controller and data path. Two possible realization techniques are 1) time multiplexing of the tasks on the same data path, and 2) combining the two task descriptions. Time multiplexing is a common technique for concurrency on sequential machine and requires saving of the task state between activations, i.e., context switch. This can be efficiently realized in hardware as demonstrated by the INMOS TransputerTM [16]. Combination of two or more tasks will grow the algorithm graph exponentially but if the tasks use similar resource during different sections of their behaviour higher utilization of hardware resources and increased levels of functional parallelism can be achieved.

The interaction between task is performed on channels through ports. Each task node holds a set of channels through which it communicates with another task. Each channel holds information about the frequency and size of interaction through the channel.

The last attribute describing the task is the set of resources allocated for realizing the data path and controller. Typically this set is empty at the beginning of the synthesis process and appended during the process.

Data Structure Definition 4.1: The task graph is represented as an array of nodes each with the above fields;

array [node] [algorithm, channel, resource]

1. node \( \in \mathbb{N} \): integer in the range \( 0 \ldots \left| \mathcal{N} \right| -1 \)
2. algorithm(node): algorithm graph
3. channel(node):
   
   collection of node, frequency, size triples
4. resource(node): collection of resources

Each channel is further defined by data type and name as it is internally mapped to a variable on the lower levels. To the algorithm the sending and receiving of data through channels is viewed as assignments and thus equivalent to actions on variables. The port is responsible for the actual transportation of data between the interacting tasks. This can, naturally, be achieved in a number of fashions, e.g., bit-wise, bit-serial, with/without buffers, etc. Again, the choice of channel realization will effect the system level performance if bound by communication rate and not computation.

The send or receive operation on a channel will alter the algorithm graph when the communication style is synchronous as the first task to arrive at the interaction point will have to busy-wait for the neighbouring task. For asynchronous communication with buffers the sender may precede without waiting for an acknowledgement from the receiver.

5. CONCLUSIONS
The hierarchical graph oriented design representation TAO allows High Level Synthesis Systems to capture and separated two primary types of parallelism. The use of an hierarchy of graphs reduces many of the problem with time complexity of graph algorithm during the process of synthesis. In contrast to other design representation TAO tries to separate the
asynchronous level of parallelism from data and control flow. The asynchronous level is used for system partitioning and choice of communication realization. The control flow level block level transformations to increase the level of functional parallelism. And last, the data flow level flow to map towards a synthesized data path. This should be built during and is the result of the synthesis process. The operation level scheduling gives a partial data path structure. Registers and interconnections, e.g., buses, have yet to be modeled. Another question in this direction is the accuracy of the cost and time estimations that are made. So far this work has concentrated on minimizing the representation on each level.

ACKNOWLEDGMENTS
The author wishes to acknowledge the fruitful discussions and comments made by Göran Rydqvist, Krzysztof Kuchcinski, and Zebo Peng on the drafts of this paper.

REFERENCES