The Attributed-Behavior Abstraction and Synthesis Tools

Lawrence F. Arnstein and Don Thomas
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Though high-level synthesis tools seem to fit nicely into the traditional top-down VLSI design methodology in which an abstract algorithmic model is transformed into a detailed register transfer level implementation, there is an important difference between filling in the details by hand and relying on a high-level synthesis tool to do so. Unlike the hand-designer, an engineer who uses a synthesis tool is not likely to be familiar with the specific register transfer level implementation. Thus, the use of high-level synthesis tools can effectively inhibit the engineer's ability to attack design problems or make informed trade-offs at either the specification or implementation level. A solution to this problem that we have developed in this research is to expose register transfer level implementation detail to the engineer, for both analysis and modification, in terms of the original algorithmic specification of the system. We introduce a new design abstraction that can uniformly represent both the input and output of high level synthesis tools. When coupled with a new type of synthesis tool, the attributed-behavior abstraction can increase the potential for high-level design space exploration by making synthesis results accessible to the engineer.

1 Introduction

In [2] a synthesis tool called Marionet was introduced that can accept assertions about relationships between operations in a control-and-data-flow-graph that must be reflected in any register transfer level synthesized result. In order to accept these assertions, a general consistency maintenance technique based on minimality in binary constraint networks was implemented. Using the information generated during consistency checking, a new force-directed synthesis heuristic was described that uniformly accounts for the affect of assertions and of register transfer level design rules on the acceptable register transfer level design space. The motivation for developing Marionet was to increase the level of cooperation that is possible between an engineer and a high level synthesis system during design space exploration.

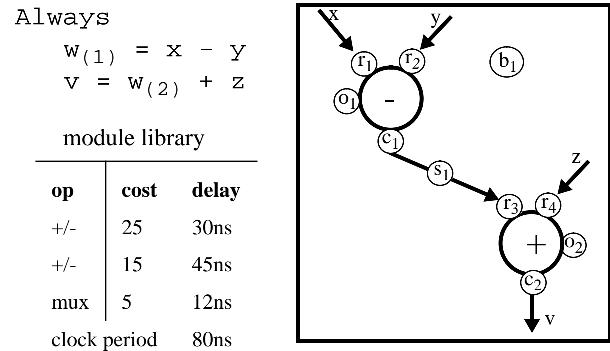
In this paper we formalize the goals of the Marionet project by defining a new high level design abstraction called attributed-behavior, and a synthesis based design methodology based on this new abstraction. In Sections 2 and 3, we define the attributed-behavior abstraction and give a simple example to show how it relates to the algorithmic and register transfer abstractions that are commonly regarded as the input and output of high-level synthesis tools. In Section 4 we present our vision of the attributed-behavior based design methodology in which the engineer develops a complete attributed-behavior model in cooperation with a general synthesis tool, and possibly a suite of specialized automated design assistants that can read and write attributed-behavior specifications.

2 Definitions

An attributed-behavior specification consists of a control-and-data-flow graph along with a set of relationships in the structural and temporal domain, called *path*, *cost* and *delay*, that can be imposed upon features of the graph. These relationships can either be provided by the engineer in the form of assertions that must be adhered to by the synthesis tool, or they can be filled in as a result of synthesis. In this way, the engineer and the synthesis tool effectively cooperate in the design of a complete attributed-behavior specification.

We have identified five features of a data-flow graph that can be involved in high level relationships. We call these features "semantic-attributes" because they correspond to the semantic content of lexical tokens from the original textual algorithmic description, with which the engineer is most familiar.

FIGURE 1. A Small Example




```
Always
    w(1) = x - y
    v = w(2) + z
```

module library

op	cost	delay
+/-	25	30ns
+/-	15	45ns
mux	5	12ns
clock period		80ns

The five types of semantic-attributes that can appear in an algorithmic model are indicated by the small bubbles in the data-flow graph of Figure 1; they are:

1. Value-reference: r_1 ***"ref(x)",*** r_2 ***"ref(y)",*** r_3 ***"ref(***$w_{(2)}$***)",*** r_4 ***"ref(z)"***

Value-reference semantic-attributes will eventually require resources (multiplexors, busses, or wires) in the register transfer level design for retrieving operands from storage or from other functional resources.

2. Operation: o_1 ***"op(-)",*** o_2 ***"op(+)"***

Operation semantic-attributes require functional resources in the data-path such as adders, and multi-function ALU's.

3. Value-creation: c_1 ***"create(***$w_{(1)}$***)",*** c_2 ***"create(v)"***

Value creation semantic-attributes require interconnection resources to deliver values to storage resources.

4. Value-storage: s_1 ***"store($w_{(1)}$)"***

Value-storage semantic-attributes may require registers or memories to maintain a value from the time it is created to the time that it is last consumed.

5. Block-entry: b_1 ***"block(always)"***

Block-entry semantic-attributes are similar to basic block entry points in sequential programming languages. Block-entry attributes are useful for expressing global temporal relationships. The code fragment of Figure 1 contains only a single block-entry attribute because it has no conditional control operations.

In our notation, the subscripted letters $\{r_i, o_i, c_i, s_i, b_i\}$ refer to semantic-attributes of the corresponding type described above, while a_i refers to a semantic attribute of an unspecified type. In general, there can be more than one semantic-attributed associated with a given lexical token. An attributed-behavior design contains an *algorithmic* and a *relational component*. The algorithmic component is a textual algorithmic description or a CDFG. The relational component is a set of expressions that describe *path, cost*, and *delay* relationships between semantic-attributes as defined below:

1. *Path*: Captures sequential delay relationships.

The function $path(a_i,a_j)$, evaluates to the difference in clock cycles between the start step (of the life-time) of attribute a_i and the start step of a_j. For instance, the statement "o_1 and o_2 are scheduled into successive control steps" is the same as the attributed-behavior expression [$path(o_1,o_2) = 1$]. The path function is defined only for static temporal relationships.

2. *Cost*: Captures resource cost and sharing relationships.

The *n*-ary function $cost(a_1, a_2, ..., a_n)$ evaluates to the total cost of the set of resources utilized by its arguments. If attributes a_i and a_j share a single resource, then the expression $cost(a_i,a_j)$ evaluates to the cost of just that resource. Otherwise, it evaluates to $cost(a_i)+cost(a_j)$. Cost relationships can also be seen as a way to describe a (partial or complete) partitioning of the semantic-attributes of a behavioral description. For example, the expression [$cost(a_i,a_j) > cost(a_i)$] implies that the two attributes do not share resources in the register transfer level design, whereas the expression [$cost(a_i,a_j)=cost(a_i)$] implies just the opposite, that the two operation attributes share a single resource. In the extreme, these partitions become isomorphic to named resources of a register transfer level model.

3. *Delay*: Captures combinational delay relationships.

The *delay()* function evaluates to the delay in continuous time (not clock cycles) from the start of the first semantic-attribute to the end of last semantic-attribute in the argument list. Both *path()* and *delay()* are defined only for static relationships, and only for one or two arguments. For example, $delay(r_1)$ evaluates to the delay of the resource (wire, bus, or multiplexor) that is used to transfer *x* to the input of the subtracter.

What makes the attributed-behavior language different from traditional algorithmic description languages for high level synthesis is that an engineer who writes an algorithmic description faces only a specification problem, whereas one who to writes an attributed-behavior description, even a partial one, faces a design problem as well. This difference is due to the fact that the writer of attributed-behavior model must consider design rules that govern combinations of path, cost, and delay relationships, while the writer of an algorithmic model is only confined by the syntax of the language in expressing the desired functionality.

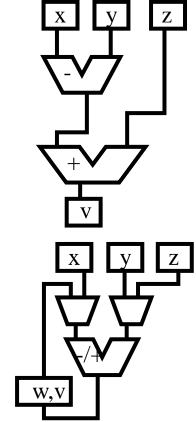
The design rules that govern combinations of relationships are similar to those that govern the synthesis of register transfer level models. For instance, the expression

$$[(cost(o_1,o_2)=cost(o_1)) \ \&\& \ (path(o_1,o_2) = 0)]$$

is inconsistent because operations that are scheduled in the same control step cannot share resources. To illustrate how attributed-behavior relationships can reflect register transfer level detail, we present two different sets of relationships and their corresponding register transfer level models for the small example of Figure 1.

Line 1 of Relational Component 1, in Figure 2 above, indicates that the always block is scheduled into a single clock cycle. This alone is sufficient to require two separate functional units for the add and subtract operations, as is reflected in the binary cost relationship of line 2. Expressions printed in light italicized text are redundant with respect to earlier bold-face expressions. Lines 3 and 4 are also redundant because, for a schedule length of one, the add and subtract operations must be scheduled into the same control step (Line 3). Furthermore, in the absence of functional unit sharing, no multiplexor delay is experienced in referencing values for the operations (Line 4).

FIGURE 2. Two Implementations



Relational Component 1

1. path(block(always)) = 1 step
2. *cost(op(-),op(+)) > cost(op(+))*
3. *path(op(-),op(+)) = 0*
4. *delay(ref(x)) = 0ns*
5. delay(op(-)=45ns
6. *cost(op(+)) < 35*
...

Relational Component 2

1. path(block(always)) = 2 steps
2. cost(op(-),op(+)) = cost(+)
3. cost(op(+)) = 15
4. *path(op(-),op(+)) = 1*
5. *delay(ref(x)) = 12ns*
...

Due to chaining from the subtract to the add operation, at least one fast/expensive functional unit must be purchased to satisfy the maximum clock cycle specified in Figure 1. Line 5 indicates that a slow resource is used for the subtract operation, so implicitly, a fast one must be allocated for the add operation as indicated redundantly by Line 6.

In contrast, Relational Component 2 specifies a two step schedule length for the basic block, so additional expressions are necessary to specify the resource sharing relationship between the two operations. In this case, the add and subtract operations share a single multi-functional unit, giving rise to increased value referencing delays and costs, but decreased resource costs. Thus, the trade-off between functional unit area, schedule length, and combinational critical path, that can be observed at the register transfer level, can also be observed at the attributed-behavior level in terms of the familiar textual algorithmic representation.

The key to the synthesis based attributed-behavior design process is to allow the engineer to specify only a partial set of relationships while the synthesis tool is relied upon to produce a full register transfer level model, which implicitly fills in the remaining relationships. Suppose for instance, that the engineer specifies a two-step schedule and provides the single assertion [*delay(ref(x))=0*]. Given a responsive synthesis tool, a new complete design will be found that has different cost and performance characteristics than the other two. One way to satisfy these partial requirements is shown in Figure 3.

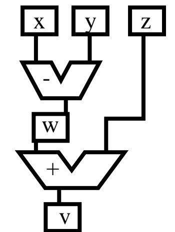
FIGURE 3. Synthesis from Partial Specifications

Engineer Provided

1. path(block(always)) = 1 step
2. delay(ref(x)) = 0ns

Tool Provided

3. cost(op(-),op(+)) > cost(op(+)
4. cost(op(+) = 15
5. cost(op(-) = 15



To satisfy the zero reference delay requirements expressed by the engineer, an additional functional unit is allocated to prevent the need for multiplexors. Though two functional units are allocated, this design is actually lower in cost than the first design because a fast/expensive functional unit is not required to meet the cycle length constraint. Fur-

thermore, it is faster than the second design because it has a shorter minimum clock period due to the reduced reference delay. This third interesting design point may have been inaccessible without the control provided by the attributed-behavior language (and responsive synthesis tool).

Note that the zero reference delay requirement can also be satisfied by re-using registers for intermediate values, thereby trading-off value-reference delay for value-creation delay. In any case, the synthesis tool is relied upon to complete the design so that all requirements established by the engineer are satisfied, if such a design exists.

The synthesis based attributed-behavior design process places a new demand on high-level synthesis tools: to be responsive to relationships provided by the engineer. In the next section, we re-cast the high-level synthesis task in term of the attributed-behavior abstraction, and compare Marionet, an attributed-behavior synthesis tool, with other existing high-level synthesis systems in terms of the degree to which they meet the new requirements of the synthesis based attributed-behavior design problem.

3 Tools for Attributed-Behavior Design

A synthesis tool that supports a meaningful attributed-behavior design process must be able to produce register transfer level designs that comply with a broad range of relationships from the general attributed-behavior language described above. The ideal tool would need to perform two main tasks with respect to an attributed-behavior specification: *verification* and *synthesis*.

The verification task involves analyzing the set of assertions to determine if they conflict with each other or imply the violation of any design rules as defined by the capabilities of a given synthesis tool. If so, then the specification itself is found to be inconsistent, if not then the specification is found to be consistent, ensuring that there exists at least one register transfer level design, and hence a complete attributed-behavior design, that satisfies all assertions.

The synthesis task is invoked if the specification is found to be consistent, but not complete. In performing the synthesis task, all of the *path*, *cost*, and *delay* relationships that remain explicitly and implicitly unspecified are filled in by the tool to produce a complete attributed-behavior design. These relationships can be abstracted form an actual synthesized register transfer level model, as long as it properly reflects all of the assertions provided by the engineer. Ideally, the tool will choose a complete design point that is optimal with respect to the design space that remains after the set of assertions has been accounted for.

Unfortunately, the verification task is NP-complete and the synthesis task is NP-hard for general attributed-behavior specifications, requiring us to weaken the ideal attributed-behavior synthesis tool for practical applications. By limiting the language of assertions that is accepted, it becomes possible to solve the verification problem in polynomial time, meaning that (1) that all inconsistent designs are detected and (2) that all consistent designs can be successfully completed. Any less ambitious goal, such as a tool that has the first property but not the second, would probably not be very useful. On the other hand, achieving optimality in the synthesis task, even for trivial attributed-behavior specifications (traditional high-level synthesis) is NP-hard, requiring us to accept less than optimal results even for a restricted assertion language. In summary, a practical attributed-behavior synthesis tool should perform the verification and synthesis tasks as re-defined below:

Verification: All inconsistent (restricted) attributed-behavior specifications are detected. For a given assertion language, the capacity to perform the verification task can be formally proven and disproven.

Synthesis: Consistent (restricted) attributed-behavior designs are correctly completed (synthesized) with quality that is comparable to existing state-of-the-art traditional synthesis, with respect to the design space that remains after accounting for assertions. Unlike the verification task, the synthesis capabilities of an attributed-behavior tool are difficult to measure, but results have shown that Marionet does perform well with and without assertions[4].

By these definitions, any traditional high level synthesis system can be seen as an attributed-behavior synthesis tool that accepts a restricted set of assertions. To illustrate this new way of viewing the high level synthesis task, we have categorized some existing synthesis tools based on classes of assertions for which at least the verification task has been formally proven. We then define a new, more general class of assertions for which we have developed the verification and synthesis techniques implemented in Marionet, which are fully described and formally proven in [3].

3.1 Notation

A class of assertions is defined by the types of expressions that are included. With one exception, all assertions consist of a path, cost, or delay function that is related to a constant by some relational operator. As an example, $C^n(\geq)$ denotes the set of n-ary cost-based clauses of the form $[cost(a_1,a_2,..,a_n) \geq X]$ where X is some constant numerical value. Similarly, $P^2(\leq,\geq,=)$ denotes the set of binary assertions of the form $[path(a_1,a_2)\{\leq,\geq,=\}X]$ where the relational operator can be member of the indicated set. The notation $D^1(\leq)$ describes the set of unary delay-based assertions of the form $[delay(a_1) \leq X]$.

In many cases, an engineer may wish to describe resource sharing relationships without unduly influencing module type selection. This is possible with two more types of cost-based assertions called *sharing* and *partitioning* assertions, denoted $C^{n^*}(=)$ and $C^{n^*}(>)$ respectively. The n-ary sharing assertion, which takes the form

$$[cost(a_1,a_2,a_3,...,a_n) = cost(a_1)]$$

defines a set of semantic-attributes that utilize a single register transfer level resource, without specifying the exact cost of the resource, and without excluding other sematic-attributes from using the resource. Similarly, the n-ary partitioning assertion takes the form

$$[cost(a_1,a_2,a_3,...,a_n) > cost(a_2,a_3,...,a_n)]$$

specifically requiring that a_1 not share resources with any of the other attributes in the argument list regardless of the specific type of resource used to implement them. Finally, the notation P^g and C^g describes the global schedule length and resource constraints that are common in many traditional high level synthesis systems. Thus, the maximal class of assertion that can be described in this notation is the set denoted *Class A*:

$$\{P^g,C^g,(P^{1,2}(\leq,\geq,\neq),C^{n^*}(=,>),C^n(\leq,\geq,\neq),D^{1,2}(\leq,\geq,\neq))_{\{o,r,c,s,b\}}$$

In some classes, a type of assertion can be applied only to a certain type of semantic-attribute. To denote such limitations, subscripts from the set $\{o,r,c,s,b\}$ are used to indicate the types of attributes to which the assertion can be applied. For instance, the notation $P^2(\geq)_{\{o\}}$ describes the set of binary path assertions that can be applied to pairs of operations. While interpretation of many expressions in Class A are unambiguous, such as $C^{2^*}(>)_{\{o\}}$, some expressions are open to interpretation. For instance, how should a $D^1(>)_{\{s\}}$ be interpreted? Perhaps it should apply to the propagation delay of the storage device. On the other-hand, maybe it makes more sense to regard it as a constraint on the life-time of the stored-value itself.

Furthermore, some expressions, though unambiguous, do not seem to be particularly useful. As an example, $D^1(>)_{\{r\}}$ assertions can be satisfied in two ways. One is to make sure that a multiplexor is used to reference the value. This could be a difficult request to satisfy, and furthermore, it is not clear that this capability would ever come in handy. In the development of Marionet, we have attempted to provide broad support for expressions that have relatively unambiguous and useful interpretations. In the next section, we define some classes of assertion languages that are supported by existing synthesis systems, and compare them to the class currently supported by Marionet.

3.2 Comparison to Existing Synthesis Systems

There are two basic categories into which all high level synthesis tools can be placed: those that support global resource constraints, C^g, and those that support global path constraints, P^g. Though some tools allow for the simultaneous expression of both P^g and C^g they can not guarantee successful verification in bounded polynomial time under such

conditions. On top of either P^g or C^g, many synthesis tools also accept $P^2(\geq)$ primarily for the purpose of specifying correct interface behavior with other processes or with the external environment. Also for the purpose of specifying interface behavior, some systems accept more sophisticated temporal assertions that are described by Classes F and G shown in Table 1.

TABLE 1. Some Existing Systems

Class	Description	Representative Systems
B	$\{P^g\}_{\{o\}}$	Sim. Ann. [6] SAM [5]
C	$\{C^g\}_{\{o\}}$	Sim. Ann. [6] List [][]
D	$\{P^g,P^2(\geq)\}_{\{o\}}$	HAL (FDS)[10]
E	$\{C^g,P^2(\geq)\}_{\{o\}}$	SAW (List) [12]
F	$\{C^g,P^2(\leq,\geq)\}_{\{o\}}$	Ariel[1] Hercules [7]
G	$\{C^g,D^2(\leq,\geq)\}_{\{o\}}$	Caddy [4]

Some of the tools referred to in the above table have temporal constraint handling capabilities or limitations that are not fully captured by the class of assertions described above. For example, A unique feature of the algorithm developed by Ku and DeMechelli [7] is that it accommodates operations that have unbounded delay; this is useful for modeling external synchronization primitives. Also, the support for unary delay constraints provided by Camposano and Kunzmann is limited to sets of constraints that have a hierarchical block structure. The set of assertions supported by Marionet (*Class H*) is shown below, but can be perhaps better understood by considering the comparison to *Class A* shown in Table 2.

$$\text{Class H: } \{\{P^1(\geq,\leq,\neq),P^2(\geq,\leq),C^1(\geq,\leq,\neq),C^{n*}(=,>),C^2(\geq),D^1(\geq,\leq,\neq)\}_{\{o\}},\\ \{D^1(\leq),C^1(\leq),C^{n*}(=)\}_{\{r,c\}}\}$$

Also for comparison purposes, Class G and F are also entered in the table. Marionet is weak in support of assertions that apply to stored-value attributes. There are two reasons for this: first, like most high-level synthesis tools, the synthesis tool upon which Marionet was developed, called SAM[7], emphasizes the scheduling and binding of operations and associated interconnect hardware, with secondary concern for register binding. For this reason, it would be more difficult to retro-fit the system to support value-storage based assertions. Second, register binding freedom gained by disallowing such assertions can be used by the algorithm to more optimistically bind and schedule operations and value transfers in response to supported assertions. These problems are addressed in greater detail in [3].

TABLE 2. Class H: Marionet

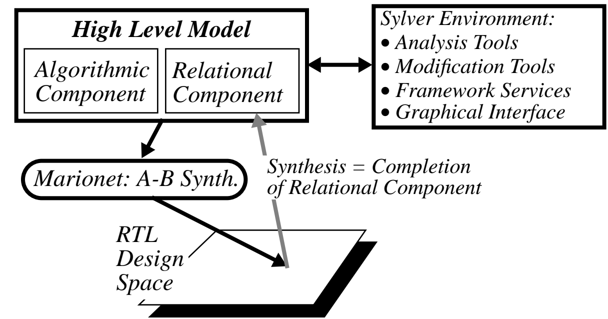
Table of Class A Assertions	✔ = supported, ❍ = not supported ✖ = not defined in this work				
	Attribute Type				
	o	***r***	***c***	***s***	***b***
P^g	✖	✖	✖	✖	✔
$P^{1,2}(\leq,\geq)$	✔,F	✔	✔	❍	✖
C^g	❍,G,F	❍	❍	❍	❍
$C^1(\leq),D^1(\leq)$	✔	✔	✔	❍	❍
$C^1(\geq,\neq),D^1(\geq,\neq)$	✔	❍	❍	❍	❍
$C^2(\geq)$	✔	❍	❍	❍	✖
$C^{n*}(=)$	✔	✔	✔	❍	✖
$C^{n*}(>)$	✔	✔	✔	❍	✖
$C^n(\leq,\geq,=)$	❍	❍	❍	❍	❍
$D^2(\leq,\geq,=)$	❍,G	❍	❍	❍	✖

Along with the syntactic restrictions imposed on the assertion language supported by Marionet, there are also some semantic restrictions that the model writer must be aware of. For example, a set of *n* operation attributes that are mutually related by an sharing assertions must have pair-wise non-overlapping initial time-frames for $n>2$. This rule prevents the simultaneous expression of P^g and C^g constraints. There are a few other semantic restrictions that are more detailed than space allows. However, in all cases, semantic violations can be detected in polynomial time prior to verification and synthesis. Adherence to the semantic restrictions does not guarantee that a correct register transfer level design exists, it only guarantees that Marionet will find such a design if one exists.

4 The Design Process

Figure 4 illustrates the cooperative design process that we envision. Design space exploration is accomplished by allowing the engineer to express local design requirements that must be satisfied by the synthesized register transfer level model. Essential to the attributed-behavior design process is a responsive synthesis tool that can perform the verification task for a broad class of assertions, and that can perform the synthesis task in the presence of assertions as defined in Section 3.

FIGURE 4. The Design Process



Because of the complexity and abstract nature of the relationships that are important in attributed-behavior design, simply having responsive algorithms is not sufficient to facilitate an effective design process at this level of abstraction. Along with synthesis algorithms, a design environment is needed that helps the engineer to understand and manipulate these abstract relationships. Sylver, an attributed-behavior design environment that has been developed specifically to address these needs, is based on a mouse driven user interface that allows the engineer to graphically superimpose attributed-behavior relationships (assertions and query results) on the original textual representation. Some of the main features of Sylver are highlighted below--a more thorough description can be found in [3].

1. ***Evaluation and Analysis Tools:*** Attributed-behavior design analysis involves the derivation and visualization of unspecified relationships that are implicit in a synthesized register transfer level design in response to queries provided by the engineer. Queries are similar to the attributed-behavior expressions that we have introduced for assertions, except that queries can contain free variables of numerical and semantic-attribute data types, and there are fewer restrictions on their construction. In the course of query evaluation, Sylver determines all of the bindings for the free variables that make the expression true based on synthesized register-transfer level details. Query results can then be graphically superimposed on the textual representation. Also included in the design analysis category are algorithms for design comparison, and a high-level simulation interface that is similar to a mouse driven source level debugger for software, but with finer granularity.
2. ***Modification Tools*:** Currently integrated into the environment are simple tools for the editing and observation of assertions. Additionally, specialized tools that can automatically perform such sub-tasks as hierarchy extraction[13], architectural partitioning[10], module selection, etc., can integrated into the Sylver environment simply by designing them to read and write assertions that reflect design decisions made automatically. The potential for the attributed-behavior

abstraction to support cooperation between the engineer and a suite of specialized automated design assistants is discussed in [4] and in greater detail in [3].

3. ***Framework Services*****:** Design space exploration in the attributed-behavior domain involves the generation of design alternatives by inheriting and modifying assertions from previous designs. Thus, we have identified some framework services that are important in managing the design process. These include design inheritance to facilitate the generation of new alternatives and design tree maintenance to facilitate backtracking and design comparison.

In [3] we present a thorough example to demonstrate the attributed-behavior design concepts implemented in Marionet/Sylver. We were able address a range of issues that arose in the design of a cell of a systolic array for nucleotide sequence comparison as described in [13]. Examples of design issues that we were able to address include pipeline synchronization problems, interconnect versus functional unit cost and delay trade-offs, and module selection based cost/performance trade-offs. In addition, we were also able to correct some idiosyncracies of the synthesis tool having to do with the scheduling and allocation of dummy functional units for extracting bit-ranges of values in the data-flow graph. To test the applicability of attributed-behavior based design concepts on a realistic design problem, the systolic array cell was synthesized using Marionet/Sylver with a special FPGA/FPIC based hardware architecture called the RASA Board as the target technology. We found that high-level assertions had a significant and reasonably predictable impact on delay and cost characteristics of final placed and routed designs [14].

5 Limitations and Future Work

One can imagine some of the useful ways in which the class of assertions can be extended beyond that which is supported by Marionet to give engineers even greater control over synthesis results. For instance, we currently offer no way for the engineer to directly influence the register-to-register storage and interconnection architecture. While the constraint network approach used in Marionet can be extended to allow assertions of this type, there are two problems. First, the mapping from the algorithmic representation to the life-times of values is more convoluted than it is for operations, so it will be more difficult for an engineer to properly express and understand these types of requirements. Second, the large number of values, homogeneity of resources that store them (registers) and long life-times all conspire to make large domains. Marionet, whose run-time is sensitive to domain size, is perhaps not ideally suited to dealing with the register binding problem.

Another capability that would greatly increase the expressiveness of the assertion language is to allow contingency relations. For example, the expression [*path(x,y)=path(a,b)*] is a 4-way relationship that cannot be directly represented in the binary constraint network used by Marionet. However, by further specifying that *path(x,y)* is subordinate to *path(a,b)* the order of scheduling decisions made by the synthesis tool can be influenced, which can be a powerful way to control synthesis results. Once *path(a,b)* is established, the assertion can be handled the same as any other binary path assertions.

The support of disjunction in the assertion language would also be useful, though problematic. For instance, one might like to specify that an operation shares resources with at least one of a set of other operations; this can be expressed by a disjunction of sharing assertions. Unfortunately, the existence of disjunction contributes significantly to computational complexity. A possible way to deal with a less structured class of assertions, such as one that includes *n*-ary relations or disjunction, would be to apply randomized approaches like simulated annealing that have been shown to be good at solving systems of constraints for which known polynomial algorithms do not exist.

In the Sylver environment only complete attributed-behavior specifications can be simulated or statically interrogated. Thus, to observe the effect of a new assertion, it is necessary to synthesize a complete design, thereby bringing to light only one of the many possibilities. It would be interesting, and probably useful, to allow the engineer to issue queries about potential relationships in a partial specification. For instance, one might like to observe the effect on the time-frame of some operation due to the addition of a single new assertion. This incremental approach to attributed-behavior design would encourage even greater interaction between the engineer and the tool.

Acknowledgment

This work was supported by and NSF Graduate Research Fellowship and by NSF Grant # MIP-9112930.

References

[1]Abramson, J.M., Birmingham, W.P., *Binding and Scheduling under External Timing Constraints: The ARIEL Synthesis System*, Technical Report Number CSE-TR-83-91, Computer Science and Engineering Division, Electrical Engineering and Computer Science Department, University of Michigan.

[2]Arnstein, L.F., Thomas, D.E., “A General Consistency Technique for Increasing the Controllability of High Level Synthesis Tools”, *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, California, November, 1993, pp. 741-744.

[3]Arnstein, L.F., *A High-Level Synthesis Based VLSI Design Methodology*, Ph.D. Thesis, Department of Electrical and Computer Engineer, Carnegie Mellon University, Pennsylvania, Dec. 1993.

[4]Arnstein, L.F., Thomas, D.E., “Applications of Attributed-Behavior Synthesis”, The Proceedings of the 7th IEEE/ACM International Symposium on High-Level Synthesis, Ontario, April 1994.

[5]Bergamaschi, R.A., Camposano, R., Payer, M., “Scheduling Under Resource Constraints and Module Assignment”, *INTEGRATION*, vol. 12, December 1991.

[6]Camposano, R., Kunzmann, A., “Considering Timing Constraints in Synthesis From a Behavioral Description”, *Proceedings of the International COnference on Computer Design*, November 1986.

[7]Cloutier, R., “Force Directed Scheduling Allocation and Mapping”, *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pp. 71-76, 1990.

[8]Devadas, S., Newton, A.R., “Algorithms for Hardware Allocation in Data-Path Synthesis”, *IEEE Transactions on Computer Aided Design*, Vol 8., No. 7, July 1989, pp. 768-779.

[9]Ku, D.C., De Michelli, G., Relative Scheduling under Timing Constraints: “Algorithms for High Level Synthesis of Digital Circuits”, *IEEE Transactions on Computer Aided Design*, Vol 11, No 6, June 1992, pp 696.

[10]Lagnese, E., *Architectural Partitioning for System Level Design of Integrated Circuits*, Ph.D. Thesis, Carnegie Mellon University, ECE Department, 1989.

[11]Paulin, P., *High Level Synthesis of Digital Circuits Using Global Scheduling and Binding Heuristics*, Ph.D. Thesis, Department of Electronics, Faculty of Engineering, Carleton University, January 1988.

[12]Powell, P., *Sequence Similarity Algorithms and a Linear Systolic Implementation*, Technical Report TR 89-44, Computer Science Department, Institute of Technology University of Minnesota, Minneapolis, Minnesota, 1989.

[13]Rao, D.S., Kurdahi, F.J., “Hierarchical Design Space Exploration for a Class of Digital Systems”, IEEE Transactions of Very Large Scale Integration Systems, Vol 1, No 3, September 1993, pp. 282-295.

[14]Schmit, H., Arnstein, L.F., Thomas, D., Lagnese, B., “Behavioral Synthesis for FPGA Based Computing”, *IEEE Workshop on FPGA’s for Custom Computing, April, 1994.*

[15]Thomas, D., *The System Architect’s Workbench User’s Guide*, Technical Report CMUCAD-91-42, Department of Electrical and Computer Engineering, Carnegie Mellon University, May 1991.