Constraint-Based Codesign (CBC) of Embedded Systems: The UML Approach

Chonlameth Arpnikanondt
Vijay K. Madisetti
Center for Signal & Image Processing (CSIP)
Electrical & Computer Engineering
Georgia Tech, Atlanta, GA 30332-0250
Yamacraw Technical Report #: YES-TR-99-01

Abstract

We propose a methodology for hardware/software codesign of embedded systems, using the Unified Modeling Language (UML) to realize it. The proposed methodology is design constraint driven, thus facilitating requirements-driven synthesis and verification for system-on-chip (SOC) and system-on-package (SOP) designs.

1.0 Introduction

The process of hardware/software (HW/SW) codesign involves four main tasks: *allocation, partitioning, scheduling* and *communication synthesis*. Allocation involves a components selection whereas partitioning a placement of allocated components into hardware and software. Scheduling insures an optimal design without violating any timing and/or resource constraints. Communication synthesis makes possible a proper interaction among components. The robustness of a codesign methodology consequently depends upon how well it addresses such tasks. A traditional HW/SW codesign methodology (Figure 1) usually commences with a set of specifications. These specifications, often incomplete and/or captured in a non-formal language, become the reference to adhere to while making design decisions. Most of the time these design decisions are made *a priori* early in the design process. Compounded by lack of a collaborative HW/SW design environment as well as by an increasing complexity of a codesign system, the resulting design tends to be sub-optimal.

Over the years researchers have produced numerous methodologies and tools to tackle such inherent codesign problems. Most, if not all, incorporate some kind of a formal specification language to resolve an ambiguity incurred by the use of non-formal languages. Techniques such as synthesis, re-use and co-validation (co-simulation/co-verification) are favorably adopted as supporting CAD tools become more readily available. Some methodologies, such as SpecC [1] and POLIS [2], exploit a unified HW/SW representation to boost the robustness of the design, while the Rapid-prototyping of Application Specific Signal Processing (RASSP)'s approach [3] depends on rapid prototyping and re-use.

 (C) Georgia Tech, 1999.

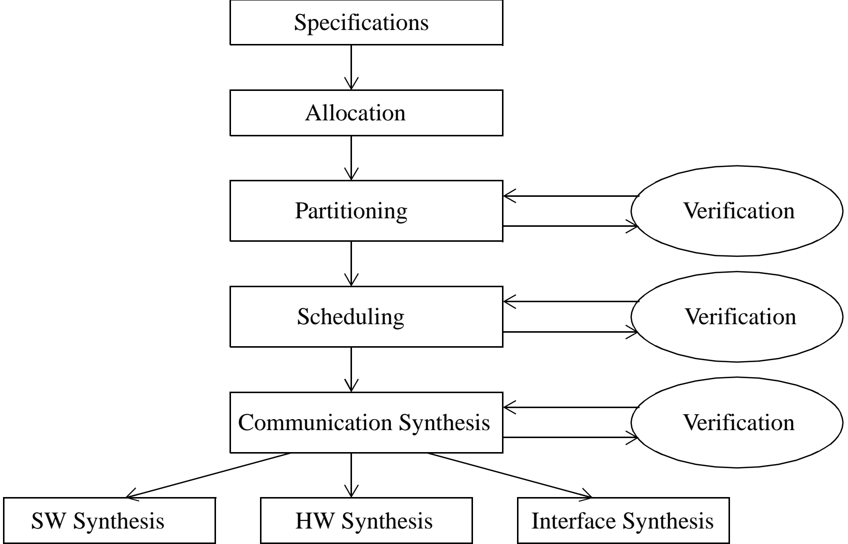


Figure 1: Generic Codesign Methodology

Of particular interest is a class of methodologies that relies on constraints to drive the codesign flow. This approach models design constraints from a set of requirements in such a way that they can, in turn, imply bounds on design components. A design is then explored within such bounds. As the design progresses, secondary constraints may be derived and the design may be explored further. When no more constraints may be derived, the design that fulfills all constraints becomes *feasible*. We will refer to this class of methodology as a *Constraint-Based Codesign (CBC)*. Figure 2 summarizes this methodology.

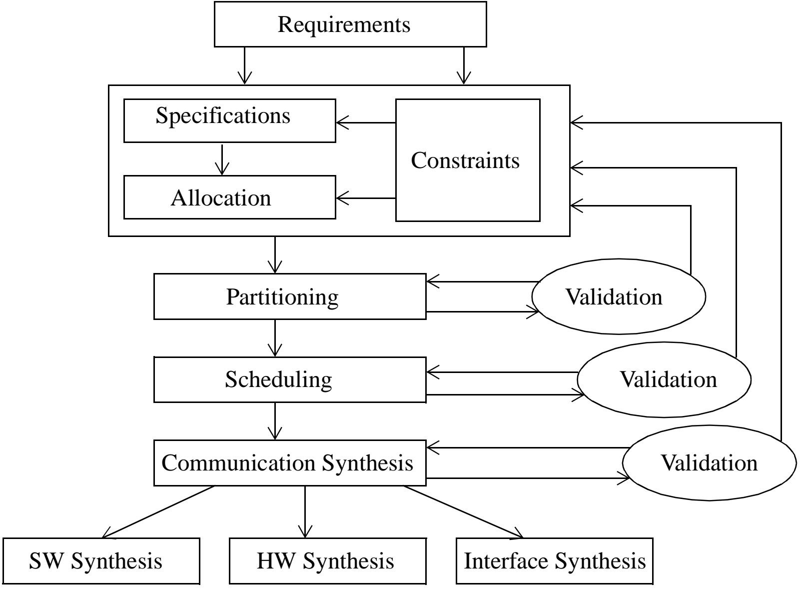


Figure 2: Constraint-Based Codesign Methodology

(C) Georgia Tech, 1999.

Few of the existing methodologies today, if any, may be *fully* regarded as being constraint-based. Many, however, do provide some mechanism for capturing timing-constraints into the design. SpecC [1], for instance, allows both time and a range of time to be captured and used as constraints by a scheduler. Such a methodology as VSpec [4] is capable of modeling both behavioral and physical constraints as well as providing means to verify the feasibility of the design within bounds of such constraints. Nonetheless, VSpec is not primarily designed to handle the complexity of codesign; its present scope only encompasses a constraint-based hardware design.

2.0 The UML Approach

The Unified Modeling Language (UML) is a convenient tool to realize the constraint-based codesign (CBC) for several reasons.

- The Object Constraint Language (OCL), a permanent part of the UML specifications, is a powerful formal language designed to cope with constraints that cannot be captured in UML's basic mechanism. OCL defines numerous useful semantics--among them are pre-/post-conditioning and composition. Indeed, it shares *most of the semantics with Rosetta*, which is another on-going effort by the System Level Design Language (SLDL) working group. References [5] and [6] contain documents and links related to both OCL and Rosetta, respectively.
- Being a semi-formal, graphical language makes UML an excellent analysis tool.
- The language constructs support modeling of such desirable system characteristics as concurency, synchronization, and composition, to name a few.
- As an object oriented language, UML allows the design to take place at a higher level of abstraction and to become more manageable. This facilitates handling of the complexity of a large SoC design.
- With appropriate extensions, UML allows for an automatic code generation.
- UML is a widely accepted standard (http://www.uml.org).

In addition to UML, the Specification and Description Language/Message Sequence Chart (SDL/MSC) is another good candidate for the CBC realization [7]. With some extensions, SDL/MSC can offer most of what UML does for the constraint-based codesign. It is a formal specification language, a widely used standard (ITU-T Recommendation Z.100) that is now object-oriented. Being a formal specification language, however, it lacks the same UML's ability for system requirements analysis --- a crucial necessity for the constraint-based codesign. In addition, it currently does not support an extensive constraint modeling capability.

Figure 3 summarizes the UML approach to the constraint-based codesign methodology (CBC/UML). The methodology begins by analyzing system requirements through Use Case. This is an iterative analytical refinement process which results in a set of design scenarios and their associated constraints. The next step essentially involves two independent activities: *Firm Formalization* and *Allocation*. Formalization is a process that tranforms word descriptions into a formal language. A formalization is said to be *firm*

Extensions & Terminology

Formalization: A process that transforms word descriptions into Java/OCL expressions. A formalization is said to be ***firm*** if all intranslatable constraints & all allocated functional units are processed. It is said to be ***hard*** only when all design constraints are processed.

Translatable Constraint: A constraint that can be translated into source codes, while ***Intranslatable Constraint*** cannot. Power, number of gates are some examples of an intranslatable constraint.

Neutralization: A process that substitutes all Java-like expressions with designated class methods so as to make them neutral of any HW/SW preference.

Note: An asterisk (*) on an activity implies a repetition of that activity.

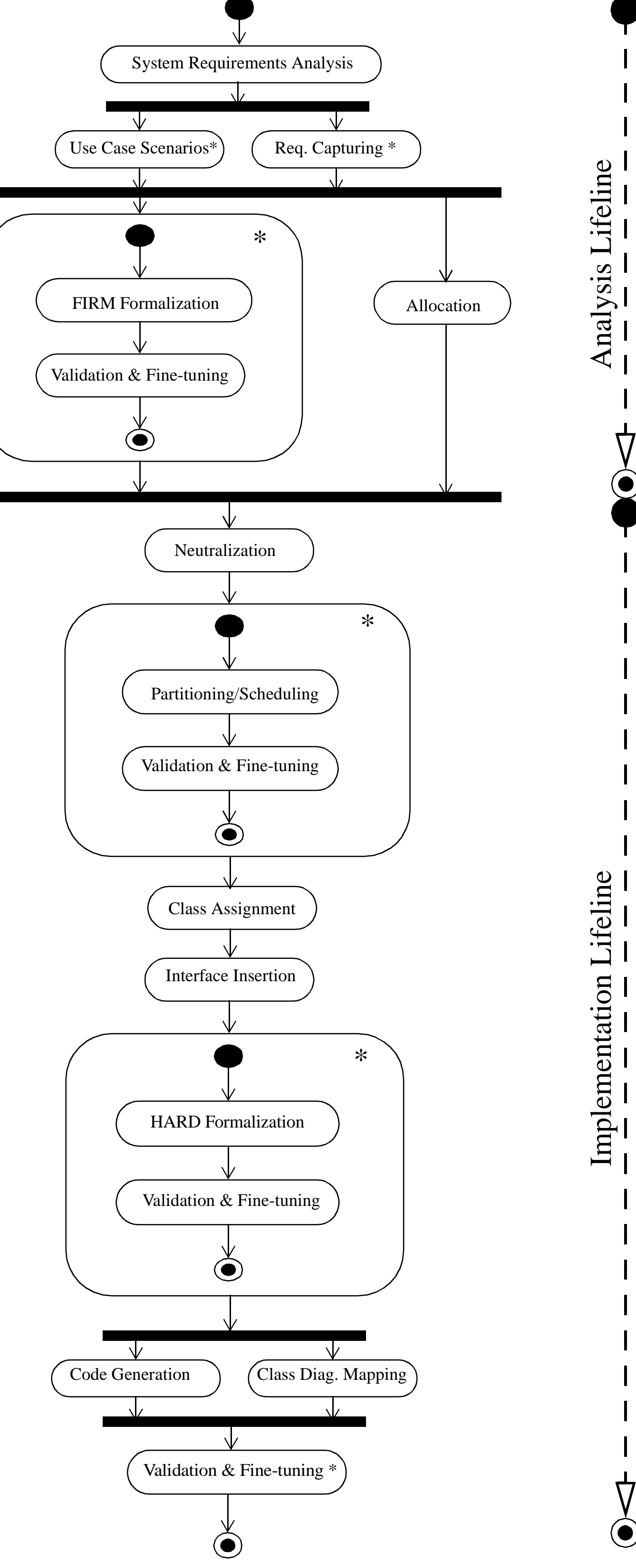


Figure 3: CBC/UML Methodology

(C) Georgia Tech, 1999.

if all intranslatable constraints and all allocated components are processed. It is said to be *hard* only when all design constraints are processed. A constraint is either *translatable* or *intranslatable*. A translatable constraint can be translated or incorporated into source codes, while an intranslatable constraint cannot. Physical constraints such as power and heat dissipation are intranslatable. During Allocation, the designer manually selects functional units (or IP cores) for implementing the design. These IP cores could represent functionalities without committing to implementations. For example, an Adder is a functional unit with a Ripple Carry Adder as one of its implementations. Then the design is made unbiased of HW/SW preference at the *Neutralization* step before being fed to a Linear Programming (LP) solver during the *Partitioning/Scheduling* process. Only after an insertion of interface (*Interface Insertion*) is complete, the design can be hard formalized. *Hard Formalization* transforms UML from being semi-formal to being formal and ready for the *Code Generation* process. *Validation and Fine-tuning* verifies the correctness of the design and possibly adjusts the constraints to optimize the design further. It is noteworthy that almost everything as represented by the CBC/UML is a constraint: diagrams, functional units, HW topology, timing, etc.

Inherently a modeling language, CBC/UML is suitable for a wide range of systems design. Systems-on-a-Chip (SoC), Systems-on-a-Package (SoP), mixed-signal telecommuncation systems or embedded systems should be well within the scope. The designer only needs an appropriate code generator to translate the CBC/UML *hard formalized* model into source code in the corresponding domain.

In the following section, we present the design of a simple embedded system example using CBC for the purposes of illustrating its approach.

3.0 Example: Root Finding via Newton Method

3.1 Problem Statement

Given an equation $y = ax^2+bx+c$, the Newton method finds a root of $y = 0$ through the following iterative procedure:

1. Acquire first guess, x_i
2. Compute $y_i = ax_i^2+bx_i+c$ and a slope $m_i = 2ax_i+b$
3. Compute a new guess, $x_{i+1} = (m_i x_i - y_i) / (m_i)$
4. Repeat step 2 until $/ y_{i+1} - y_i / \leq eps$, where eps is some small number

However, this example further requires:

1. Number of gates to be less than 20k
2. Maximum of 15 μsec/iteration
3. Exception occurs if no root found after 30 iterations. The system resets.
4. Exception occurs if no response from HW to SW for longer than 1 second. The system resets.

5. User inputs parameters via a PC.
6. Synchronous system, with clock speed ≤ 66 MHz.

3.2 System Requirements Analysis

The main task of this phase is to analyze and capture all requirements for the system in terms of UML representations. Two UML tools come in handy: Use Case and Activity Diagram.

3.2.1 Iteration 1: Scenarios & Requirements

This iteration shows an interaction between a system and its environment, as well as how a certain requirement affects each scenario. The designer may spend time refining the Use Case before actually coming to the final representation, as shown in Figure 4.

Observation 1: Some scenarios in the Use Case diagram may not associate themselves with any actor because of the autonomous nature of an embedded system.

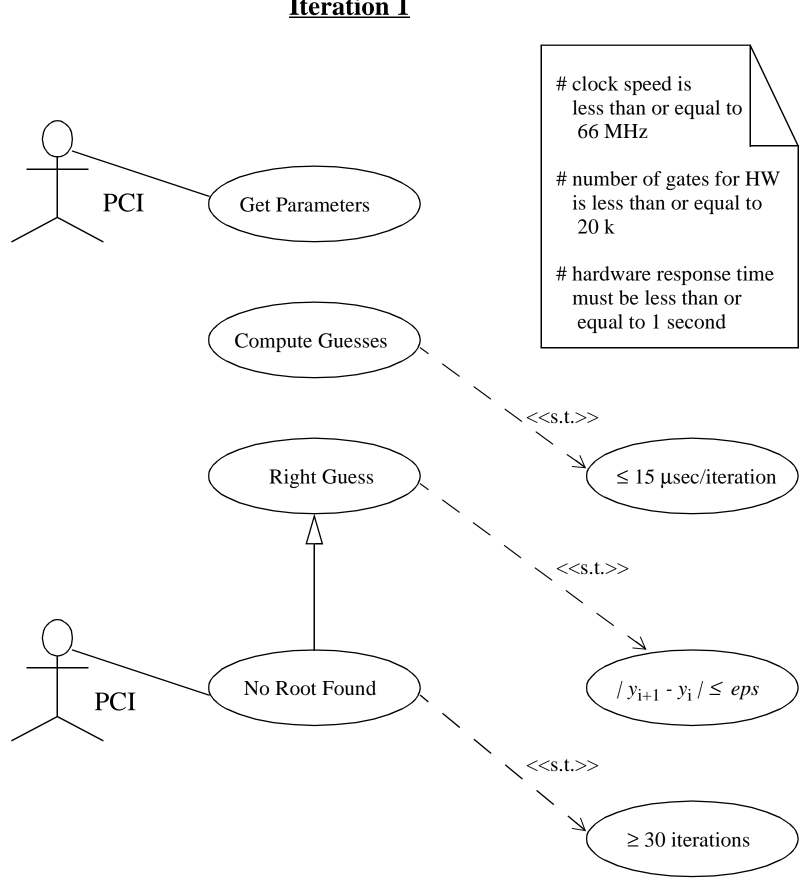


Figure 4: System Analysis with *Use Case*

(C) Georgia Tech, 1999.

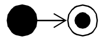
3.2.2 Iteration 2-3: Activity Diagrams

These iterations prepare for the Firm Formalization process by first mapping the Use Case in Figure 4 to an Activity diagram as shown in Figure 5. A more refined diagram, such as the one illustrated in Figure 6, can be derived iteratively.

Extensions & Terminology

Bubble Label: A bubble with a label posted within an activity diagram. A bubble label is a class with predefined methods all by itself. Placing a bubble label on an activity provides means for imposing constraints onto that activity. A single bubble only lives through an imposed activity. A double bubble has a lifeline along the path which starts with the first bubble and end with the second bubble of the same label.

Globalization: The following symbol globalizes any bubble input to it.



In Iteration 2, a globally distributed clock signal is represented by the pre-defined clock bubble input to the globalization symbol. This clock is constrained to have the speed of less than or equal to 66 MHz.

Clock: A clock signal is represented by a bubbled *c*.

Initialization: Initializing a certain value within an activity may be carried out by placing an initialization operation *init()* on an input arrow to that activity. This is shown in Figure 6.

Exception: An exception is represented by a bubbled *e*.

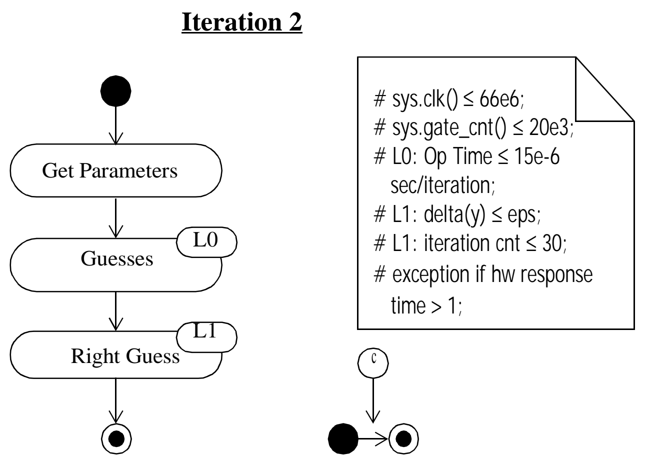


Figure 5: A Derived *Activity Diagram*

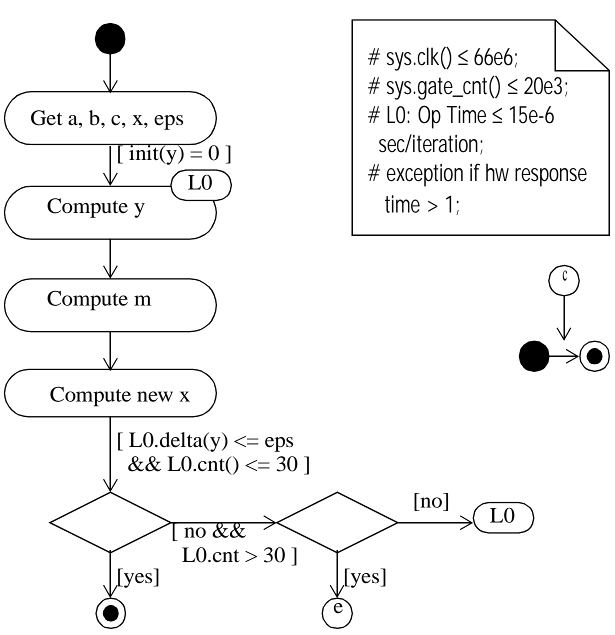


Figure 6: A Refined *Activity Diagram*

3.2.3 Iteration 4: Firm Formalization

Observation 2: Hardware is *latch-capable* if it intrinsically can latch outputs from a specified task without requiring additional latches to be explicitly placed. A FIFO, a D-F/F are some examples of latch-capable hardware.

2-1) Outputs of any activity may be latched with // (Figure 7),

2-2) If it is decided that a latch-capable component be chosen for an activity with latched outputs, the latches at the outputs shall be resolved so as to prevent redundancy.

Extensions & Terminology

Output Flow: While a pair of square brackets represents a set of constraints, a pair of parentheses represents an output flow.

Data Type: An extension tagged after each data refers to its corresponding data type. For example, *a.f* is equivalent to saying *a of type float*.

Note: A simulator is needed to test a functional correctness of the system at this iteration.

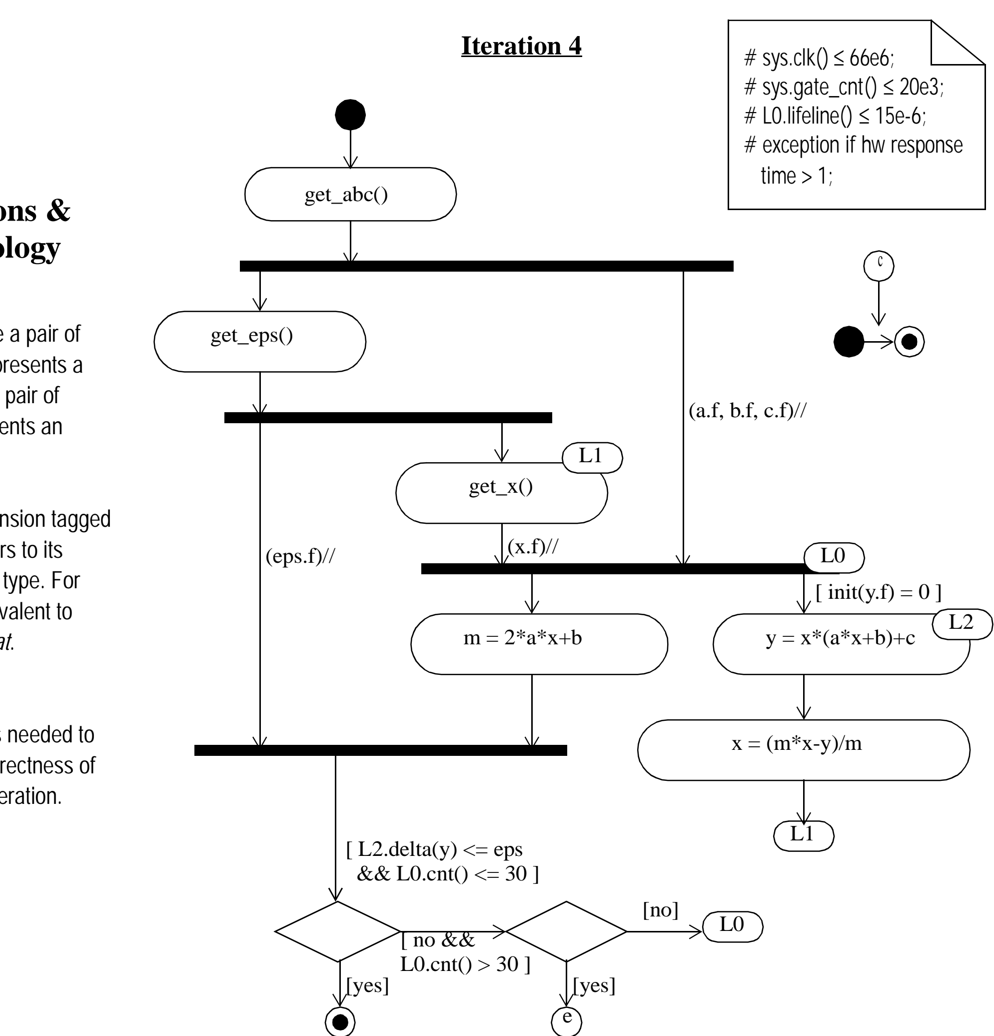


Figure 7: A Firmly Formalized *Activity Diagram*

3.2.4 Allocation

When functional units are allocated, they become an additional set of constraints. The partitioner/scheduler runs this new set of constraints against its algorithm, and classifies activities to belong to either HW or SW.

The following functional units (or IP cores) are selected for this example:

- multiplier
- adder
- left shifter
- FIFO

Observation 3: Any activity at all can be implemented in SW, but can only be implemented in HW so long as the allocated functional units allow.

O3.1) By representing an expression in an activity in terms of pre-defined class methods, such an activity becomes neutral of HW/SW preference.

O3.2) Any operator in an activity that does not have a functional unit associated with it is strictly SW.

O3.3) Any operator in an activity that has a functional unit associated with it is either HW or SW or both.

O3.4) Multi-functional unit can be represented by a class with several methods.

Table 1 defines neutral class methods for each operator in an activity diagram.

Observation 4: Prior to partitioning/scheduling, a functional unit is either *determined* or *undertermined*.

O4.1) A *determined* functional unit is one already known to belong to either HW or SW.

O4.2) An *undetermined* functional unit is one that may be later classified by the partitioning/scheduling algorithm as either HW or SW or both.

Table 1: Neutral Class Methods Definition

Neutral Methods	Operators	Allocated HW Components
add(*,*)	+	adder
mult(*,*)	*	multiplier
lshift(*)	(2^n)*	left shifter
fifo(*, ..)	get()	FIFO
div(*,*)	/	
twocomp(*)	(-1)*	

Note: "*" designates *any name* while ".." designates *unlimited number.*

3.3 System Implementation

As the design flows through the implementation lifeline, the system is gradually built to meet all the constraints captured earlier in the Analysis phase. This phase starts with a neutralization process, followed by a design partitioning/scheduling. Class Assignment and Interface Insertion then follow before the design gets its final formalization after which both HW and SW source codes can be generated.

It is to note that a good *cost-estimation algorithm* becomes a must for having the design automatically and correctly partitioned/scheduled.

Observation 5: Hardware topology enters the design as an additional set of constraints.

3.3.1 Iteration 5: Neutralization

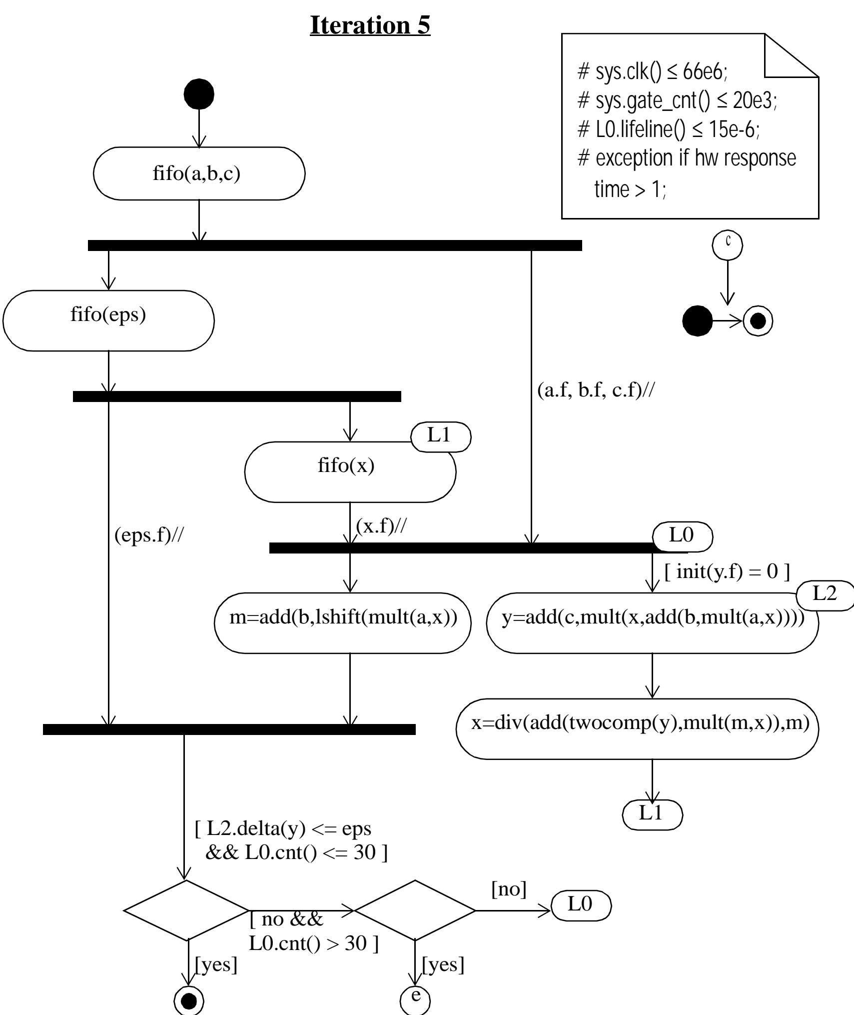


Figure 8: A Neutralized *Activity Diagram*

3.3.2 Iteration 6: Partitioning/Scheduling

Extensions & Terminology

Strike-through: Intranslatable constraints are struck through when they are resolved.

Last Value: A double quote can be used to capture an output value from a previous activity.

Activity ID: An activity id in the top portion of an activity capsule refers to a class to which such an activity belongs. An activity id is the string preceding the colon.

Class Hierarchy: A class hierarchy is also captured in the upper portion of an activity capsule. Each level of a hierarchy is separated by a period.

Note: Many notations come about to make UML representations of the same context more concise. Examples are //, ", etc.

Note: Specific implementations of functional units are picked as an LP solver searches for an optimal design.
The designer may opt to keep certain implementations while discard others and then iterate the partitioning/scheduling step to explore the design with a new set of constraints.

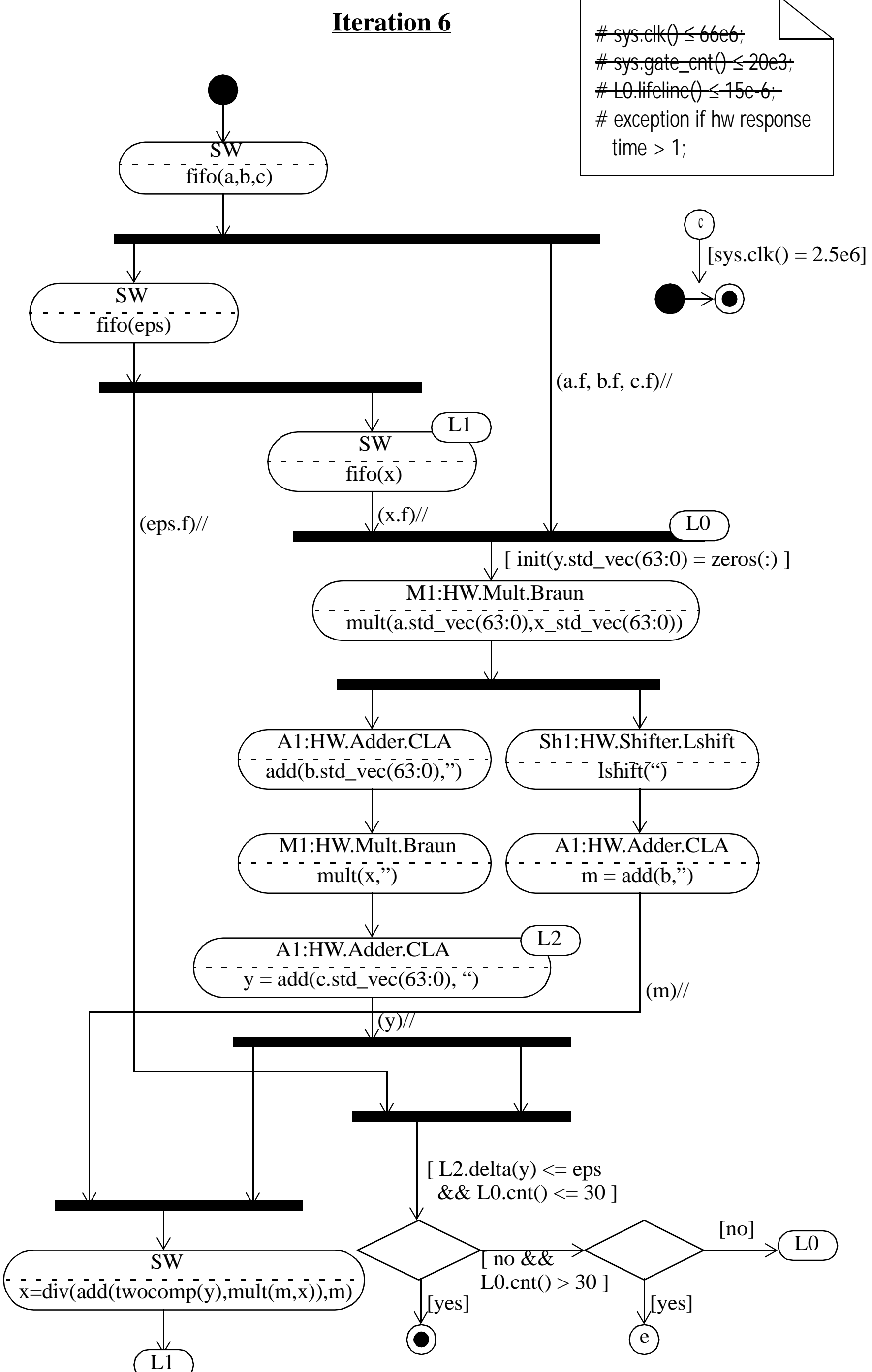


Figure 9: A Partitioned/Scheduled *Activity Diagram*

3.3.3 Iteration 7: Class Assignments & SW Activities Mapping

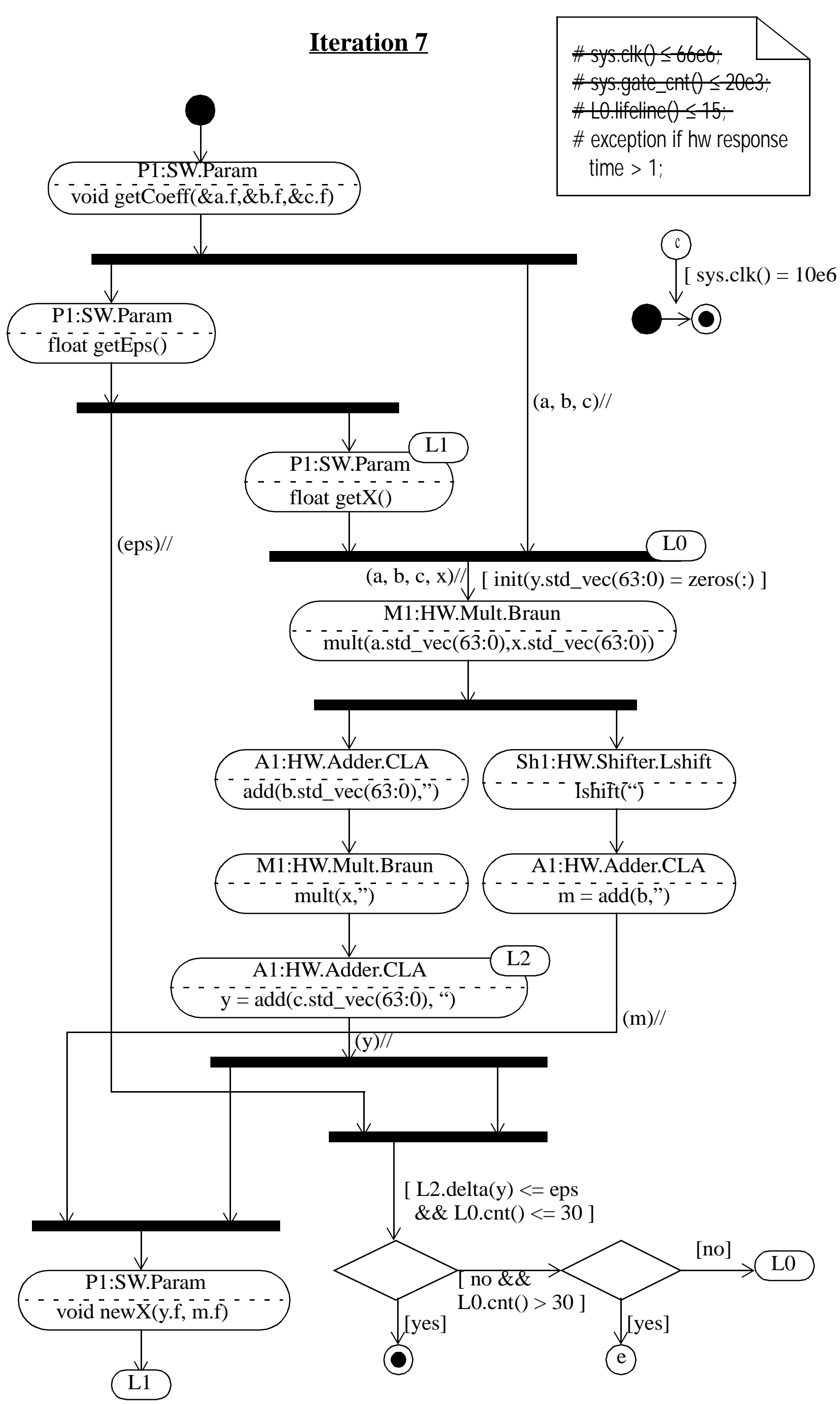


Figure 10: A Detailed *Activity Diagram*

3.3.4 Iteration 8: Interface Insertion & Hard Formalization

Extensions & Terminology

Timer: A timer is represented as a method within a bubble class. A timer associated with a single bubble implies that an activity imposed by that bubble is expected to finish its operation before the timer expires. Otherwise, a certain action (second argument) takes place. A timer associated with a double bubble implies that time elapsed between the first activity imposed by the first bubble and the second activity imposed by the second bubble must not exceed the time set by the timer at the start of the first activity.

Interface: An interface is represented by a pair of parallel lines with an arrow crossing them orthogonally. The head of an arrow points in the direction of data flow An interface may be ID'ed just the same as a class hierarchy. A bubble may be placed at the head or at the end of an arrow to impose the domain (SW/HW) in which a constraint should take effect.

Note: For our simple example, it has been assumed all along that communications incur no costs. However, in practice communications do incur costs. Such costs can be captured in much the same way as bounds for clock speed and number of gates are captured.

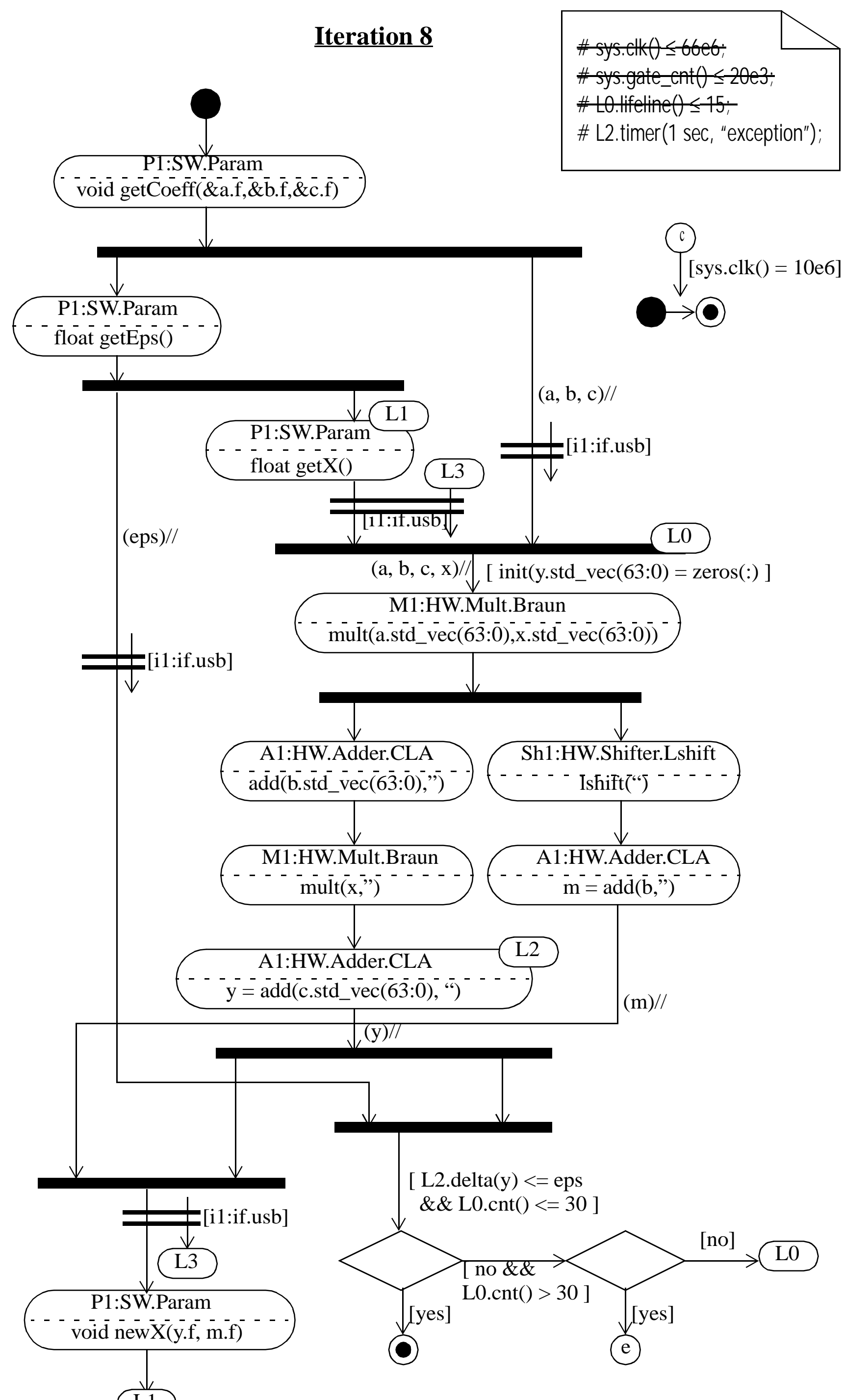


Figure 11: The Final *Activity Diagram*

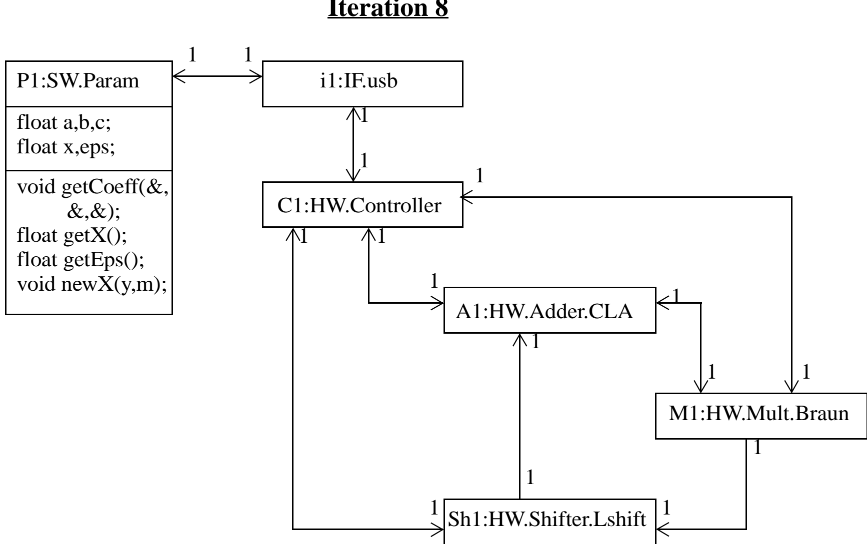


Figure 12: A Corresponding *Class Diagram*

4.0 Summary and Current Work

This paper has thus far presented, in essence, the Unified Modeling Language (UML) approach to realize the constraint-based codesign (CBC). The CBC/UML methodology employs Use Cases to analyze and capture design constraints from requirements. After the final refinement to the Use Case, the Activity Diagram and the OCL-based constraint language transform the Use Case into a more formal model in which most constraints are tightly incorporated and the design is firmly formalized. At the same time an allocation of design functional units may take place. The Neutralization process then unbiases the design of HW/SW preferences before feeding it into the partitioner/scheduler. Communication interfaces are then included and the design hard formalized.

It can be seen from the example that the CBC/UML methodology is capable of capturing a wide range of constraints, behavioral and/or physical, and incorporating them seamlessly into the design. The methodology also provides a high degree of freedom in design exploration within bounds of such constraints.

Libraries of existing HW/SW components are an integral part of the CBC/UML methodology. For the purpose of an automatic code generation, CBC/UML only allows allocated components to be fetched from these libraries. Furthermore these libraries are a collection of inputs for the cost estimation algorithms. Several trifle modifications to the existing libraries may be all that is needed to make them work with CBC/UML. However, to make them work *best* with CBC/UML the existing libraries may have to be converted to CBC/UML-based models. A CBC/UML-based library allows the object-oriented property of UML to work out to its fullest. With such a library, the methodology not only allows a system composition from library components and previously designed systems, but also allows a component decomposition if desired.

(C) Georgia Tech, 1999.

A unified interface representation adopted by the methodology also aids the design greatly. Another obvious benefit is that the methodology supports the design at a higher level of abstraction.

Our current work (for the proposed CBC/UML methodology) is on the following areas:

- Formal syntax and semantics for CBC/UML,
- Development of supporting automation tools & model libraries,
- A cost estimator that will assist in the tradeoff analyses,
- Optimization software for problem solution,
- HW/SW libraries of cores and components (for digital, analog & RF domains).

Acknowledgement

This research has been supported in part by the State of Georgia Yamacraw Research Program in Embedded Software (YES) at Georgia Tech, 1999-2000.

(C) Georgia Tech, 1999.

5.0 Glossary of Current UML Extensions

5.1 Use Case

<<s.t.>> : allows a constraint (also specified in an oval notation) to be associated with a scenario; such that

```
# sys.clk() ≤ 66e6;
# sys.gate_cnt() ≤ 20e3;
# L0.lifeline() ≤ 15e-6;
# exception if hw response
  time > 1;
```

: descriptive constraint specifications are placed within a Note notation. Each constraint always start with a pound sign (#) and ends with a semicolon (;).

5.2

Label:AbstractClass.ImplementingClass : short-handed for **AbstractClass** ← **Implementing Class**

5.3 Activity Diagram

Bubble Label	: A bubble with a label posted within an Activity diagram. This is to provide means for imposing constraints in the diagram.
	: Globalization; this notation globalizes any notation input to it.
c	: Clock
e	: Exception
[init(value) = expression]	: Initializing a value in an activity pointed to by an input arrow.

(C

⟶(L0)	: An arrow to a bubble label means "go to an activity labeled with such a bubble".
(S)	: Parentheses represent an output flow in an Activity diagram, where S is a set of output values separated by a comma (,).
//	: Short-handed notation for a Latch.
variable.type	: A short-handed representation of a data type. *A.i* would read a *variable A of type integer.*
strike-through	: Intranslatable constraints are struck through when they are resolved.
"	: An output value from last activity.
# label.timer(TIME,Action);	: Timer is represented as a method within a bubble class. A timer may be associated with a single or a double bubbles. The TIME argument specifies a period of time before the timer expires. The Action argument specifies an action to take place when the timer expires. An action can be a predefined action (e.g. "exception") or a label connected to a user-defined action.
═╪ (arrow down)	: Interface. It is always true that the domains at both ends of the direction arrow are never the same. If one is SW, the other must be HW.

Note: Although the constraint semantics used throughout this paper resemble those defined by OCL, they represent our realization of how constraints could be captured.

(C) Georgia Tech, 1999.

6.0 References

[1] D. D. Gajski, G. Aggarwal, E. Chang, R. Dömer, T. Ishii, J. Kleinsmith, J. Zhu, “Methodology for Design of Embedded Systems,” *Technical Report UCI-ICS-98-07*, *University of California, Irvine*, March 1998, <see: http://www.ics.uci.edu/~gajski/>.

[2] University of California, Berkeley*, A Framework for Hardware-Software Co-Design of Embedded Systems, POLIS Release 0.4,* December 1999, <see: http://www-cad.eecs.berkeley.edu/Respep/Research/hsc/abstract.html>.

[3] J.A. DeBardelaben & Vijay K. Madisetti, “Hardware/Software Codesign for Signal Processing Systems: A Survey,” Proc. of IEEE *29th Annual Asilomar Conference*, Asilomar, October 1995.

[4] University of Cincinnati, *The UC KBSE VSPEC Homepage,* December 1999, <see: http://www.ececs.uc.edu/~kbse/projects/vspec/>.

[5] The Object Modeling Group (OMG), *OMG Unified Modeling Language Specification Version 1.3,* June 1999, <see: http://www.omg.org>.

[6] P. Alexander, R. Kamath, “Rosetta Facets Semantics Strawman,” *The System Level Design Language,* December 1999, <see: http://www.inmet.com/SLDL/semantics/semantics.html>.

[7] J. Ellsberger, D. Hogrefe, A. Sarma, *SDL Formal Object-Oriented Language for Communicating Systems*, Prentice Hall Europe, 1997.