# Interface Constrained Processor Specification and Scheduling

Jack Greenbaum and Forrest Brewer
Dept. of Electrical and Computer Engineering
University of California, Santa Barbara, Ca. 93106

**Abstract**

*We propose a novel specification for tightly constrained processing elements based on recognizing concurrent sequences of data flow. This specification has several benefits, notably it allows a very concise representation of complex internal semantics and interface protocols. Particular advantages exist with specifying timing constraints and performance cost functions for control dominated applications. The specification can be made canonical and leads to an interesting formulation of the scheduling problem.*

## 1.0 Introduction

There are many systems which have complex constraint requirements as well as complicated behavior. Such systems are very difficult to correctly specify in a procedural language without introducing structure based on the syntax of the hardware description language (HDL), as opposed to the requirements. Example systems are microprocessors, memory and cache controllers, and other control-dominated components tied to computer communication busses. We propose a specification system at a higher level of abstraction which simplifies these problems.

### 1.1 Qualities of a "good" specification

There are several qualities that a specification must embody in order to be "good" from the many points of view within a synthesis system. The first view is strictly behavioral. Does the specification express the desired behavior? This can be checked by running the specification on sample data. Thus a second desirable attribute is that the specification be runnable. A good specification must also be efficiently analyzable. Syntactic specification aids for behavior and constraints must be added judiciously or else a seemingly simple specification may be inately ambiguous and require complex analysis to determine correct implementations. Although behavioral specifactions are eventually analyzed by machine, they must be readable and writeable by humans. The writeablity of the language lends itself to more efficient specifications. Designers are only human, and a tool should work for them instead of against them. An example of this is the use of high-level languages for software systems. Few people write large programs in assembly language, and indeed, few if any modern control dominated programs (parsers, scanners, etc.) are written using a procedural language.

Designs written in a procedural languages do not meet all of the goals of a good specification as defined in the previous section. Procedural specifications define not only data flow, but control flow too. This results in an overly constrained design because artifacts of the language show up as constraints on the flow of control in the design. The formalism we propose which contains the elements of a good specification without the pitfals of procedural languages is that of tail-recursive context-free grammars (CFG's). We call this formalism SL-PBS, for System Level Production Based Specification.

### 1.2 Related work

In general, prior work in specification for behavioral synthesis has revolved around a procedural specification of the behavior using languages similar to standard software structured programming languages. One example is Instruction Set Processor Specifications (ISPS) [Barb81]. This language is used by the DAA and SAW systems from Carnegie Mellon University [LaTh89]. Concurency of execution is supported in the language through explicit declaration. Standard structured programming flow of control operators such as IF, DECODE (case), and procedure call are provided.

Silage [Hilf85], developed at UC Berkeley, is geared toward specification of Digital Signal Processing, and has a syntax that supports this goal. In particular it supports iterative operations on time-bounded arrays of values. Silage is

0-8186-2610-0/92 $3.00 

the language used in the Cathedral synthesis system from IMEC [Note91].

Several commercial vendors provide tools that use VHDL [IEEE88] as input. The syntax is similar to the software languages Ada and Pascal, but allows structural as well as behavioral specification. Typically these tools define a subset of the language that is "synthesizable", and all other constructs are ignored. The EXEL language presented in [DuGa89] presents similar facilities for expression of structure and behavior, but is geared toward interactive synthesis.

The expression of timing constraints for synthesis in the context of behavioral specification has been addressed using procedural languages. Nestor [NeTh86] adds timing points between labeled operations in an ISPS description. [CaKu86] use timing constraints attached to a group of one or more operations in the procedural language DSL. EXEL [DuGa89] provides constructs for data flow timing constraints for data-dependent values as well as absolute timing information for non-data-dependent signals. Most similar to the model presented here is the BFSM representation described in [LeTaWo91]. In this representation pairwise timing constraints are specified for input and output events that appear on the interfaces of an finite state machine. The machine is derived from a VHDL specification.

Other systems allows specification of timing information with regard to simulation. VHDL [IEEE88] in particular has a robust timing simulation model. The OEgraph representation of [AmBo91] is geared toward specification of timing behavior as well as the checking of the behavior against a set of constraints. These constraints are expressed in a first-order calculus, and checked incrementally during simulation. The design is described as an OEgraph, which is an event driven simulation.

Regular expressions and context free grammars (CFG's) have been used for specification and verification. In [FIUl82] regular expressions are used to specify an NFA which is implemented as a PLA. The machines recognize input and produce output. However the output is only at the signal transition level, as opposed to high level behavioral actions. Seawright [SeBr91] presents PBS (Production Based Specification). A PBS description contains a set of hierarchical productions annotated with behavioral VHDL actions. The productions are compiled into a state-minimal FSM implemented in procedural VHDL tailored for hardware synthesis. The actions are executed on transitions of the machine implicitly specified by the productions.

CFG's have also been used to verify that a design is in the space of allowable constructs. The GRASP system of Bamji [BaAl89] uses CFG's to represent the design space of a particular circuit technology. Any legal design in that technology must be contained within the grammar.

### 1.3 Modeling behavior as language translation

This name SL-PBS expresses the dependence on and relationship with the existing PBS. SL-PBS passes a scheduled grammar to PBS. The scheduling takes into account system level trade-offs between control and allocation. The SL-PBS grammar for a design specifies a translation between the input signals (input language) and the output signals (output language) on the interfaces. This is analogous to a software compiler that translates a programming language, e.g. C, into assembly language [AhSeUl86].

Figure 1 shows the relation of this model to a typical synchronous digital system. The token recognizer is a combinational circuit which produces a symbolic token based on the state of the interface signals, data path registers, and controller state. This sequence, or stream, of tokens comprises the input language which is recognized by the controller. The output language of the controller is a sequence of tokens where each token is a vector of output interface signals.

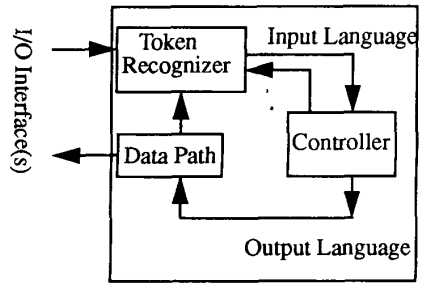



**FIGURE 1. Behavior as Translation**

This formalism gives us the power to verify timing and semantic qualities of the design at many levels of hierarchy. Since the input to the system is described as a language, we can check the specification for correctness by verifying that the specified input language is accepted by the design. We can also verify a state-machine implementation by comparing the implementation with the FSM specified by the grammar. The output of the design is also a language which can be verified. Timing constraints may be specified between any pair of tokens, or sequences of tokens. By examining the controller's states traversed between the two tokens we can predict whether or not the constraints can be met. If probabilistic methods are used, we may calculate the probability that the constraint will or will not be met.

CFG's may be represented in a canonical representation known as Chomsky Normal Form (CNF) [HoUl79]. Therefore, any two grammars that represent the same control se-

quences can be determined to be the identical. The difference between any two grammars for the same language are the hierarchy of the productions used to express the translation. This hierarchy is chosen by the designer, and should reflect the logical grouping of functions as seen by the designer. Using this hierarchy as a point of reference, we may specify timing constraints that are meaningful at a hierarchical level, not just at the interface. In this way the designer can provide constraints that are important in a abstract sense that relates to the view of the system under design.

## 2.0 Specification using SL-PBS

An SL-PBS specification consists of a tail recursive context free grammar with actions. This describes the machine's behavior in terms of streams of tokens and behavioral actions. A subset of the language described by this grammar is distinguished as the interface grammar, which describes legal sequences of interface tokens.

### 2.1 Context free grammars

Context Free Grammars define a class of sequential machines referred to as Push Down Automata, or PDA's [HoUl79]. By allowing only tail recursion a push-down stack is no longer required, and the machine defined becomes an NFA recognizing the class of regular languages.

The elements of the specification are called *productions*. A production has a head, which defines a non-terminal symbol, followed by a "production operator", and then a regular expression over the alphabet of terminals and non-terminal symbols, where terminals are the sequential atomic states of the external interface and state variables of the system being specified. A non-terminal appearing in this expression is interpreted hierarchically. A production can be flattened into a regular expression over the alphabet of the input tokens by recursively substituting the regular expression from the right-hand side of the non-terminals production declaration.

Each production can have actions associated with it. The actions are procedural code with no flow of control operations. All control flow is expressed within the grammar. Currently any VHDL sequential statement is legal within an action.

The recognition of the language defined by this grammar, the input language, proceeds as follows:

1. Determine next tokens from current states and the external interface signals.
2. Perform any action(s) associated with the transitions to be taken.
3. Determine next set of active states.
4. If in a final state, return. Otherwise go to step 1.

Figure 2 presents the CFG for the familiar greatest common denominator (GCD) benchmark. The `top-level` production waits for a `reset` token, latches the inputs, then begins the GCD inner loop. The inner loop production `gcd` recurses until the `x_equal_y` token is produced as a result of the actions in the tail-recursive clauses. At this point the production is recognized, the answer is latched to the output, and the `top-level` production recurses to wait for new input. Actions in an SL-PBS description are understood to execute under the rules of behavior that apply to the VHDL environment under which they are run, as defined by PBS. See [SeBr91] for details of the VHDL model.

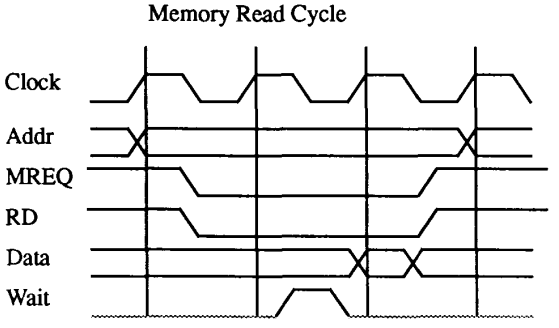
```
x_less_than_y { x < y }
x_greater_than_y { x > y }
x_equal_y { x = y }
::
top_level-> reset {
        x := Xin;
        y := Yin;
    } gcd top_level;
gcd ->
 x_less_than_y {
    y := y - x;
 } gcd |
 x_greater_than_y {
    x := x - y;
 } gcd |
 x_equal_y {
    GCD <= x;
 };
```

**FIGURE 2. CFG and tokens for the GCD problem**

The behavior of the external interface is described by a subset of the language accepted by an SL-PBS grammar. This subset is referred to as the interface grammar. It expresses the semantic and timing requirements required to transfer operands to and from the external world. The interface grammar, which may be partially ordered, provides a mechanism for automatically propagating timing constraints into the design, as well as verifying that the design meets the requirements for correctly communicating with its interface. Within the complete specification the details of interface communication may become buried. By identifying the essential interface explicitly we provide a means for insuring that the ensemble design communicates correctly with external data sources and sinks.

Figure 3 shows a timing diagram for a simple microprocessor memory interface read cycle. Below the timing diagram is a regular expression which describes the sequence of tokens that a grammar will read as input and produce as output while performing a read. The notation `token+token` is the concatenation of two tokens. The tokens `data`

and `wait` are inputs to the grammar, while all others are outputs. Timing constraints are specified as pair-wise time intervals between tokens.




Regular Expression Format:

addr mreq+rd (wait not_wait)? data
not_mreq+not_rd+not_data not_addr

**FIGURE 3. Timing Diagram as a Regular Expression**

```
addr mreq   1 clock
addr rd     1 clock
rd data     3 clock
rd wait     2 clocks
addr not_addr  6 clocks
rd not_rd      4 clocks
mreq not_mreq  4 clocks
wait not_wait  1 clock
```

**FIGURE 4. Pair-wise timing constraints**

In Figure 3 all tokens are totally ordered by the regular expression denoting the sequence of interface tokens. However robust designs will have partially ordered interfaces, for example using two memory interfaces which may insert different numbers of wait states.

Figure 5 shows a set of productions with partially ordered tokens. The notation `A&B` denotes that productions `A` and `B` are partially ordered with respect to each other. This means that `foo` may come before `bar`, or the other way around, or they may arrive simultaneously, followed by a single `bar`. The valid sequences are not always equivalent to the cross product of the partially ordered productions because pair-wise timing constraints may be specified between tokens across the productions.

## 2.2 Grammatic hierarchy, differences from software model

The SL-PBS form of machine specification looks quite similar to the production systems used as input to "compiler compilers" which generate software translators (i.e.

```
S -> A&B;
A -> foo bar;
B -> baz bar;
```

Accepts sequences:

```
foo baz bar
baz foo bar
foo+bar baz
```

**FIGURE 5. Partially ordered productions**

YACC [Jo75]). However our specifications are for a smaller class of machines, NFA's, not general push-down automata (PDA), and our model of execution is different. The first difference in execution is that our machines have no stack, thus several levels of syntactic hierarchy can be traversed by a transition on a single token without intermediary state transitions required by push-down machines.

This effect can be seen by looking at a partial parse tree of the example grammar of Figure 6. A state in our machine is described by a path from the root to a leaf node. The gray arrows in Figure 7 point to the lower levels of the grammar, eventually reaching a token (terminal symbol). The tree is traversed from left to right along the leaves.

```
run -> i-read instruction;

instruction ->
    load_instruction |
    store_instruction |
    arithmetic_instruction |
    flow_instruction;

aritmetic_instruction ->
    add {
        regFile(destReg(IR)) <=
            regFile(aReg(IR)) + regfile(bReg(IR))
    } |
    subtract {
        regFile(destReg(IR)) <=
            regFile(aReg(IR)) - regfile(bReg(IR))
    } | ...
```

**FIGURE 6. Using Grammatic Hierarchy**

Each successive token therefore produces another path

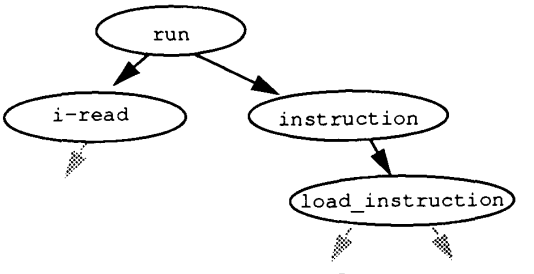



**FIGURE 7. A Parse Tree**

through the grammar no matter how many levels of hierar-

chy are introduced. So when the final token of the `i-read` production has been seen, the next state of the machine is described by a path from the top node down through the production hierarchy to a terminal symbol in the parse tree subgraph for the `instruction` production. This behavior is different from the software model of execution in that two separate state transitions (a reduction and a shift) would occur for each level of production hierarchy traversed.

What this model of execution shows is that hierarchy in the grammar does not translate into hierarchy in the control flow of the model. This is important when a specification is used as input to a scheduling system because of the added complexity required to remove control points that are not required by the behavior, but which often show up as control (in the form of procedure calls or if statements) in procedural HDLs.

Another difference between this model and that used in software recognizers in context dependence. Software translators that construct PDA's use CFG's as their input specification. However the hardware systems we endeavor to specify are context dependent in that some tokens depend on internal state as well as external signals. Thus the next token produced must be a function of the interface and internal state. These tokens are mapped to valid transitions only from certain machine states where these tokens are expected. For example the program counter might be considered when dispatching out of one state, while the condition flags of an ALU would be checked in another. These tokens need not be mutually exclusive except within the context in which they are used. Boolean expressions for tokens are specified by the user. The system then analyses the grammar, and produces the state-dependent token function. Currently the tokens valid for any particular state are not checked for mutual exclusion, although they must be for the model to execute correctly.

## 3.0 Verification

Given an interface subset and a grammar for a design, the SL-PBS representation allows verification that the design recognizes the input language, and that it produces the output grammar. This is *syntactic verification*, that is we check to see that appropriate data signals are read from and written to the interface at appropriate times. No information is available as to the semantic correctness. The notion of syntactic verification is distinct from the type of verification usually discussed in relation to state machines. [GhDeTh90], [DeKe90], and [LeTaWo91] all discuss verification that particular implementation is identical to the specification. What we are testing with syntactic verification is whether or not a design correctly drives it's external interface. The success of this process says that any schedule and allocation that satisfies the requirements of the actions and that meets the timing constraints will accept valid input sequences and produce syntactically valid output.

In this section we will use the SL-PBS specification to formally define three machines used by the verification algorithms. The first is the DFA which is constructed from the SL-PBS grammar with procedural actions and context dependent tokens. We will call this the *translator machine*. The second is a subset of the translator machine that only depends on the external inputs and has no actions, the *input machine*. The third is the input machine annotated with only the actions that produce external output, the *input-output machine*. Each machine is defined as a 4-tuple $M(\sigma,\alpha,\tau,\delta)$, where $\sigma$ is a set of states, $\alpha$ is an alphabet of tokens, $\tau$ is a set of transitions from state to state, and $\delta$ is a mapping over the domain of states and tokens to $\tau$, the transitions. We denote the translator machine $M_t(\sigma,\alpha,\tau,\delta)$.

The input machine is denoted $M_i(\sigma_i,\alpha_i,\tau_i,\delta_i)$. The tokens for the input machine, $\alpha_i$, is a subset of $\alpha$ such that each element of $\alpha_i$ depends on at least one external input. Each transition $\tau_i$ in $\delta_i$ corresponds to one or more transitions $\tau$ in $\delta$. The set $\tau_i$ is found by calculating the states reachable from any state by a single input token of $\alpha_i$, treating transitions due to internally dependent tokens as $\varepsilon$-transitions as in an NFA. The set of states $\sigma_i$ for the input machine is a subset of $\sigma$ such that at least one transition exists for the state in $\delta_i(\sigma,\alpha_i)$, and thus $\tau_i$. The output machine is a superset of the input machine. It is constructed by annotating each $\tau_i$ from the input machine with a regular expression machine defining the possible outputs produced during that transition. This machine for this regular expression is nothing more than the nodes and transitions in the translator machine that are represented by each transition in the input machine. Only actions that write to output signals are included.

Input verification, that is, verifying that the translator grammar specifies a machine that accepts the external input, is performed by constructing the DFA representing the interface subset grammar over the input tokens, and then testing the input machine described above for equivalence. If the input machine is equivalent to the interface input language then it is proven that the translator machine will accept this interface input, given the restriction that all context-dependent token sequences lead to valid transitions. This complexity of this test is $O(|\sigma_i|+|\tau_i|)$ since each state and each transition is checked for a correspondence within the machine under comparison. One should observe however that the number of states may be prohibitively large since the number of states in the input machine may

grow exponentially with respect to the number productions. This is particularly a problem when partially ordered productions exist in the grammar.

Output verification, that is, verifying that the actions attached to grammar productions produce syntactically correct output, is performed by constructing a machine from the interface grammar which for each transition taken from input alphabet is annotated with a DFA describing the legal sequences of tokens in the output alphabet. This machine is then compared for equivalence with the output machine described above. The equivalence is checked both ways insuring that the output machine produces the interface output, and only the interface output. Again, the time complexity is linear in the number of states and transitions, but can grow exponentially with the number of productions.

This provides *syntactic verification*, that is that the translator machine accepts syntactically valid input as defined in the interface specification, and that the output from the translator occurs during valid times in relation to the input, and in a valid sequence. It says little about the semantic correctness of the interface signals, which is a much harder problem. For general context dependent machines this problem is a halting problem, and is formally undecidable.

## 4.0 Posing the scheduling problem

Most work in scheduling does not deal directly with issues of control trade-offs or control dominated circuits. The Cathedral system achieves good results for algorithms in the domain of DSP, but the applicability of their approach to control dominated designs has not been reported on. The use of integer linear programming [GeEl91] for scheduling has focused on the standard computation intensive applications as well. However our experience with the Chippie silicon compiler system [BrGa90] has shown that the ability to consider control as well as data operations while scheduling is crucial to achieving a design that is globally efficient. The SL-PBS approach endeavors to solve this problem by making trade-offs between control and data flow visible, specifying timing constraints hierarchically, and providing a framework over which a designer specifies operator mobility constrained by the interfaces of the design.

### 4.1 Control/data trade-offs

To analyze timing information to evaluate trade-offs we construct an event driven data flow model, which we will call simply event graphs. This data structure is similar to the OEgraphs of [AmBo91]. Our goal however is to only express events as abstract entities. An event graph is constructed for each production. It consists of nodes, which are data operators, and data flow arcs. Operators wait for an event to occur on all inputs, then produce event on one or more outputs. Figure 8 shows the event graph for the `gcd` production in the example of Figure 2. The inputs to the graph are the values of the data items `x` and `y`. The operators wait for all operands to arrive, then produce output. A comparator produces either an event signaling `x = y`, which assigns the current value of `x` to the interface signal `GCD`, or an event signaling which subtracter value to propagate, `x` or `y`. When constructing the event graph, the nodes for the actions are constructed first. Then nodes which derive the tokens are added next. The token nodes control the outgoing data flow that results from the actions.

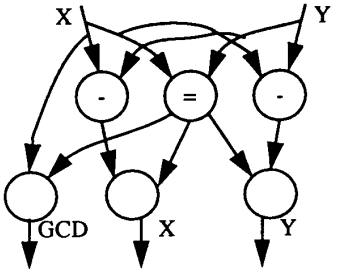



**FIGURE 8. An Event Graph**

The event graph for an entire grammar is constructed by expressing lower level productions as operator nodes within the event graph of productions further up the hierarchy. Figure 9 shows the top-level event graph for the GCD example. This graph is a hierarchical representation of the data flow from which we can calculate timing properties at an abstract level. The factoring of the design is provided by the designer. So the points from which we can examine timing are exactly those points which the designer feels are important. We believe this to be an advantage when scheduling because constraints can be specified over a meaningful hierarchy of the design.

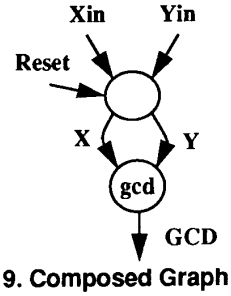



**FIGURE 9. Composed Graph for GCD**

Notice that these are acyclic graphs. Our point of view is that cycles imply control. A standard data flow graph with cycles must use fork or join nodes, which imply control signals that are not part of the specification. Thus the un-

scheduled event graph for tail-recursive productions is actually an infinite graph. For each X and Y output from the event graph of Figure 8, another tree of productions is created in Figure 10. No control is needed because events will be propagated down only one outgoing edge. It is only when one wishes to constrain resources that control is used, via back arcs, in the event graph. Thus the node marked `gcd` in Figure 9 is the infinite graph of Figure 10

If the bit-width of operators is restricted then this graph may be bounded. By examining the operations performed in the event graph of Figure 8 we can characterized the symbolic values `x` and `y` as monotonically decreasing. Thus we can determine that the depth of the tree in Figure 10 is limited by the magnitude of the data values. Of course this analysis can only be performed on those event graphs with whose iteration bound can be calculated.

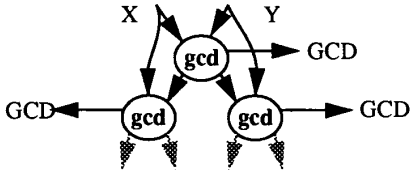
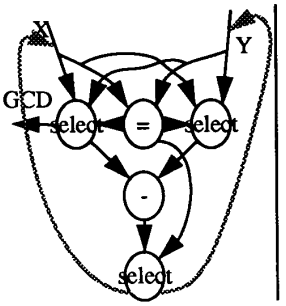



**FIGURE 10. Inifite graph**

The event graph is a directed acyclic graph that reflects the operations required for both control and data flow at a particular level of hierarchy. Given this graph we wish to create a possibly cyclic graph which meets constraints set by the designer. Constraints are examined on a production-by-production manner. Thus the constraints are specified at points in the design factoring that is provided by the designer. The structure of the grammar allows assemblage of low-level constraints on the interface into metrics for evaluation of these constraints.

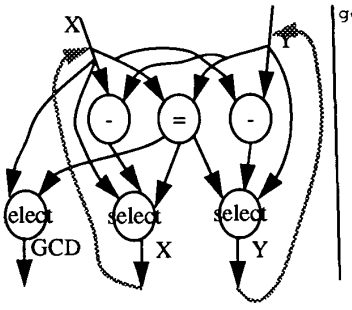
Figures 11 and 12 show two possible resource constrained cyclic implementations of the event graph for the `gcd` production. Figure 12 uses two subtracters, and Figure 11 uses one. Back arcs (shown in gray) have been added to allow reuse of the subtracters in both cases. The grammar on the right of each picture is input to PBS [SeBr91] derived from the scheduled event graph on the left. PBS compiles the grammar into a procedural model expressed in synthesizable VHDL. The VHDL synthesis tool allocates registers where necessary to preserve signals between clock cycles.

These two implementations represent a trade-off of control vs. computation. In the one-adder version there are more assignment statements and a sequencing state added to the machine in order to save one adder. We may directly evaluate the effect on speed of computation by evaluating the time delay through each operator along the cycles. In the two subtracter version the cycles go through a comparator or subtracter, then a selector. In the one subtracter version the cycles must go through a comparator *and* subtracter, as well as a selector. Thus the one subtracter version runs slower over the same technology.




```
gcd ->
    x_less_than_y {
        a <= y; b <= :
    } subtract {
        y <= out;
    } gcd |
    x_greater_than_y {
        a <= x; b <= y
    } subtract {
        x <= out;
    } gcd |
    x_equal_y {
        GCD <= x;
    };
```

**FIGURE 11. Single subtracter GCD**




```
gcd ->
    x_less_than_y {
        x <= sub1;
    } gcd |
    x_greater_than_y
        y <= sub2;
    } gcd |
    x_equal_y {
        GCD <= x;
    };
```

**FIGURE 12. Two subtracter GCD**

Not only is the critical path longer, but an extra sequencing state was necessary in the controller as result of reusing the single subtracter. Control has been added to allow for a smaller operand allocation.

We can now perform verification on the scheduled machine to ensure that the control structure generated by the scheduling process is still correct for the given interface grammar.

### 4.2 Operator Mobility

The previous section showed trade-offs manifested in allocation. The minimum allocation is that which covers all operand types and all operands that must execute simultaneously. We may influence this minimum allocation by rescheduling operators to execute at some time other that specified while maintaining semantic correctness. The term "operator mobility" is used in the literature on scheduling [PaGa87] to refer to the range of control states during which an operator in the event graph can execute. In the

point of view proposed with SL-PBS, operator mobility refers to the set of states of a DFA which recognizes the input language during which an operator may be scheduled. This may not be equivalent to the control state space since individual tokens may be recognized only after many control steps.

The anchors [KuDe90] of operator mobility in SL-PBS are the tokens of the input language. In this way constraints specified on pairs of input tokens are automatically propagated into the structure. The mobility rules are:

1. Any read or write to an external signal is locked down.
2. Any assignment to a signal involved in a subsequent token derivation in the original specification must occur before the same token derivation in the rescheduled version.
3. References are totally ordered with respect to assignments to the same signal, and are partially order among themselves. Assignments are totally ordered with respect to references to the same signal, and partially ordered among themselves.

Rule 1 ensures that external interface specification is met in terms of timing. Rule 2 ensures that the context of the token derivation is preserved. The constraints on interface tokens appear as manifestations of rule 1 in that all computation that is required by rule 2 to occur in grammar positions between the token pair must be completed in the interface constraint's specified time period. Rule 3 ensures the semantics of the actions are not violated by write-after-read errors.

## 5.0 Conclusion

We have presented a language translation model of specification for synthesis. By using grammars we allow hierarchy in the specification that is not expressed in the control flow. Specifying the interface as a grammar allows for syntactic verification of the specification as well as the scheduled design. The hierarchy of productions provides a factoring of the design from which to evaluate constraints. Our future work involves implementing a scheduling algorithm for this model which takes advantage of control/data trade-offs, operator mobility, and hierarchical constraints. The expression of applicable constraints must be further developed in order to meet this goal. As well the interface to the underlying tools (PBS, procedural synthesis) must be more well defined.

## 6.0 References


[AhSeUl86] Aho, A., Sethi, R., Ullman, J. "Compiler, Principles, Techniques, and Tools", Addison Wesley, Reading Mass., 1986.

[AmBo91] Amon, T., Borriello, G. "OEsim: A simulator for Timing Behavior", Proc. of the 28th ACM/IEEE Design Automation Conference, 1991.

[BaAl89] Bamji, C., Allen, J. "GRASP: A Grammar-based Schematic Parser", Proc. of the 26th ACM/IEEE Design Automation Conference, 1989.

[Barb81] Barbacci, M. "Instruction Set Processor Specifications (ISPS): The Notation and Its Applications", IEEE Transactions on Computers, V C-30, NO. 1, Jan 1981.

[BrGa90] Brewer, F., Gajski, D. "Chippie: A System for Constraint Driven Behavioral Synthesis", IEEE Transactions on Computer Aided Design, V9N7, July 1990

[CaKu86] Camposano, R. and Kunzmann, A. "Considering TIming Constarins in Synthesis from a Behavioral Description" Proc. of the 23 IEEE Design Automation Conference, 1986, pp. 6-9.

[DeKe90] Devadas, S., Keutzer, K. "An Automata-Theoretic Approach to Behavioral Equivalence", Proc. of the IEEE ICCAD 1990.

[DuGa89] Dutt, N. D., Gajski, D. "EXEL: A Language for Interactive Behavioral Synthesis", Computer Hardware Description Languages and their Applications, North-Holland, New York, 1990.

[GeEl91] Gebotys, C., Elmasry, M. I., "Simulaneous Scheduling and Allocation for Cost Constrained Optimal Architectural Synthesis", Proc. of the 28th ACM/IEEE Design Automation Conference, 1991.

[GhDeTh90] Ghosh, A., Devadas, S., Newton, A.R., "Verification of Interacting Sequential Circuits", Proc. of the 27th ACM/IEEE Design Automation Conference, 1990.

[Hilf85] Hilfinger, P., "A high-level language and silicon compiler for digital signal processing", IEEE Custom Integrated Circuits Conf., Portland, May 1985.

[HoUl79] Hopcroft, J., Ullman, J., "Introduction to Automata Theory, Languages, and Computation", Addison-Wesley, Reading, Mass., 1979.

[IEEE88] IEEE Standard VHDL Language Reference Manual, IEEE New York, 1988.

[Jo75] Johnson, S. C. "Yacc: Yet another Compiler Compiler", Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J. 1975.

[LaTh89] Lagnese, E. D., Thomas, D. E. "Architectural Partitioning for System Level Design", Proc. of the 26th ACM/IEEE Design Automation Conference, 1989

[LeTaWo91] Leeser, M., Takach,A.R., Wolf, W. "Behavior FSMs for High-Level Synthesis and Verification", Computer Engineering Technical REport No. CE-W91-13, Dept of Electrical Engineering, Princeton University, 1991.

[Note91] Note, S., Geurts, W., Catthoor, F., De Man, H., "Cathedral-III: Architecture-Driven High-level Synthesis for High Throughput DSP Applications", Proc. of the 28th ACM/IEEE Design Automation Conference, 1991

[NeTh86] Nestor, J. A. and Thomas, D. E., "Behavioral Synthesis with Interfaces", Proc. of IEEE ICCAD, 1986.

[PaGa87] Pangrle, B., Gajski, D. "Slicer: A state synthesizer for intelligent silicon compilation", Proc. of the IEEE ICCD, 1987

[SeBr91] Seawright, A., Brewer, F. "Production Based Hardware Specification and Synthesis", ECE Technical Report #91-20, University of California, Santa Barbara, 1991.