Constraint Propagation in an Object-Oriented IC Design Environment

Tai A. Ly and Emil F. Girczyc

Dept. of Electrical Engineering,
University of Alberta,
Edmonton, Alberta, Canada,
T6G 2G7

ABSTRACT

This paper describes a framework of constraint propagation that supports the least-commitment strategy of design. This is implemented in an object-oriented IC design environment. The resulting system incrementally propagates design characteristics whenever they become available, and checks for design specification violations. By managing the interaction among bottom-up characteristics and top-down specifications, constraint propagation can be a powerful aid in IC design.

I Introduction

In IC designs, hierarchical design techniques are widely used to partition a design task into several smaller design tasks, each of which is similarly partitioned in turn. This reduces the complexity of the design but limits interactions among different sub-circuits in the design. In a purely top-down approach, designers start with initial cell specifications, successively decompose the cell into smaller cells and define their interfaces with additional specifications, until the leaf cells of the design hierarchy consists of simple components which can be either designed directly or implemented using library cells. In a purely bottom-up approach, small cells are successively combined into larger cells, whose characteristics are functions of their component cells and interconnects, until a cell which satisfies all design specifications is obtained. Top-down design offers flexibility in high-level design decisions, which is advantageous for design space exploration. However, the impacts of a design decision are often poorly evaluated because low-level considerations are ignored until much later. This results in design iterations, in which low-level design characteristics are used to correct high-level decisions, generating new specifications that often require major redesigns. A better design process would use low-level characteristics much earlier in the design cycle.

Mixing top-down specifications and bottom-up design uses low-level characteristics early in the design cycle to guide high-level design decisions. The critical parts of a cell are first designed in considerable detail, which then guide the refinement of design specifications for the rest of the cell. This has been referred to as the "least-commitment" strategy of design [1] [2], in which design decisions are deferred for as long as possible.

For example, if the critical delay path of a cell has two subcells, top-down design would assign estimated delay specifications to each of the two subcells such that their sum satisfies the overall delay specification. The two subcells are then designed independently. The fact that delay of one subcell may be less than its specified delay is not used to relax the delay specification for the second subcell until design backtracking (when some specifications cannot be met). On the other hand, the "least commitment" strategy of design would only require that the sum of delays of these subcells satisfy the overall delay specification. The delay specification of each subcell is not "committed" until the characteristic delay of the other subcell is known. The bottom-up characteristics of a subcell are used, as soon as they become available, to refine (implicit) specifications of other, related subcells.

However, this leads to complex design interactions. One approach to managing these interactions is to declare explicit design constraints among cells, and use propagation of constraints to resolve design interactions. This paper reports on an object-oriented, hierarchical framework of constraint propagation, implemented in STEM (SmallTalk Environment for Module design) [Girc87], that is designed to manage arbitrary design interactions, and support a least-commitment strategy in IC designs.

II Related Work

Many design and synthesis problem can be formulated as constraint satisfaction problems. Interactive layout systems like Electric [4] and procedural layout languages like Igloo [5] incorporates linear inequality constraints. These constraints specify relative placements of layout components, and facilitates graphical and procedural specifications of parameterizable layouts. Standard satisfaction algorithms can be used to solve a system of linear inequality constraints to generate compacted layouts. However, constraint satisfaction fails when arbitrary constraints are introduced into these constraint systems.

Different types of constraints are incorporated in VEXED [2], which relies on causal knowledge embodied in CRITTER [6] to propagate and check constraints on the value, encoding, and timing of signals from one part of the circuit to another. VEXED also incorporates the BULLDOG [7] system to manage layout constraints. By propagating specifications and behaviors of designs, VEXED supports the least-commitment design strategy. However, by embedding propagation knowledge as part of the causal knowledge of "datastreams" and "functions", VEXED makes it difficult to extend its constraint system. A unified framework of constraint propagation,

by which different types of constraints are integrated, would greatly facilitate arbitrary extensions to the constraint system.

Propagation of constraints as a general analysis method has been used in both analysis programs like EL [8] and synthesis programs like MOLGEN [1]. EL performs electrical analysis of circuits by propagating algebraic expressions for voltages and currents across circuit elements. Physical laws and circuit models serve as the constraints, and locality of circuits is exploited to reduce the amount of algebraic computation. On the other hand, MOLGEN performs hierarchical planning for genetic cloning experiments. It uses constraint propagation to coordinate requirements from different subproblems, and to ensure that separate parts of the solution plan do not interfere with one another.

ThingLab [9] is an interactive, constraint based environment for building simulation systems. Written in Smalltalk-72 [10], ThingLab has a powerful graphics interface controlled by constraint propagation, and provides an object-oriented paradigm for building system models. Being a property of objects, constraints are inherited in the object hierarchy. However, since ThingLab does not recognize class objects as "Things", hierarchical models cannot be constructed.

CONSTRAINT [11] is an interactive language that does permit the use of hierarchical models. The language provides a set of primitive constraints, from which compound constraints can be defined according to a rule of abstraction. Hierarchical constraint propagation is performed whenever user changes variable values, and dependency information of propagated values are recorded so that users may trace the consequences of a value assignment as well as the antecedents for a derived value.

III Constraint Propagation

This section describes the general framework of constraint propagation in STEM. Although this framework is developed with STEM in mind, it is general enough to be implemented in any object-oriented programming environment. This provides a foundation on which application specific constraints and variables can be easily implemented.

Variables and Constraints

In STEM, design objects have properties that are Variable objects (simply "variable" from here on). A variable has a value, a list of constraints, and a dependency record. Two operations on a variable may change its value: value-assignment, and value-propagation. Value-assignments are performed by objects outside the constraint networks, and represent external input to the networks. Value-propagations are performed by constraints on the variable, and represent internal adjustments of the constraints. When a value is "assigned" to a variable, the variable propagates this new value to all of its constraints. When a value is "propagated" to a variable, the variable propagates this new value to all of its constraints except that which propagated the value in the first place.

A constraint object specifies a "desirable" relationship among its arguments. To be useful, one or more arguments of a constraint must be variables. Each type (or class) of constraints has a propagation procedure (or method) and a satisfaction test, which collectively define the semantics of the constraint. When a variable propagates its new value to a constraint object (as a result of a changed value), the propagation method of the constraint is responsible for making any necessary value-propagations (to other variables in the constraint) to maintain the validity of the constraint. This, in turn, may trigger further constraint propagation. At the end of propagation, the satisfaction methods of all affected constraints are invoked. Inconsistencies in the new variable values manifest themselves as constraint violations.

Propagation Scheduling

When a variable propagates a new value to a constraint, it activates the constraint. Depending on its nature, an activated constraint either performs the propagation method as soon as it is activated, or schedules itself on an agenda of activated constraint, to be propagated later. For example, when a variable activates an Equality-Constraint, the new value of the variable is propagated to all other variables in the constraint, maintaining equality among these variables. For multi-directional constraints like this, in which the propagation variable depends on the variable which activated the constraint, the propagation is performed as soon as possible.

On the other hand, when a Max-Of-Constraint is activated, it calculates the maximum value of its argument variables and propagates this to its result variable, no matter which variable has changed value. For functional constraints like this, in which the propagation is unique and independent of the variable that activated the constraint, propagation can be delayed. A scheduled constraint may be activated several times (due to changes in multiple variables in the constraint), but only propagated once. By careful use of such scheduled constraints, pre-processing of constraints to remove cycles is not necessary.

Generally, there are more than one constraint agenda in the system, each with a different priority. Whenever (activated) multi-directional constraints are exhausted, a scheduled constraint is removed from the highest priority agenda that is non-empty, and propagated. This may trigger additional multi-directional constraint propagation and schedule more constraints. Constraint propagation therefore traverses the constraint networks in a combination of depth-first and breadth-first manner: depth-first for sub-graphs of multi-directional constraints, and breadth-first otherwise.

Termination Criteria

In order to guarantee termination of constraint propagation, each variable is allowed to change value only once in response to each (external) variable value-assignment. This prevents circular constraint propagation while allowing circular constraints. The wavefront of propagation stops either when an activated constraint does not make further value propagation, or when a propagated value agrees with the current value of a variable.

Constraint propagation terminates on the first detection of constraint violation. This occurs whenever a propagated value disagrees with the current value of a variable but cannot overwrite it, either because the variable has a user-specified value, or because the variable has already been visited once in the current cycle of propagation (i.e., circular propagation). If there is no constraint violation, propagation terminates when there are no

scheduled constraints after all activated multi-directional constraints are propagated. This termination is guaranteed since no variable propagates its value twice.

Constraint Violation

The propagation methods cannot guarantee that consistency is maintained in the constraint network. For example, invalid user-assigned variable values manifest themselves as constraint inconsistencies. These are checked for by the satisfaction tests of all visited constraints after each successful termination of constraint propagation.

When a constraint violation is detected, a violation handler is invoked. The default (inherited) handler restores the state of the constraint networks by setting all visited variables to their original values. However, different types of constraints may replace this with different violation handlers. For example, specialized constraint editors in STEM can be invoked to help users examine and debug the constraint networks. Constraint propagation can also be turned off if design violations are intentional. For example, changing the bitwidths of two connected cells triggers constraint violation as soon as the first bitwidth is changed, but changing the second bitwidth would remove the violation. Such changes can be made quickly if propagation is turned off.

Dependency Analysis

In an interactive environment, the editing of constraint networks is a frequent and important operation. Whenever a constraint is removed from a network, all propagated values which depend on the removed constraint must be erased to maintain the integrity of the network. This is accomplished by dependency analysis, which is traces all consequent (i.e., propagated) values of a variable (and/or a constraint), as well as all variable and constraints that a propagated value depends on. Dependency records in the variables are maintained for this purpose.

Dependency records specify the source of variable values. If a variable has an assigned value, its dependency record may be either "USER" or "APPLICATION". This is used to determine if a propagated value can overwrite the variable. The overwrite rule in STEM is that users have the highest priority, followed by constraint propagation, and then applications.

For a propagated value, the associated dependency record indicates the source constraint, and whatever data that are necessary for the source constraint to determine the variables this propagated value depends on. This record is constructed by the source constraint during propagation, and analyzed by the same constraint during dependency analysis.

For example, for most functional constraints, the result variable value depends on all argument variables in the constraint. Consequently, no explicit dependency data is necessary in the dependency records of functional result values. The dependency is implicit in the semantics of the functional constraints. On the other hand, a multi-directional constraint must record at least the variable which activated the propagation, in order to determine the dependencies of its propagated values.

Constraint Addition and Deletion

When constraints are added and deleted, constraint propagation must be performed to adjust variable values to changes in the constraint network. However, since no variable has changed value, the normal propagation trigger is not activated. A separate triggering mechanism is therefore developed for constraint editing.

When a constraint is added to a variable, all variables in the constraint are successively given a chance to propagate their values to other variables in the constraint, starting with the user-specified variables. If a consistent set of variable values cannot be found, constraint violation occurs. When a constraint is removed from a variable, all propagated values that depend on this constraint-variable pair are erased. Dependency analysis facilitates this erasure, and ensures that no unwarranted values remain in the network.

IV Integration of Constraints

STEM is an object-oriented IC design environment aimed at integrating design automation tools with manual design. It is written in Smalltalk-80 [12], and is based on Smalltalk's Model-View-Controller concept. A cell is represented by a Smalltalk class in STEM, and the use of a cell is represented by a Smalltalk instance of the cell class. In order to incorporate constraints in STEM, all instance variables of cells are initialized to variable objects on instantiation, and all access methods are modified accordingly.

Constraint Formulation

Constraints can either be added and deleted by the access methods in cell objects, or by designers through a constraint editor. For example, at the start of a design, a designer can specify the io-signals of the cell, and declare delay constraints between pairs of io-signals (e.g., delay from A to B must not be longer than 100ns). As the internal cell design progresses, delay constraints are instantiated (when subcells are added) and removed (when subcells are removed) by "addCell" and "removeCell" routines, respectively. If the internal delay network produces a delay time longer than the user-specified constraint, constraint violation results and the designer is warned. To continue the design, the designer can either turn off constraint propagation, change the internal cell design to achieve a shorter delay, or relax the violated delay constraint on the cell.

Hierarchical Propagation

Constraint networks in STEM are organized around the design hierarchy to take advantage of abstractions inherent in the designs. STEM's dual declaration of instance variables in cells [3] is central to this scheme of hierarchical propagation of constraints. When the user declares an instance variable for a cell in STEM, a pair of similarly named instance variables are created: one in the cell class definition, another in the cell instance definition.

The dual instance variables are implicit constraints on one another. Instance variables in a cell class specify the characteristics of the cell and constrain the uses of the cell, while instance variables in cell instances specify the different ways the cell has been used and constrain the internal design of the cell. For example, if the bitwidth of a cell class is variable but cannot exceed 16, then bitwidths for instances of this cell cannot exceed 16. On the other hand, if an instance of a cell has a bitwidth of 8 by being connected to an 8-bit port, then the cell class's internal design must be such that it can

be used as an 8-bit component.

These implicit constraints establish the only linkage among constraint networks in different cells, abstracting these networks into a hierarchy that parallels the design hierarchy. Implicit constraints are scheduled in an agenda with the lowest priority. This scheduling propagates constraints within the same levels in the design hierarchy as much as possible, before propagating to constraints in other levels of the hierarchy. Hierarchical constraint propagation checks for inconsistencies between the characteristics of the cell and the ways that cell has been used.

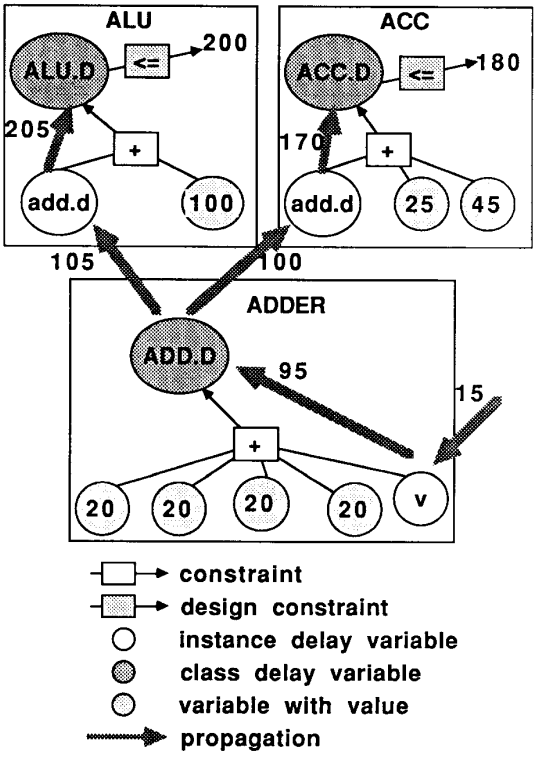

Fig. 1: Constraint propagation between class and instance delay variables

Fig. 1 shows an example of constraint propagation from instance variable of a cell class ("ADDER.D") to instance variables of cell instances ("add.d"). Two instances of an adder have different delay requirements. One in an ALU design has a delay constraint of 100ns or less, due to constraint on ALU delay ("ALU.D") and internal design of the ALU. Similarly, another adder instance in an accumulator (ACC) design has a delay constraint of 110ns or less. When the adder designed is finished, the last delay value in the critical path is known (15ns assigned to "v"). Constraint propagation determines the class delay ("ADD.D") to be 95ns, and invokes delay calculation routines to propagate values to its dual variables. This sets the adder delay in the ALU

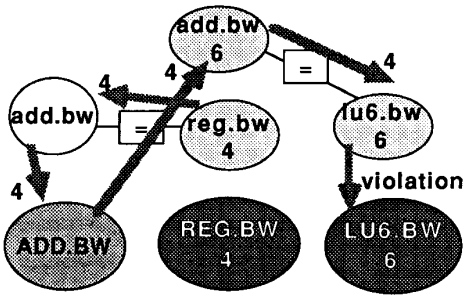

Fig. 2: Constraint propagation of bitwidths between instance and class variables

to 105ns (due to loading) and "ALU.D" to 205ns. Similarly, the adder delay in ACC is set to 100ns, and "ACC.D" is set to 170ns. After constraint propagation, when all affected constraints are tested for satisfaction, the "<=" constraint in ALU reports a constraint violation. The designer then has the option of re-designing either the adder cell, or the ALU.

As another example (see fig. 2), if the designer connects an instance of an adder to a 4-bit register, constraint propagation sets the bitwidth variable of the adder instance to 4, which sets its dual variable the adder cell class to 4. If the bitwidth variable in the adder class has no previous value, its value is set to 4. If it has a user-specified value other than 4, constraint violation occurs. In fig. 2, the adder class bitwidth has a propagated value of 6 due to a connectivity constraint in another adder instance. What happens in this case is that the variable overwrites its previous value with 4, and propagates it to its other dual variables. Constraint violation is not triggered at this point because this may be a transient inconsistency, and that subsequent propagation may remove the justification for the previous bitwidth of 6. Eventually, constraint violation occurs in the 6-bit adder instance when it tries to change its bitwidth to 4.

V Example Applications

In order to demonstrate the potentials of constraint propagation in IC designs, three types of sample applications have been implemented in STEM using its framework of constraint propagation.

Consistency Maintenance

Update-constraint [13] is one of the first types of constraints used in STEM. It maintains data consistency by erasing the calculated value of a property variable whenever any data this value depends on changes. Update-constraints facilitate application integration in STEM by allowing results of applications to be stored in property variables without the risk of inconsistency.

Simple constraints like Equality-Constraint and Arithmetic-Constraints are useful for specifying simple relationships among parameters in a design. These constraints reduce the amount of data entry and maintain (user specified) consistency among design variables.

For example, a designer can easily specify that all ports of a cell must have the same bitwidth, by instantiating an Equality-Constraint on the bitwidth variables. Subsequently, this Equality-Constraint sets all of these bitwidth variables whenever one is assigned a value, and prevents different values to be assigned to different bitwidth variables.

Incremental Design Checking

Constraints on delay, area and signal types have been integrated into STEM. As a design is entered, constraints implied by structural and electrical connectivities and physical placements of components are instantiated by the access methods (e.g., the connection of two signals implies a compatibility-constraint on their electrical signal types). Designers can also declare explicit constraints (e.g., maximum area and delays) that represent specifications for the design. For delay variables, constraint networks corresponding to circuit connectivities compute the delay between two points as the maximum delay of all paths between these points. The implicit constraints of area variables check that the area of a cell instance is at least the area of the cell class's, and derive default values for cell areas when these are not specified. For signal typing variables (e.g., bitwidth and data-type), constraint networks corresponding to nets derive signal types from connected signals, and check for connections among incompatible signals (e.g., signal with different bitwidths).

These constraints perform design checking by propagating design characteristics up the design hierarchy as soon as they are available and checking them with design specifications. Incremental checking is achieved by the incremental editing of constraint networks (as the design is edited), and the incremental propagation of constraints in these networks. Additional design checking can be incorporated by introducing new constraint types into the networks and modifying access methods in cells to maintain these new constraints.

Fig. 3 shows an 8-bit accumulator with delay and area constraints that is built by butting an adder and a register, along with the constraint networks instantiated for the structure. When the designer connects the output of the register to the output of the accumulator cell, signal typing constraints check that the bitwidth of the register, as well as that of the adder connected to it, is 8. When the adder is designed, the delay and area of the adder class are propagated to the delay and area of the adder instance.

The design of the register is then guided by the accumulator constraints and the adder characteristics. As soon as the area of the register becomes too large to fit in the accumulator with the adder, constraint violation warns the designer. Similarly, as soon as the designer chooses a register bit-slice that has a delay longer than what the adder (characteristics) and the accumulator (constraints) permit, the designer is warned. Again, designers have the option of turning off constraint propagation in STEM if these violations are temporary and intentional.

Module Validation

STEM's class hierarchy of cells implements a design hierarchy that groups alternative designs as descendents of a generic cell. For example, ADDER can have several subclasses with different carry propagation

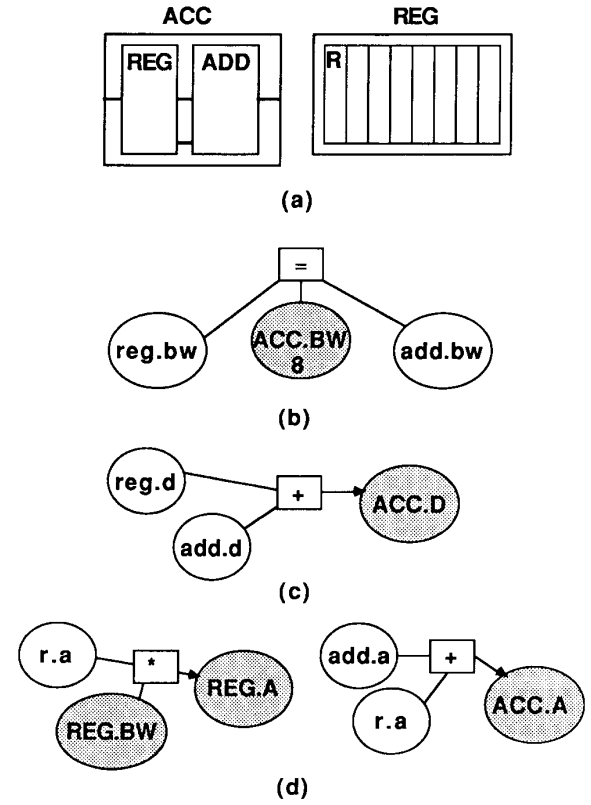

Fig. 3: (a) ACC and REG internal structures (b) bit-width constraints (c) delay constraints (d) area constraints

schemes, and each subclass can have subclasses that implement different design tradeoffs. With automatic module selection, a cell can be designed using generic subcells (e.g., instances of ADDER). The internal implementation of each instance of the cell is customized by module selection to the constraints of its environments.

Module selection is implemented in Cell and inherited by all cells in STEM. A simple search technique of generate and test, augmented with hierarchical planning and heuristic pruning of the search tree, is used to find valid implementations for a cell instance. The candidate implementations of a generic subcell consists of instances of all descendent classes of the generic cell. Each of these can be tested for validity by replacing the generic subcell with the candidate implementation and testing for constraint violation. Constraint propagation checks if the characteristics of a candidate cell satisfy all constraints in its environment.

Module selection either selects the first valid design, or generates a list of valid designs and lets the designer selects one. Currently, delay, area and signal typing are the only criteria for module selection, although more can be added by defining additional constraint types.

Fig. 4 shows a simple example of module selection. ADD8 is a generic, 8-bit adder cell with two subclasses: ADD8.RC (a ripple-carry adder), and ADD8.CS (a carry-select adder). ALU8 is designed with an instance of LU8 (an 8-bit logic unit), and an instance of ADD8, the generic adder. If two instances of ALU have different constraints, one with a tight area constraint (3a), and the other with a tight delay constraint (3b), different implementations are selected by module selection.

VI Performance and Limitations

The speed performance of constraint propagation are acceptable for interactive design provided that the constraint networks are sparse. Low-level constraints, such as layout constraints, are not suitable applications of this framework because more specialized data structures (e.g., corner stitching) and constraint satisfaction algorithms (e.g., shortest-path algorithms on graphs) are necessary to achieve adequate performance. However, higher-level constraint networks tend to be sparse, and as such are good candidate applications for the constraint propagation techniques reported in this paper.

VII Conclusions

An object-oriented framework of constraint propagation that supports arbitrary constraints in hierarchical design environments has been presented in this paper. Hierarchical propagation of constraints is used to manage the interactions among designs, and supports a least-commitment strategy in IC design. By itself, constraint propagation can be used to maintain consistency of calculated data, reduce the amount of data entry, and provide a convenient way for designers to specify design-specific relationships among variables. When integrated into other design tools, constraint propagation can perform incremental design checking, and help designers and design tools make design decisions. By managing large quantities of simple design interactions, constraint propagation can become a powerful tool in an IC design environment.

VIII Acknowledgements

This work was supported in part by the Natural Science and Engineering Research Council of Canada, and by the Alberta Microelectronics Center.

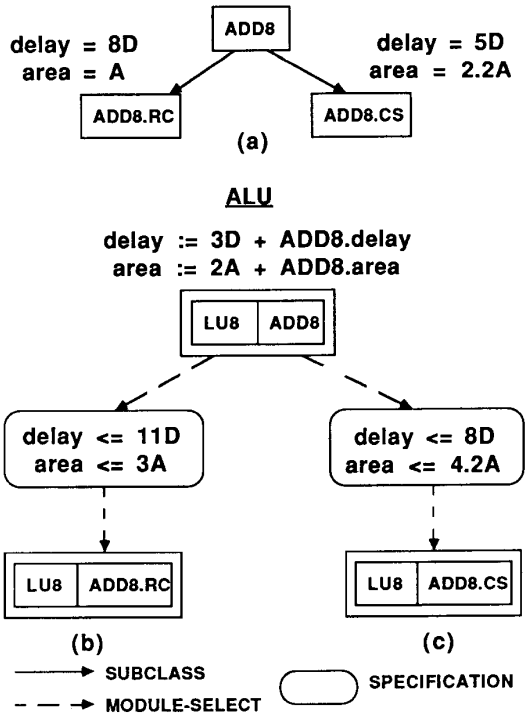

Fig. 4: (a) ADD8 has two subclasses; (b) tight area spec: use ADD8.RC; (c) tight delay spec: use ADD8.CS

References

[1] M. Stefik, "Planning with Constraints (MOLGEN: Part 1)", Artificial Intelligence, Artificial Intelligence 16, 1981, pp111-140.

[2] T. M. Mitchell, L. I. Steinberg & J. S. Shulman, "A Knowledge-Based Approach to Design", IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. PAM1-7, No. 5, September 1985, pp502-510.

[3] E. F. Girczyc & T. A. Ly, "STEM: An IC Design Environment Based on the Smalltalk Model-View-Controller Construct", Proceedings, 24th Design Automation Conf., 1987.

[4] S. Rubin, *Computer Aids for VLSI Design,* Addison-Wesley, Don Mills, Ontario, 1987.

[5] M. Pulver & M. I. Elmasry, "Using Igloo: A Constraint Based Layout Language for VLSI", Proceedings, CCVLSI-87, 1987, pp81-86.

[6] V. E. Kelly, "The CRITTER System", Proceedings, 21st Design Automation Conference, 1984.

[7] J. Roach, "The Rectangle Placement Language", Proceedings, IEEE 21st Design Automation Conf., June 1984, pp405-411.

[8] R. M. Stallman & G. J. Sussman, "Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis", Artificial Intelligence 9, 1977, pp135-196.

[9] A. Borning, "ThingLab -- An Object-Oriented System for Building Simulations Using Constraints", Proceedings, Fifth International Joint Conference on Artificial Intelligence(IJCAI-5), MIT, Cambridge, Aug 1977, pp497-498.

[10] A. Goldberg & A. Kay (eds.), *Smalltalk-72 Instruction Manual,* Xerox Palo Alto Research Center, SSL 76-6, 1976.

[11] G. J. Sussman & G. L. Steele Jr., "CONSTRAINTS- A Language for Expressing Almost-Hierarchical Descriptions", Artificial Intelligence 14, 1980, pp1-39.

[12] A. Goldberg & D. Robson, *Smalltalk-80 The Language and Its Implementation,* Addison-Wesley, Don Mills, Ontario, 1983.

[13] T. A. Ly, R. Miller & E. F. Girczyc, "Interfacing Application Programs to an Object-Oriented Database Using Views and Controllers", Proceedings, CCVLSI-87, 1987.