Elements of System Design

Classification and Organization

TATA CONSULTANCY SERVICES

Authored by : Hrishikesh Sharma Date : August 20, 2004
Subramanian P S

Reviewed by : Subramanian P S Date : August 24, 2004

Reviewed by : _____ Date : _____

Elements of System Design

Printing the document

To print this document optimally, take colour print outs of pages 12, 13, 19, 23, 25, 28, 30, 34, 35 and 42. All other pages can be printed in black and white.

Viewing the document

Pages 42 is best read on screen with a magnification of 125% or higher. Other pages can be read at normal zoom.

Scope

The document reviews the ontological aspects of majority of terms used in systems' design community. It proposes a semantic framework that encapsulates the fundamentals of heterogeneous system design. The document then goes on to describe the elements of system design process from the purview of this framework.

Contents

List of Figures

List of Tables

List of acronyms

ADL	Architecture Description Language
ASIC	Application-specific Instruction Processor
ASIP	Application-specific Integrated Circuit
CAD	Computer-aided Design
CDFG	Control-data Flow Graph
CFSM	Communicating Finite State Machines
CPU	Central Processing Unit
CSP	Communicating Sequential Processes
DFA	Deterministic Finite Automata
DFG	Data Flow Graph
DSL	Domain-specific Language
DSP	Digital Signal Processor(Processing)
FSA	Finite State Automata
FSM	Finite State Machine
GPL	General-purpose Language
IP	Intellectual Property
MoC	Model of Computation
NFA	Non-deterministic Finite Automata
OS	Operating System
PCB	Printed Circuit Board
RPC	Remote Procedure Call
RTL	Register-transfer Language
RTOS	Real-time Operating System
SOS	Structural Operational Semantics

UI	User Interface
UML	Unified Modelling Language
VHDL	(Very High Speed Integrated Circuits) Hardware Description Language
VLSI	Very Large Scale Integration

Document Organization

The document is organized as follows. A section of introduction brings out in detail, the purpose of our work. It is followed by reviewing an operational classification of systems, which is to be used later. Here, the concept of system modelling is introduced gradually. Elements such as model of computation, its constituents, relationship to languages and themselves, as well as an example operational semantics is introduced. Now, the design process is elaborated using the new systems modelling concept. Factoring for optimization, heterogeneity(the major treatment) and details about design methodologies is introduced. The design elements such as analysis, synthesis, refinement and abstraction are treated in the same section. Finally, the emerging fields of hardware-software Codesign and Reconfigurable Systems' design are explained using the new model.

1 Introduction

Systems have been employed by both nature, and humans, to perform specific functions. There is a sense of ***order*** in all natural processes, which one can ***observe***; even when there exist apparently random, or chance events. This is because of the tendency to fill in our lack of understanding of nature, with explanations that tend to replace noise with meaningful pattern and effects with probable causes. Earth's climatic system, human immune system etc. are examples of natural systems(otherwise known as physical systems), which exhibit a specific *pattern of behaviour*.

Human race itself has been involved in creation of systems, mainly as aids for the day-to-day work it has to carry. Starting with simple mechanical systems such as a crowbar and the flint(stone)-based spark ignition, the evolution of such systems has come to a point where they can be as complex, as their natural counterpart. Internet is an example of such a communication (network) system. Such systems are also frequently called *Artificial Systems*, mainly to underline the fact that they are man-made.

Structurally, artificial systems are a collection of objects called components, or subsystems. The components can be **heterogeneous** in nature, and their interaction may be regulated by some simple or complex means. Note that it is only (complex) artificial systems, which *require* a structure; for providing a structure(divide-and-conquer) is the human's way of dealing with complexity.

Human-made systems can be classified into many categories; prominent amongst them are electronic systems, power systems and mechanical systems. This document focuses on electronic systems, or (embedded) electronic component of a complex system, such as the *information processing systems*. Any reference to the term, system, has also to be taken as an artificial system. Electronic systems are getting more prevalent, because they demonstrate richness of functionality, superior performance, safety/reliability and low-cost aspects.

Microelectronics domain, especially VLSI, has had the "more specialized/guided/formal" set of design methods so far. The design of microelectronic systems consists of realizing(synthesis) the desired functionality(using circuits etched on a wafer), while satisfying certain design constraints(optimization). Optimization is done to maximize certain system quality(qualities) for competitive advantages. Such optimizations lead to *design tradeoffs;* some quality is gained on the expense of the other. Computer-aided design methodologies, which have been successfully used in reducing design times and performing optimization for integrated circuit designs, can also be applied/scaled for system design purposes. The terms "design space", "design trade-off", "design space exploration" etc. have been carried over from the IC design domain into the *broader* system design domain.

There is been at least one comprehensive work on similar area already[22], and few useful tools and frameworks based on that(PTOLEMY, POLIS) are already being in use now. Though it covers a lot of "what all exists", the authors feel that there is still a scope of debating, "why all the gamut exists", or in other words, ontology. That is the question, which is tried for treatment in this short document.

There have also been papers pointing out the disparity in the various definitions floating about many of the terms used by design community. Also, there have been specific papers, which have tried to formally address semantic definitions of these, in order to bring precision. Our approach is novel in the sense that we propose a generalized semantic framework that encapsulates all the aspects of heterogeneous system design; and hence provides opportunity

to look into all possible design terminologies, with it's aid. Going a step further, we propose a way to use this framework for all system-related design, analysis and verification tasks.

2 Systems Classification

Much of the background established in this section deals with *information-processing systems.* Information processing systems are automatic, electronic machines whose function is to generate a **set** B of output information items(e.g., the results of computation on the data), from some **set** A of input information items(for example, data, and at times, a program representing a computation in some form). The mapping is formally represented by a function F : A $\rightarrow$ B.

Before we move into describing our modelling framework, it is important to classify the information-processing systems. Different elements of the framework will be used in the corresponding design methods for these systems. A traditional classification provided in [14] is detailed out as follows.

2.1 Transformational Systems

These systems take a body of input data, and transform it into a body of output data. Generation of the correct result is the primary concern of these systems, and hence issues such as program termination gain importance. Initially, the term information-processing systems used to refer to these systems only.

A continuous-time processing view of these systems is that they have all inputs ready when invoked and the outputs are produced after a certain computation period. After the computation, the system halts, till the next set of inputs is provided. Hence, the model of such system is required to specify only the functional/transformational aspect.

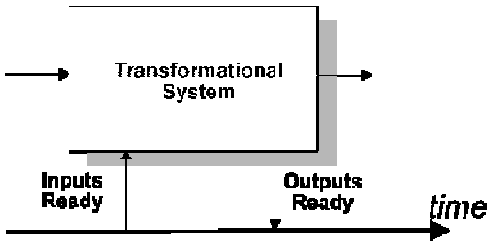

Figure 1: Transformational Systems

2.2 Reactive Systems

These systems are repeatedly prompted by the environment, and their role is to continuously respond to external inputs, at the speed required by the environment. Such a system never has all its inputs ready--the inputs arrive in endless and perhaps unexpected sequences. Hence, in general, it does not compute or perform a function, but is supposed to maintain a certain ongoing relationship, so to speak, with its environment. Such systems do not lend themselves naturally to descriptions in terms of functions and transformations, making it virtually impossible to write a transformational program for them. However, meta-models such as tagged-signal model[25] exist for reactive systems, for a purpose described in section 3.3.2.

A model of such systems thus requires specification of behaviour, which includes both the functional and timing aspects[8]. It will be clearer later(section 3.5), that such systems can also utilize temporal logic expressions to specify timing constraints in the model itself.

In fact, most controllers(control-dominated systems) are by definition reactive, not transformational, with application domains ranging from process control, military, aerospace, and automotive applications to signal processing, ASIC design, medical electronics, and similar embedded systems. Real-time systems also get classified under reactive systems.

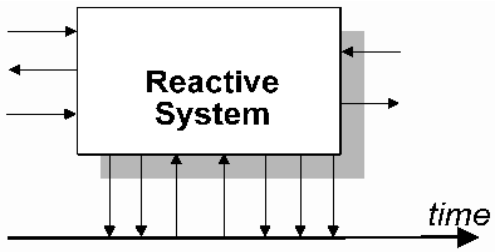

Figure 2: Reactive Systems

2.3 Interactive Systems

These types of systems use additional inputs from the user, to determine the output. This makes such systems as an extension of the classical transformational systems, and an old definition[27] merges these into transformational systems. Unlike reactive systems, such systems react with the environment at their own speed(respond to the user only when they can), gathering more inputs, and also providing control(to user) to manipulate the sequence of steps towards eventual output, which the user wants. Such finer distinctions make them a separate class than both transformational and specially reactive systems(unlike few authors, who treat reactive and interactive systems as functionally similar).

Graphical user interfaces and operating system are examples of such systems. These systems are also frequently termed under human-computer interaction systems, though the user need not be a human being.

This document will not focus on interactive systems.

Few other categories, such as concurrent, distributed and parallel systems are introduced[12] while dealing with systems that *necessarily* exhibit some kind of concurrence in their computation. We do not use any such distinction; and hence we omit the details of these as well.

3 Systems Modelling

Much of the usage of models can be attributed to usage of *formal methods*. A formal method in system development is a method that provides a formal language for describing an artefact (e.g. specifications, designs, prototypes, to be introduced later) such that formal proofs are possible, in principle, about properties of the artefact so expressed. The benefits of proving, for example, that unsafe states will not arise, can justify the cost of using such approach.

It will be clearer in section 4.5 that formal proving(known as formal verification activity) needs the artefact description to be *rigorous*. While formal methods can never dislodge the requirement of gathering the system requirements from general users in a non-formal way, the design process can benefit if these informal and hence non-rigorous requirements were converted into a formal specification. Defining what a system should do, and understanding the implications of these decisions, are one of the most troublesome problems in design of complex systems. Hence, usage of formal methods here also, can have major benefits. In fact, practitioners of formal methods frequently use formal methods solely for recording precise specifications, not/not just for formal verifications.

As we can sense from the definitions of the various classes of systems, not many systems are simple enough to be directly(formally) represented as a function mentioned in the beginning of section 2. Similar to a given non-linear transfer characteristic, and design of a corresponding circuit required; the design process of such systems becomes complex. In such cases, the power of heterogeneous systems' modelling is used to assist the design and verification process. Such models are often captured using particular *model of computation*.

3.1 Motivation for Model of Computation

Different papers have touched upon the definition of model of computation: [17], [19], [22], [24], [25] and [26]. None of these deals with the historical perspective of it. To understand that, we need to understand it with respect to the notion of computation.

3.2 Computations and Computational Models

Informally, an algorithm(for a function f) is a finite set of instructions which, given an input x, calculates and yields after a finite number of steps, an output y = f(x). When these steps include arbitrary branching and looping steps to control the formation of output, then the algorithm implies a computation rather than a calculation. The algorithm must specify how to obtain each step in the calculation/computation, from the previous steps and from the input. The algorithm may only yield a partial function[1]. An algorithmic partial function, which is defined on all arguments(i.e., which is total), is called computable or effectively calculable function.

Certain classes of computable functions can be obtained using certain types of steps/procedures/operations; these are the computational models corresponding to those classes of functions. For example, to specify the motion functions(trajectory etc.) of a ball in a Euclidean space, analysis was carried out, and a Newtonian model of computing these functions(the type of steps are the laws of mechanics here) was born. Another example could be the way arithmetic as a model was born: various types of counting

[1] Hence, non-deterministic for certain inputs.

systems(integers/rational/irrational etc.) can be modelled using same steps(brackets/division/multiplication/subtraction/addition).

Thus, these models define a framework for **composing** certain types of computations(global actions)(as a sequence of steps, or local actions), much in the same way as axioms are combined in proving theorems. The composition carries some semantics, and hence the framework can only encompass those compositions, whose semantics is expressible. Such a restriction eliminates the possibility of arbitrary compositions: something into which the designers are never interested. Further, the composition style can have space-time flavour. If the local action(steps) are combined in time(one after the other), then the model of computation has a sequential nature(e.g., Von Neumann model). In contrast, If the local action(steps) are combined in time(one after the other), then the model of computation has a parallel nature(e.g., CSP).

3.2.1 Power of Computational Models

These models can be classified[2] according to their computational power, i.e. how large is the class of functions and languages they can cope with. Historically, the initial work done around Turing's time was to figure out a model with "universal power", i.e. model for universal computation. Many different people came up with different models for universal computation at the same time: notable among them are Turing's Machine, Recursive Functions and Lambda Calculus. The same people also proved the equivalence and inter-convertibility of all of these(Hence the name, Church-Turing Thesis), something that is at the core of heterogeneous system design methodology, to be introduced later.

But to perform, or rather model and understand computation behind other kind of scenarios(such as many computers in a room, doing coordinated computation such as pointer-jumping), the sequential model of computation fell short. Hence *distributed computing* was born, which is a different model of computation, and different powers. Similar example can be taken in *quantum computation*, arising out of needs of neural systems' modelling.

Further, these models can also be characterized by their behaviour, or the class of computable functions they represent. Such representation of steps lends them to a domain of application for computation(an example being signal processing, for data flow model of computation). The introduction and usage of *domain-specific languages*(introduced in section 3.5.1) is an outcome of development of such models.

Thus, to summarize, a model of computation is a domain-specific(or domain-independent), often-intuitive understanding of how the computations(in that domain) are done. They arise out of attempts to understand certain physical phenomenon(such as ball-motion), or attempts to use computer in various applications. The components of model of components will be clear in the following sections.

3.3 Relation to System Models

The above definition of model of computation can be concretely *instantiated*, when looked from the angle of various **types** of systems. It is easy to analyze it first from the perspective of transformational systems, such as a compiler.

[2] Chomsky's hierarchy is an example of such classification.

3.3.1 Modelling Transformational Systems

A formal design of system starts from formally specifying the desired behaviour(informal designs such as that used in certain software development communities need not follow this paragraph). To describe system behaviour with a level of precision, we need to think of the system as a collection of simpler subsystems, or pieces, and the method or the rules for composing these pieces to create system functionality. Example of this is the arithmetic model: it deals with a set of objects called integers etc., and operations become the composition rules. Such a specification *models* the desired behaviour. A model is a formal system consisting of objects and composition rules. Thus, whenever a transformational system is tractable, it can be represented as a *composition* in a suitable model of computation. Most simple systems of this kind need only one model of computation, and a representative language for specification of all their pieces.

The notion of composition physically manifests as interaction pattern between objects. In real life, a model of computation typically imbibes sense of concurrency, sequential behaviour and data communication, to name a few types of interaction patterns amongst (computing) objects. Note that interaction patterns noted here are properties of computational models pertaining to system design only; computational models such as arithmetic(used to express calculations) need not exhibit such patterns. Henceforth, we will be using the term *model of computation* only in context of system design.

3.3.2 Modelling Reactive Systems

Though the primary concern of reactive systems is to react to the environment with the speed of it, according to us, even the simplest systems such as an alarm clock has an element of computation hidden behind it(such as the counter). It is that computation, which is functional, and hence tractable[3]. Perhaps for this reason, not only we need to model communication behaviour of such systems, but also the functional behaviour[17]. By considering time itself as an input(or output) parameter[4], it is possible to depict certain reactive systems as a transformational system.

In any case, a *functional specification* for such system is represented as a set of components[5], which can be considered as isolated monolithic blocks, interacting with each other and with an environment, which is not part of the specification.

Because of the complexity of (reactive) systems prevailing now-a-days, it makes sense to use domain-specific models of computations to specify sub-behaviours only. Such systems are termed as heterogeneous systems. We will be focusing on heterogeneous system design, section 4.2 onwards.

It is important to differentiate between models used to represent functionality, with the models used to represent intermediate design stages of the system. The models there tend to be successively more detailed in terms of *implementation information*, and the visibility of "which computation" gets subdued. Perhaps the worst confusion arises from the usage of the following terms: abstraction hierarchy of models, synthesis mapping and refinement mapping. A more *generalized* model covering all such definitions, with hopefully more clarity, will be introduced in the remainder of this document.

[3] Models of computation such as CSP and CFSM observe this "local computation" phenomenon; see section 4.6.
[4] E.g., Synchrony Hypothesis
[5] Simple systems such as 8051-based timer used in alarm-only device(hypothetical) may just have a singleton set of components

3.3.3 Modelling Interactive Systems

Interactive systems also have their idiosyncratic behaviour. Some categories of behaviour commonly observed in interactive systems are: performing an activity in a repeatable way, performing a set of activities in *any order* assuring that each activity will be performed once(e.g., clicking of start/stop/refresh buttons of a web browser), etc. Also required for specifying models is way to capture the dataflow between users and the various elements of the system[6] itself.

In modelling interactive systems, visual formalisms have been observed to reduce the gap between users and analysts. Object-oriented methods like UML offer one of these formalisms[7]. Especially in UI design, UML or the object-oriented methods are now quite prevalent. Even Petri-nets have been quoted as usable model of computation for interactive systems[36].

3.4 Using Model of Computation

Given that a model of computation has primitives to compose functions, one needs to understand how it is used in the process of system design. Let us understand it *first* from the view of simple systems(or a particular view of system), ones that will *not require*[8] more than one model of computation for detailing, due to their homogeneity.

3.4.1 Enumeration of Behaviour

A typical system *design task* is an enumeration of the behaviour instances that it is supposed to produce. It the set of behaviours carry some structure, the term *behaviour space* can be used in the context.

Given a behaviour space, one can determine the suitable model of computation which denotes a particular kind of behaviour, and hence can be used for, e.g. by formal verification or simulation-based verification to check the consistency of the design. The behaviour can be deterministic, or non-deterministic in the automata-theoretic sense. Even the non-deterministic behaviours can be captured using formalism of model of computation; such capturing is further elaborated in [25].

Behaviour is defined as the unordered set of behavioural units(extensional definition). When the input and output spaces are defined by a set of (important) variables, then a (determinate) behavioural unit is an ordered tuple of values for the input and the corresponding output (vectors). This is the operational view of behaviour. The set of all possible behaviour units denotable by a model of computation is the corresponding behaviour space[9]. When the system is physically excited with a particular input vector, it generates a *simulation run* for the system. Such a run yields an element(point) of the set of all possible behaviour units of the system[10]. Note that such variables should adhere to observability criterion(some authors denote

[6] The destination of dataflow tends to be different, for each dialogue between the user and the system.
[7] Harel's StateCharts is the model of computation behind UML
[8] There are times when same system(view) can be represented in two or more models of computation; in one, the representation is succinct, while in the others, not. We are ignoring this detail here; though it helps to make a succinct model in general.
[9] Only if the units show some structure, the points can be organized in some space. E.g., model of computation for signal processing can imply a linear space of behaviour.
[10] A system will typically implement a part of the overall behaviour space of the computational model, which need not be a sub-space. If the behaviour is determinate, it will have an alternate functional representation.

behaviour in terms of internal state change, or a snapshot, of a system, which is defined but not observable for all types of systems[11]).

A denotational view at the term behaviour tells that it is a view, which captures both the function and the timing aspects of any system(already pointed out in section 2.2).

3.4.2 Providing Specification

One can also represent the desired system behaviour as a formal specification(for computation) within the model of computation. For details of specification languages, see section 3.5; at this point it is sufficient to say that such a formal representation requires a (formal) specification language. A sentence[12], or a syntactical structure[13], in a specification language represents a set of behaviour units(in other words, subset of the power set of behaviour units denotable by a model of computation). Hence, a set of sentences representing the system behaviour, actually imply set of such subsets. A *functional* input-output *relation*(behaviour), when specified using some language, is known as the system functionality.

A misnomer for behaviour is the algorithm, and to the extent, specification[17]. Actually, an algorithm is outcome of an implementation decision, and hence cannot represent either. Our definition of behaviour and specification makes it clear that using algorithm is indeed a misnomer. As another contradiction to above statement, sometimes the algorithm(for system) is standardized, in which case behaviour and specification get (reverse-)derived from the same!

Formal specification languages are based on mathematical/logical semantics, and hence the sentences of specifications in them can be used to reason about(validating) *extensional* properties of the system, or the behaviour. Like the (truth) *satisfaction* criterion of logic formulae, a specification covers a system, if there exists a *satisfaction relation* between the set of sentences of the specification, and the set of expected behaviours.

3.4.3 Providing Structure

One can construct physically, an implementation[14] of the system, by providing a structure consisting of (countable) sets of components of various types, and their interconnections as well as interaction patterns(e.g., scheduling strategy). While the specification tells how the system will work(behave), the design process is incomplete till it is described, how the system will be manufactured. Many-a-times, especially for simpler systems, the design process is interchangeably used for the implementation (sub-)process, which is a loose usage. An implementation of a component, or an artefact, subdues certain information about it, such as design constraints, standards, performance goals, etc., which are part of the design process.

While we discuss the difference between the terms, *architecture* and *implementation*, in a later section, it is important to state that implementation within a model of computation does not

[11] E.g., the state of an executing program over a processor is supposed to be the snapshot of all its variables/registers and the program counter.
[12] Abstractly represents notion of equation between two sets of variables, whose solutions are points in the behaviour space.
[13] Similar to notion of a non-terminal in context-free grammars
[14] We will differentiate between architecture and implementation in a later section: a difference that is important, but can only be understood while dealing with multiple models of computation.

necessarily imply an "executable" artefact[15]. This is because the (high-level) structure may not have all the details required for execution.

A usage diagram for model of computation can be viewed as follows.

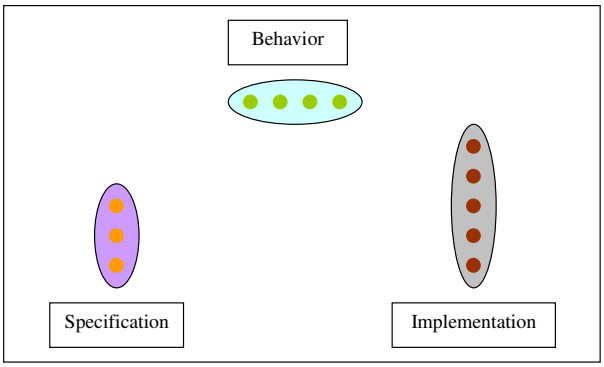

Figure 3: Using Model of Computation

3.4.4 An Example

Finite-state automata[2](the corresponding languages being *regular languages*) is a typical example of a model of computation, used in design of control-dominated systems, and also components such as user interface.

The behaviour space for this model of computation is the set of all regular sets $\beta \subseteq \Sigma_k^*$. Σ_k is the k-element alphabet for input/output. β is a particular collection of strings formed out of Σ_k^*, hence the behaviour units involved here the strings itself. The behaviour of an individual FSA is thus a regular set.

The specification of an individual FSA is done via the notion of *language*: the language describes the *intension* of the regular set in some way. For example, { w: w consists of equal number of 0's and 1's }, and 01* + 10*(the regular expression notation) are two different (types of) specifications for automata.

The structure, or the implementation of an FSA is provided by the five-tuple notation: A = (Q, Σ, δ, q_0, F). Visually, it is also represented as a graph. The structural elements of the FSA implementation are the states and inputs symbols. There can be multiple implementations of a FSA: DFA, NFA and ε-NFA are equivalent implementations. A more detailed structure can be captured in the Esterel-Lustre common format, OC.

The usage diagram for the FSA thus looks as following.

[15] Unless one uses an interpreter.

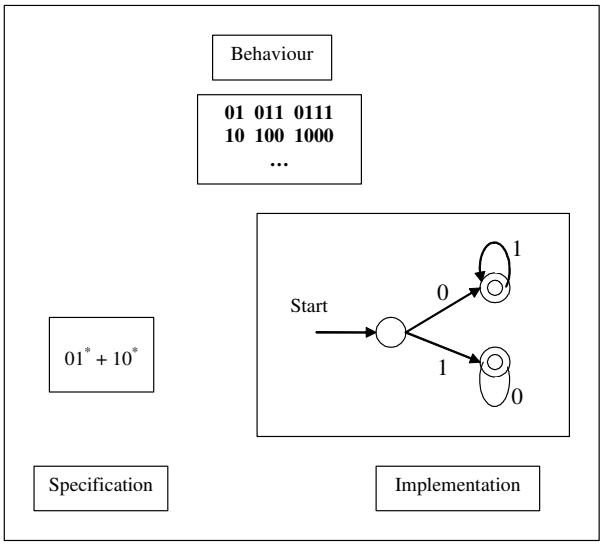

Figure 4: Using Model of Computation for FSA

3.5 Languages and Models of Computation

To do formal specification, modelling(or specification) languages are required in each model of computation. But the scope of what can be specified using a language need not be restricted to elements of single model of computation. Some good discussion on languages and models of computation can be found in [17], [24].

It is important to distinguish between programming languages and specification languages. Programming languages have been conceived primarily to express algorithms, which, as said earlier, is an outcome of implementation decision. Hence, programming languages should be seen as a tool to describe *implementation* details. In terms of generality, programming languages support many models of computation, and hence sometimes be classified into a separate, heterogeneous model of computation[24]. Specification languages are used to capture *a set of* properties of the system; for the model to be able to represent, what is to be designed for. Hence, not only specification languages focus on providing aid for specifying one or more properties of the system(such as temporal aspect of behaviour), but also they can be abstract and incomplete. This is perfectly valid, for the purpose of modelling is to look at useful abstractions of a system. We will focus here on specification languages.

The semantics of a language maps a given(syntactically correct) specification into a point of the behaviour space corresponding to the model of computation. The mapping may not be surjective: it depends on the level of abstraction, or the degree of incompleteness, that the specification language exhibits. This degree is captured using the term *expressiveness* of a language. Expressiveness implies which behaviour units can be described using a particular language. Intuitively, there can be preorder relation between the languages based on expressiveness[15]. Such partial order of languages is denoted by language embeddings. It is such partial order, which is exploited while doing design (process) using a set of specifications.

3.5.1 Domain-specific Languages

A *domain-specific language* (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain. Hence, in different domains, more succinct models can be designed.

The initial work on models of computation was mostly domain independent, i.e. universal model of computation. Scott introduced denotational semantics, and it resulted in domain-specific models of computation. Correspondingly, the new-generation languages such as SILAGE differ from the GPL such as 'C'.

Because these languages are declarative and hide much of the implementation details, some of them can be considered more as specification languages than programming languages. Hence, one of the (many) important advantages of these languages is that they enable more properties about programs to be checked. In contrast to a GPL, the semantics of a DSL can be restricted to make decidable some properties that are critical to a domain. For example, Unix command *make* reports any cycle in dependencies and thus totally prevents non-termination (assuming the individual actions do not loop).

3.6 Ordering using Expressiveness of Models of Computation

Just like the expressiveness of language can form a partial order, a set of models of computation may themselves be partially ordered.

The "order" relationship gets formed due to the notion of relative computational power(defined in section 3.2). The computational power, i.e. what functional aspect and what temporal aspect a model of computation can cover(we are looking from system design perspective throughout) can be understood(if not a quantitative comparison) using denotational and other semantics of the model of computation[17],[22] and [25].

A reflection of such order can be seen in the so-called abstraction-based hierarchy of models of computation(an example being the stages of design flow in hardware-only system, or digital circuits', design). Though that hierarchy is based only on the level of implementation details which can be captured in models of computation, it will be clear in section 4 that even specifications can be refined. This is possible only when the model of computations involved have a known subset sequencing[16] in terms of computational power($MoC_1 \subseteq MoC_2 \subseteq MoC_3 \subseteq \ldots \subseteq MoC_n$).

At times, the order of subsuming is not very evident, or perhaps cannot be quantified. In such cases, analogies can be used to approximate the order. For example, differential equations(model of computation for analog domain) which govern a sequential circuit may not be comparable to finite automata. But, differential equations can be approximated into difference equations, which drive the design of an FSA for the sequential circuit. Hence, they are comparable. Of course, analog equation subsumes finite automata, due to additional semantics(there can be non-linear differential equations).

Diagrams such as Gajski and Kuhn's Y-state diagram, and their extensions become a specialization of such a formulation(a planar diagram can be developed for the order mentioned

[16] Note that only reflexivity and transitivity are required from the relation: not the anti-symmetry, and hence the sequencing forms a preorder rather than partial order.

above). Once we introduce the mapping functions amongst the three views of usage, as depicted in Figure 3, in section 4, this statement will become intuitively clear to the reader.

Interested people can also look into meta-models of computations, such as X-framework[11], or the tagged-signal model[25]. Such meta-models form the basis of comparison of models of computation. A category-theoretic formulation will be introduced in section 4.3.4.

3.7 Operational Semantics and Model of Computation

To be able to specify runtime behaviour of a specific model, the following aspects can be used:

- Nature of components from which the system will be constructed (e.g. cyclic, sporadic, protected, passive)[17]

- Scheduling paradigm under which the system is executed and the associated mechanisms(e.g., interleaving or non-interleaving semantics).

- Means of communication between components/objects (e.g. mailboxes)(not required for SOS)

- Means of synchronization between components/objects (e.g. clock edge, semaphores)(not required for SOS)

- If applicable, the means of distribution and inter-node communication (e.g., shared memory, RPC)(not required for SOS)

- Means of providing timing facilities(clocks)(not required for SOS)

- Means of providing asynchronous transfer of control(interrupt controlling)(not required for SOS)

This list is only suggestive; there is no general method to define the semantics(of any kind) for any model of computation.

[17] E.g., intensional definition of sets in E-R model of computation for databases.

4 Design Using Formal Models

A significant application of formal models in designing systems arose from the semiconductor design area. There has been evolutionary change in the functionality packaged in an IC. Previously, there used to be a lot of printed circuit boards(PCBs) doing various individual functions. This required a lot of hardware integration skills. But as the Moore's law defines the temptation of chip making companies to pack more and more electronic functions on the same chip, the world within the chip moves from just being a computational component, to a complete system. Such an evolution brings the idea of dealing with models and synthesis, which come from hardware domain. Thus, the digital circuit design for VLSI is moving into what is known as digital systems' design, with integration level still as high. This is what is known as the system-on-chip design movement in semiconductor area[18]. However, it is not that all components get inside one chip. It is the system design process, which takes such architectural decisions.

We will first try to look at designing systems using one model of computation. By this, we imply the dominant view of a system, which can be one, such as FSA in an FSA simulator. Note that the *structure* we come up with is not the final implementation. For that, refinement is required to make it either a hardware or software piece, something, which we introduce in later sections. For notational purposes, we will call such systems *simple*.

4.1 Simple Design

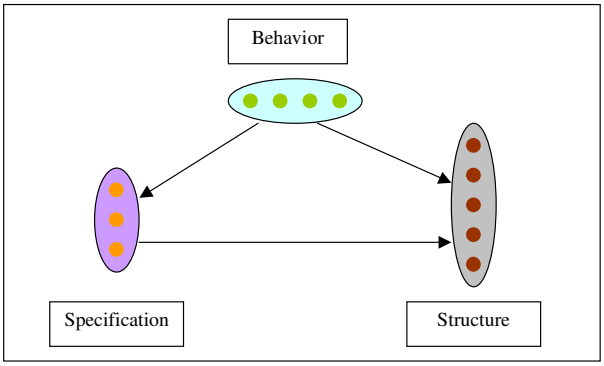

Figure 5: Mappings within Model of Computation

Mapping a specification into an implementation, or structure, within a model of computation is known as synthesis step[19]. It is also possible to map(at least in few cases) a set of orthogonal behaviours, of a behaviour space, into the required structure of the system. For example, using the *Shannon's expansion theorem*, and using multiplexers, complex logic circuits can be built

[18] Also to note that applications of system design methodology is not restricted to chips only; embedded systems as simple as a digital wristwatch can be designed using this. In fact, Esterel, a language to specify reactive systems has been already used in design of digital wristwatches.
[19] We reserve a term, "heterogeneous synthesis", to be used in context of heterogeneous system design.

inductively from 1-variable systems, using finite number of synthesis steps. Another example of such atom-level synthesis is the usage of (trivial) state-walker design pattern. A computation function can be designed for each such atom(can be FSA/procedure/circuit), and notions of combining them in parallel(using ε-transitions/switch-case/multiplexer) be used to make them all running together. Such a synthesis in most probability will not be optimal: element-by-element design may ruin the cost requirements(see next section).

4.1.1 Cost Functions

No structure can be designed, without having certain cost constraints as requirements. We quote here a few examples.

- Transformational systems such as compiler may not have run-time specified as cost, but most reactive systems will show stringent run-time costs.

- A software processor(running a RTOS) can be characterized from the performances point of view by its utilization rate (acceptation level can be less than 80%).

- A communications node such as a bus can be characterized by its throughput, a shared memory by its read/write latencies.

- Code size is another cost, which is measured in case of automatic software generation.

- Clock cycles taken per instruction execution can be taken as cost for a microprocessor.

- The cost can be *business*-driven: life-cycle cost(time-to-market), monetary cost(product pricing), speed, reliability, size, weight etc.

It is imperative, that the costs need to be captured during the time of specification. The specification process captures primarily the properties of computation involved(can be used for formal verification), and secondarily, the cost properties(sometimes as *constraints*). The design process, hence, also needs to analyze the cost of a particular structure.

For this purpose, we introduce two domains in our diagram: the evaluation and the approximation domain. These domains can be extensionally defined as sets of integers, reals etc, and the cost functions become the mapping to such domains. If a cost is measurable(the domain of the mapping observes either a total order, or a lattice), then the functions can be *evaluated*. In a particular model of computation, certain cost can just be approximated(such as *delay* calculation in a DFG). In such case, the actual cost can mostly be evaluated(e.g., *delay* calculation after a gate-level model evolves from DFG, post scheduling/binding), when sufficient *implementation details* are available in another model of computation, into which we have *refined* our structure(see section 4.3.2).

Thus, we introduce side functions to these domains: they can be performance evaluation or constraint[20] analysis. The model is analyzed to derive value for these constraints. One might need to look at *intensional* aspects of the domain of the mapping(say, for property verification), or at the *extensional* aspects(say, performance evaluation). There can be secondary parameters, rather the just properties of computation, which give rise to these costs. Implicit overheads of event handling, access time to driver are examples.

[20] The constraints can be min/max delay constraints, execution rate constraints etc.

[21] provides an example treatment on constraint specification and analysis. Recent frameworks such as METROPOLIS also provide support for this, explicitly.

The evaluation domain can also have members as properties that the design must specify. For example, presence of a dead state in an FSA points to something wrong in the synthesis algorithm. These are the condition evaluations; i.e. the conditions, which are supposedly true for some model expressing a type of computation.

Abstract interpretation is one of the ways of realizing the approximation functions. Once approximated, techniques such as model-checking(refer section 4.5) can be combined as well, for property analysis[37].

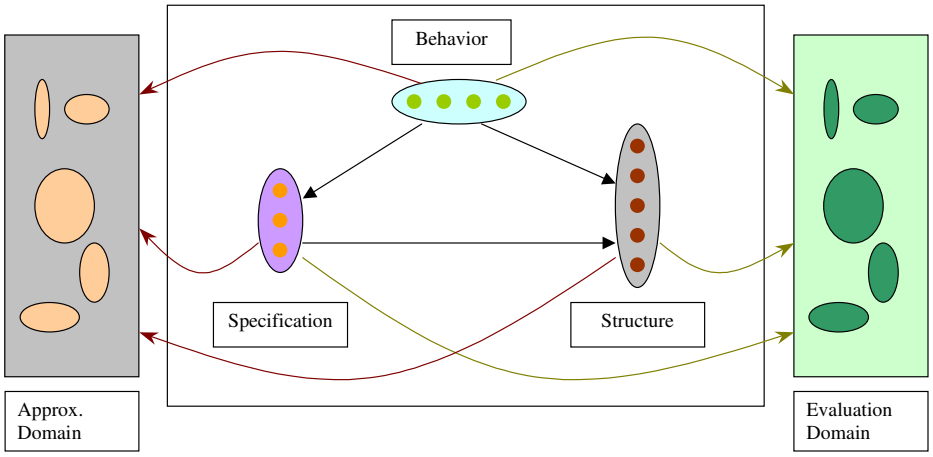

Figure 6: More Meaningful Structure of Model of Computation

4.1.2 Sources of Cost

The problem can also be thought of describing the *domain* for the cost *function*. The cost is calculated for a particular implementation, or a structure; though specification/behaviour analysis can also provide approximate costs, or bounds on the (range of possible structures).

Irrespective of the model of communication, the major *quantifiable* resources constituting any information-processing system are *processing* power, *communication* bandwidth and *storage* capacity. This is because these are terms are also embedded in the notion of computation. Their representation is known as the info-action triangle[21][10].

[21] The term **info-action** is used to denote any of the **info**rmation handling **action**s listed above.

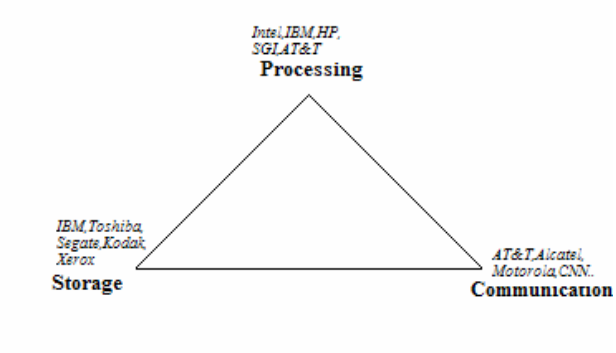

Figure 7: Info-actions and companies built upon them[22]

Let us take a simple, sequential machine system. The processing power is representative of the power of individual elements: e.g., 2-input AND gate has less processing power than 3-input AND gate. The communication power manifests as the bandwidth of internal busses. The storage manifests as the capacity of registers.

As another example for a *non-trivial* data-flow graph[3], in which the processing and communication resources are obvious, but not the memory resource, it can be found represented in the loop-back edges of the graph.

Thus, these terms are polymorphic; in various models of computation, they arise under different guises. A few exceptions are, for example, a resistive network. Though memory is present in terms of parasitic inductance and capacitance, it does not play significant role in the computation represented.

We conjecture that these resources are *implicit constituents* of any known cost function, in any model of computation. As an example, the communication aspect of a system/computation impacts the wire/bus/protocol cost(in a distributed system) of a particular structure. A high-end processor may impact the monetary cost, simply because it is very expensive. A memory-intensive design can impact the reaction speed of a system to a stimulus, simply because i/o operations tend to take more time[23].

As noted in [10], the costs associated with these resources are functions of the "state of the art" of the core technologies associated with these areas. Since this "state of the art" is continuously changing due to the advances in *scientific understanding* and *technological innovation*, it is clear that these costs themselves are a function of time.

Since these constituents impact cost, one would ideally like to minimize each one of these. They cannot be but minimized to zero; the *lower-bound theory* ensures that they can be designed to meet time/space complexity asymptotically. As an example, the Myhill-Nerode theorem tells that there need to be a minimum number of states in a FSA designed for a regular language. This impacts the size of memory, which is correspondingly required for

[22] Adapted and extended from *Wavelet Analysis*, *The scalable structure of Information*, H. L. Resnikoff and R. O. Wells Jr., Springer 1998

[23] The invention of B-trees as data structure for efficient search was driven by the fact that secondary storage devices, on which huge amount of data can be stored(required for certain database applications), are slow in responding, and hence disk i/o operations needed to be minimized.

computation. As another example, in a plastic cell configuration for Reconfigurable computing(see section 4.9), the processing can be just as less as lookup table function.

The three resources mentioned above, though conceptually independent, admit the existence of techniques, *which allow one resource to be traded off for the others*. This is quite important from an economic point of view, since *at a given point in time* the relative costs of the resources **do show** a large variance. Supposing we are in need of a resource B which is costlier than A and C. In such a situation, it may be economical to use A and/or C in conjunction with techniques which trade off B for A and/or C.

Tradeoffs are a way of life in doing *optimal designs*: there are resource tradeoffs not just to optimize a single property, but also there are tradeoffs to optimize a property at the expense of other. Hardware literature gives a nice picture of such optimizations: an example being the delay-area trade-off(arising out of a lower bound on area-time complexity).

4.1.3 Choices during Design

Most of the arrows in Figure 6 imply one-to-many mappings that lead to availability of choices during design process.

Since the behaviour is fundamental to a model of computation, it can be captured by more than one specification in the same model. For example, 0^n1^n and {w: w has equal number of 0s and 1s} are essentially the same specifications for some language. Hence, the relation between behaviour and implementation is a one-to-many relation.

Similarly, a specification may have many correct implementations. Such an implementation is said to satisfy the specification. The reason driving many correct implementations is the optimality of solution, given the constraints. There can be many optimal solutions; such a phenomenon is known as Pareto Optimality in design community.

Finally, behaviour to structure mapping is one-to-many. The notion of model of computation for the overall design can be intuitively described as composing the required computation, or behaviour, in terms of various computing elements(executing certain types of steps, or sub-computations). There can be many such compositions, the better design will correspond to the degree of satisfaction of constraints specified. Hence, one might think of this mapping as many-to-many as well.

All the mappings can be partial: not all behaviours are *expressible* using formal specification notations. Similarly, not all behaviour/specification are *implementable*. An example is the existence of synthesizable subset in VHDL behavioural modelling.

By modifying Figure 6, we have the following figure, in which arrowheads denote the direction of the many-side of the mappings.

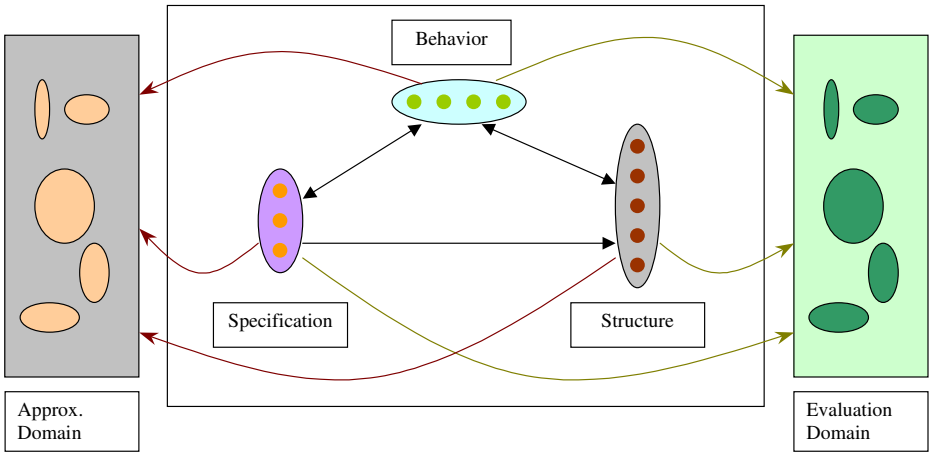

Figure 8: Complete Structure of Model of Computation

4.2 Motivation for Heterogeneous Systems' Design

The systems we discussed in last section are far too simple to be used in practical life. In general, the systems are *heterogeneous*, implying that there is co-existence of a large number of components of disparate type, function and properties.

As an example, a banking system involves transformational payroll programs, interactive access to database of clients, and reactive automatic teller machines and man-machine interfaces. Another example is of a compiler, that may be seen as a transformational from the outside, but internally it may be constructed in an interactive style out of concurrent processes which communicate via data streams; or a client-server application is interactive, but the calculation of the server's return upon the client's request may be a transformational step such as just returning the contents of a requested file.

A formal capturing of heterogeneity can be done using the concept of model of computation. Not all systems are simple enough to exhibit monolithic behaviour in one particular model of computation. For example, a simple numeric calculator not only requires a model in which a sets of "modes" and transitions can be handled, but also a model in which calculations, it's primary function, can be done. In such a case, perhaps a marriage of FSM model for mode-manipulation, and "arithmetic" model for representing numeric calculations will be sufficient.

This gives us an opportunity to think, that there can be choice for sub-systems or components as to which computational model is most suitable(e.g., type of FSM for representing the UI component). The next section is devoted on such details.

There are times, when the presence of human-friendly mechanisms[28] nudges a person to make the specification in a different model of computation. For example, 'C' derivatives: HardwareC/SpecC etc., or the C++ derivative: SystemC. In such a case, we anyway need to deal with multiple models of computation. We cover this angle as well.

4.3 Tackling Heterogeneity

4.3.1 Multi-modelling

We already hinted the usage of multiple(see section 4.6 for details) models of computation. We elaborate here using an example of a mobile handset. The RF transmitter/receptors can be designed using analog electronics as the computational model. The baseband processing of signals can be designed using the dataflow family models of computation such as synchronous dataflow. The protocol stack can be designed using models such as CFSM, or in a little more crude way, Harel's StateCharts. The middleware components such as event manager can be represented using semantics of entity-relationship model, and the applications can be driven by a normal FSM.

Another fundamental requirement of having multiple model is to accommodate components, which are *external IP*. By this term, we imply a component, which has been specified and developed by others, and purchased for the sake of making a bigger system.

Given such a thrust, we also need to look at how these models combine together. Furthermore, we need to look at what it means by combining models of computation.

4.3.2 Cost Distribution

We now refer back to section 4.1.2, and notice a few things about the cost or property mappings. One thing to notice that certain costs and properties(barring the ones in info-action triangle) are only measurable in a limited(or one) model of computation. For example, number of dead states can only make sense to be evaluated in FSA model, though the presence of significant number of such states can adversely affect the overall performance of a heterogeneous system, in which it is a component.

What is consistent, however, is the presence of the info-action triangle, which, according to our conjecture, drives all these cost mappings. The limit to which they can be reduced is dependent on the model of computation(e.g., in a plastic cell configuration, processing can be really reduced). Still, once the trade-off between these within a model of computation hits a wall, in a heterogeneous system, one can further try to redistribute these costs by appropriately choosing the models of computation. E.g., the whole cost of memory can be reduced by choosing an appropriate set of models of computation.

Design of heterogeneous systems based on info-action cost distribution, though obvious and intuitively appealing, according to our limited study, has not been explored. Hence concrete examples cannot be provided.

4.3.3 Refinements

Refinement literally means that crude oil still produces refined oil! Refinement(and abstraction) mappings are one aspect of combining multiple models of computation. Refinement implies defining vertical mapping between same aspect of two MoC: say a structure in MoC 1, mapped to structure in MoC 2. Refinements can be **concatenated**: this forms the design methodology. The terms initial MoC, intermediate MoC and final MoC have their meanings implied, thereby.

There is an associated notion of abstraction level. Refinement is defined as mapping from an (more) abstract MoC to a less abstract MoC. The level of abstraction can be measured by comparing implementation details[8]. Furthermore, the MoC in question theoretically should

also be related. The relation implies that the behaviour space should be preserved(in at least one direction), while refinement, though this relation may not be as straightforward to state. Finally, the abstraction level perhaps is defined only **within** a particular flow chosen; the same MoC may be less abstract, or more abstract, depending on which MoC it is compared with. Hence, the hierarchy of abstractions used in a flow might look independent of each other.

Why is refinement done? Or how is it identified, that an artefact in a particular MoC needs to be translated into another MoC as intermediate, before having a specific MoC as final? We point here few of the reasons:

I. One of the reasons is the availability of tools for verification/simulation at a particular level of abstraction. There are instances of such trials: e.g., usage of VHDL(RTL/behavioural) simulator for doing co-simulation in POLIS, while the specification language is in ESTEREL. Multiple specifications can be refined to such a level, where commercial tools are available.

On the negative side, such a refinement, e.g. refinement to VHDL, implies restricted symbolic debugging capabilities. This is because the correspondence between the refined VHDL code and the original specification can be fairly loose.

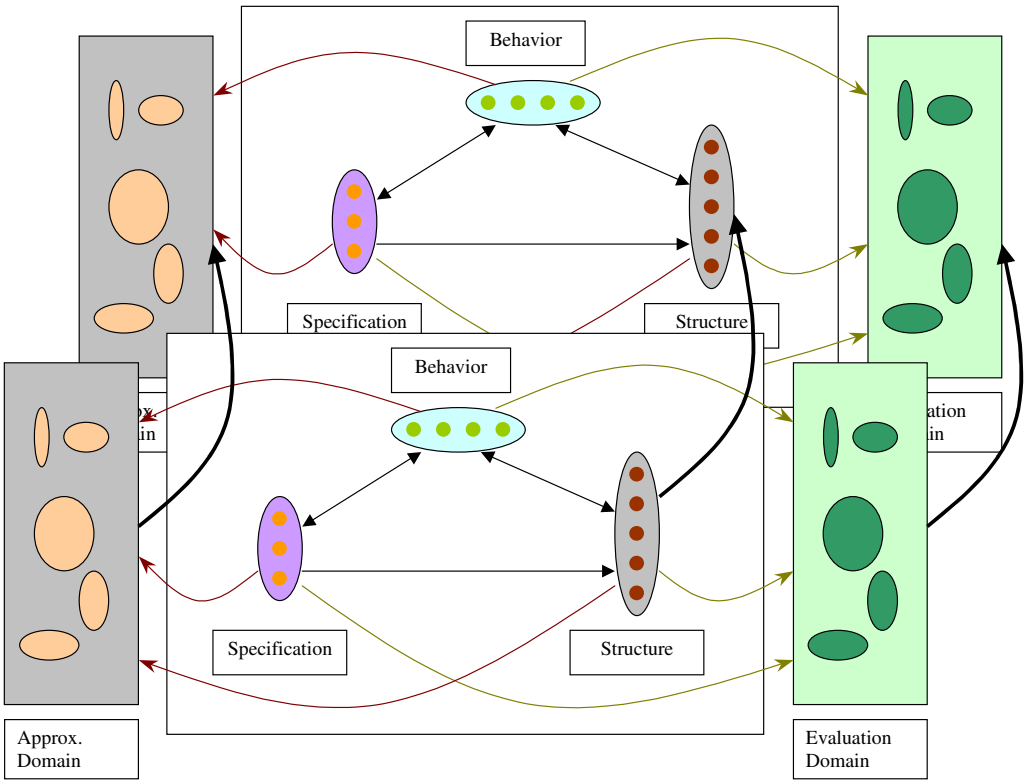

Figure 9: An Example Refinement

II. The other reason is exploiting the partial order of MoC, wherever possible, for better calculation of associated costs. Partial order can give rise to refinement of properties(both intensional and extensional), which can be measured at some MoC, and which can be close approximate of a final calculation.

One hidden problem in this case is that one MoC can be refined into more than one(can participate in multiple partial orders). In that case, one has to decide which one to choose. Parameters such as familiarity with a particular model of computation are used to help resolve the tie in practice.

III. By refining few models to a particular MoC, insertion of external IP can become easy. Bringing them to a single platform saves the overhead of thinking and using the interaction semantics. Non-sophisticated tools can be used at this particular MoC level.

IV. For formal verification of some property, abstraction mapping(opposite of refining) are used at times. This implies that certain properties are dropped from consideration. There are instances of CFSM being converted into synchronous FSMs[24]. When we explicitly focus on proving one property, the job can be eased by omitting the non-relevant details.

V. At times, an intermediate MoC is defined for portability reasons. That is, a model is translated into another model in another MoC, from which components can be translated independently into multiple MoCs. E.g., the CDFG created by 'C' compilers. The CDFG can then be synthesized into the instruction-set model of computation for various processors.

There are many types of refinements, all driven out of formal methods. Interested person can look into [12].

Figure 9 depicts a simple refinement process.

4.3.4 Category Theoretic Formulation of Problem

The best way to mathematically formulate the above two issues into a single problem is by using Category Theory[4], which is a general theory of structures. It deals with describing precisely many similar phenomena, or specifically, constructions with similar properties. One can thus transport, and study the set of all models of computation using Category Theory. By doing so, one essentially studies the semantics of various models of computations, and the relationship(such as embeddings) amongst them.

As an example, a category can be formed to deal with study of class of all vector spaces, and linear transformations amongst them.

Quoting [4], a category is denoted as a quadruple (Θ, H, id, o), where

- Θ denotes a class of similar structures, such as class of vector spaces
- H is a set of relation between two members of Θ: members of H(A,B), where A,B $\in$ Θ are known as morphisms.
- Id is an identity morphism for each object A: id_A: A $\rightarrow$ A.

[24] CFSM is more detailed MoC here.

- o is the composition law for morphisms. If f: A → B, and g: B → C, then gof: A → C. Composition is the fundamental primitive in Category theory, in the same sense that membership is the fundamental primitive in Set theory.

By looking carefully, it makes sense to treat each MoC as a Monoid. That is, the refinement relations(wherever applicable) between them, which form the morphisms, will be associative, and there will be an identity element. The identity element maps the set(in our case, a MoC) to the same MoC, thus forming a loop. One of the interpretations of such loop is local optimization, which may lead to a different structure within the same MoC. The morphisms can be interpreted as interaction between two MoC, i.e. how two structures in two different MoC interface with each other, and cope with the dynamics of such interfacing(paramount problem in composing heterogeneous systems). In real life, such morphisms are implemented using transducers.

Point work has existed so far, marrying two models of computation at time. Category Theoretic approach promises a "grand unified model of computation".

The issues with doing such unification are perhaps expected: notable among them are

- Out of the universal set of computational models, few subsets form a set(non-zero) of partially-ordered computational models. The partial order relationship is a kind of embedding.

- A particular computational model may be part of more than one subset. In such a case, work can still be done to form a total order out of these two partial orders.

Pioneering work has already been done on these lines: one can refer to [25] and [30].

4.3.5 Terms and Details

Now, we introduce few terms related to system design.

4.3.5.1 Design Task

A design task is a bare-bones definition of the system to be designed: it is like a customer telling: "Gimme a reference design of a digital camera having 256 KB flash, auto focus ability, …". The design task may further mandate re-use of certain pre-designed components. The design task may also have a collection of specifications(only a part of the overall system behaviour) of for re-use purposes.

4.3.5.2 Design Methodology

Typically known in the industry as the job of an architect, this involves breaking up the design task into design subtasks. The design subtasks imply modelling and designing components in the MoC decided for them, e.g. auto-focus algorithm on a DSP processor. To make a choice of componentization and their bindings requires understanding of various application domains. The design subtasks also utilize refinements, wherever required.

If we were to visualize a vertical tiling of various MoC, as depicted in Figure 9, then we can see that a design methodology forms a path in the graph. The jumps in the graph consist mainly of synthesis and refinement steps(Harel[27] was the first to note these). There is also at least one node with in-degree as 1, and multiple out-degrees. This is the node where the high-level

architecture is defined. This node can be the source itself, or can be a virtual node, where we simply start with predefined set/architecture of components.

A few figures, based on the reasons cited in section 4.3.2, are shown below.

Thus, formally speaking, a methodology is a set of models and transformations, possibly implemented by CAD tools, that refines the abstract, functional or behavioural specification into detailed implementation description ready for manufacturing. Typically, in the hardware verification domain, the machine derived from a high-level system description like RTL or behavioural spec is termed the specification, and the description corresponding to the low-level difficult system description, perhaps at the gate or the switch-level, is termed the implementation. Such incomplete definition gives rise to confusion, for it does not reflect the essence of behaviour-preserving refinements. For example, compiling a Lustre specification into gate-level netlist(using it's compiler), with FSA as intermediate MoC, resembles a vertical, diagonal refinement, which can lead to confusion.

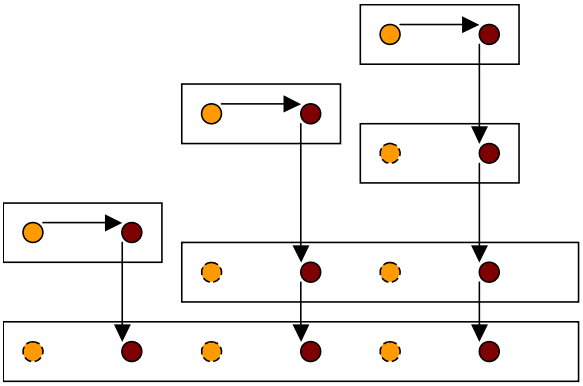

Figure 10: General Methodology

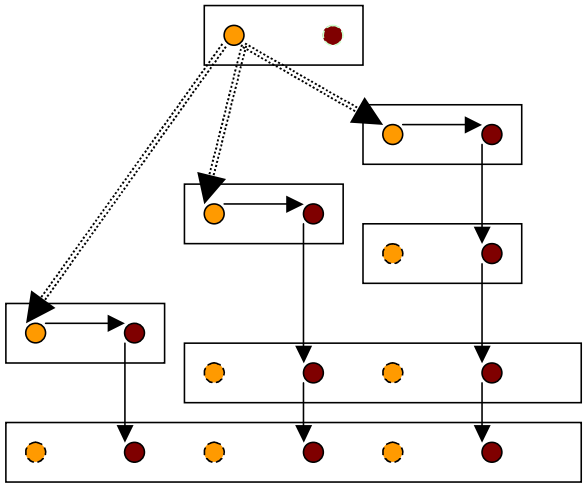

Figure 11: Methodology for reason V

Mostly the refinements are done structurally; but it has to explicitly or by simulation, guarantee that the behaviour is also consistent(equivalent), or it gets refined as well. The independent hierarchies(functional/behavioural and structural) are defined in [8].

The design flow need not start at all from a specification. The requirement can be changing a structure in one MoC to another. For example, someone gives a digital filter, and wants it to be converted from DSP code, to sequential circuitry, and then to hardware. This is driven by the fact that the customer foresees some kind of constraint satisfaction/cost advantage, for which the implementation is being changed. There can be point conversions also, driven by similar constraints. By adopting such definition, we counter the myth that the design methodology is a sequence of steps that transforms a set of specifications, described informally, into a detailed specification that can be used for manufacturing. Such a definition implies that intermediate steps are characterized by a transformation from a more abstract description, to a more detailed one[22], which is not always the case in industry(author's experience).

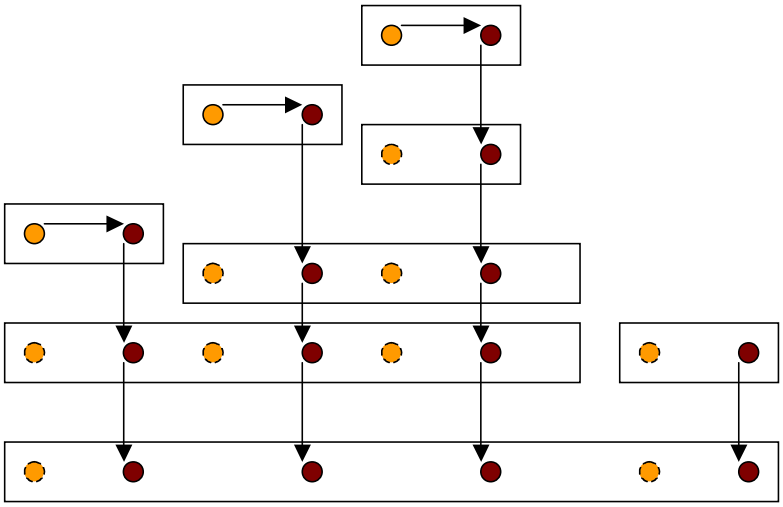

Figure 12: Methodology for reason I and III

4.3.5.3 Design Process

The design process comprises of the complete details("how") of all the sub-tasks under the design methodology *adopted* for the required design task. It includes defining configuration management, quality matrix, tools to be used at various stages and in various MoC, checkpoints during refinement and the method used, the choice of language for software components, to name a few. Needless to say, this is the point where abstraction stops, and reality sinks in!

4.4 Elements of Methodologies

Methodologies exist for hardware-only design[3], software-only design[12], mix-mode IC design(having both analog/digital hardware), hardware-software Codesign, and the emerging field of Reconfigurable systems' design. We look here at details of design methodologies, in general.

4.4.1 Models and Specifications

What is provided at times, or is rather available in case of heterogeneous systems is the collection of requirements, mostly stating input/output tuples for the system in some form, in the name of design task. The analysis step from this should lead to regrouping them into subsets of related requirements. If it is possible, then the design task is said to have a behaviour space. In the degenerate case, the set of independent requirements in practice being finite, they form a space by themselves: we do not cover such case in our definition. In such a case, one can write down specifications(as some type of logical formulae), in some specification language, which will relate to each of these subsets. It is from these subsets, the author's experience says, that one can derive the first hints of a possible architecture. A variation of this

problem is also known as the hardware/software partition problem, to be introduced in section 4.8; and this is currently an active area of research.

From these specifications, models are prepared, which then go into the design flow. For complex systems, it is inevitable that specification and implementation views of an evolving model have to be intertwined or interplayed, during the development activities, at different moments and also at distinct levels of abstraction. For instance, the whole system can be seen as a module and a state machine devised for it. We can later decompose the system in sub-systems and create, for each one, an activity diagram that represents the respective function. The sub-systems can by themselves, be decomposed in objects, which can have their life-cycles represented by a Petri net. We can go as many levels as we want, and as modellers, we are always changing specification/behaviour rather, into structural view, and vice-versa.

4.4.2 Design Flow

Design flow refers to the patterns shown in section 4.3.5.2. A design flow need not be a vertical line: though any strand of vertical line implies that models show an embedding. There can be many models of computation, which will play part of intermediate models of computation. A set of models of computation will be the final models: after reaching an implementation/structure here, the designer/architect will not bother about any further conversion to any such structure. At these MoC, implementation details are very clear, and each component also has a clearly defined function/behaviour.

The design flow can branch out at any point(see Figure 11) into multiple MoC. This introduced additional interfacing requirements, which can be broadly grouped into *synchronization*(e.g., software scheduling) and *communication*(e.g., transducer design) requirements.

Integrating reusable blocks of IP into a particular model requires compatible abstraction levels between the reused IP and the rest of the model, or at least bridges between levels. For example, most of today's IP blocks are available either at the register transfer or transaction level for hardware, and either source or object code for software. Bring to same MoC can help in joint verification(an important requirement) of the entire model.

Existing design tools are basically of two sorts:

- Mainly focusing on functional design, the first come from the specifications domain to reach the detail design of the components(top-down process).

- Mainly focusing on platform design. This introduces a re-use based design process, and hence tends to be bottom-up.

The current problem in industry also lies in the fact that different people follow different design methodologies tailored by the experience of their development communities. Still, these methodologies can be broadly grouped under three categories.

4.4.2.1 Top-down Approach

In top-down approach, a comprehensive model is captured using a single specification language, such as SpecC/SystemC. At the topmost level, both the behaviour and the structure may be highly unspecified(lacking details). The behaviour of such a model is then decomposed. Decomposition includes the delegation of part of behaviour(function) to an architectural

component. As a side observation, the structure of this graph resembles the structure of a DFA. A detailed picture of this approach is given in [27].

The advantage of such an approach is that one can postpone the decision of *binding*: an algorithm can be implemented on a DSP, or on a general-purpose microprocessor. Such a binding is done after approximating and analyzing the various cost factors of such configurations.

Such a unified modelling language is developed has the potential of being far too expressive and generalized. In that case, verification of models and ease of synthesis/refinement becomes very tough.

4.4.2.2 Bottom-up Approach

A behaviour can also be composed, from elementary components, and found to be equivalent to what is desired. This way, one can choose a required composition of behaviour from an available set of elementary behaviours(computational elements). Bottom-up flow requires organization of specification, such that components can be identified(equivalent notion is identification of strongly connected, directed sub-graphs)[35].

Platform-based approach of system design falls under this category. The term platform implies a set of reusable, general-system(components). Following [31], it is important to distinguish between special-purpose systems, from general-purpose systems. This is because for these, the design methods and constraints differ widely. While special-purpose systems are designed with specific purpose in mind, and hence have stricter design constraints, general-purpose systems are developed with no single application in mind, and hence focus shifts on designing a system framework, which can best match multiple targets.

The above is a simplified view: applications, or rest of the system components(which need not share functions, and hence do not have re-usable behaviour elements) need to be designed, which tailor the use of the services offered by the platform. Thus, the system design can be divided in two distinct processes: the bottom-up design of the platform, and top-down design of the application. The first process aims at offering a development platform to application developers, while the second aims at deploying an application on a platform.

The bottom-up approach has not been worked upon, much. Also, software development of systems, from the author's experience, is moving towards component-based design techniques across the industry. A recent work on such cross-fertilization of ideas can be found in [35]. Much more work is still to be seen in this direction.

4.4.2.3 Mix Approach

Many researchers are doubting whether top-down approach will work, because it is very difficult to combine multiple MoC in single framework([25] has combined a limited set of MoC). Hence there are other approaches that explicitly use more than one MoC. Here the choice of MoC is made before the modelling phase starts. Each component is then modelled in an MoC specially targeted towards that type of computation. The cost of altering the choice later is too high, and hence in this approach, the choice is typically not altered. The integration offered by this approach is less deep, but simulation is used to validate the design and integration of the various MoC(an example being co-simulation).

This approach should not be confused with bottom-up approach, where *pre-implemented* components are re-used, and made part of a bigger specification(e.g., an application framework). In mix approach, the participant specifications are almost functionally complete in themselves(e.g., specification of a equalization unit in DSP).

The practical patronage to mixed approach arises from the fact that system development features, mostly business-requirement driven, various preconceived, unavoidable limitations on the structure, distribution, capabilities and interconnections of their components. Hence natural decomposition, though a better choice, may not be possible; some components of overall behaviour, or structure may be forced for re-use. Availability of design tools also at times restricts a component to be sticky to a particular MoC.

There are shades of mixing in certain top-down methodologies[17]. Such approaches are dominantly top-down, which has elements of bottom-up(for IP re-use).

4.4.3 Architecture Exploration

Architectural design(or exploration) is the assignment of a type system(intension) of components(e.g., binding/allocation in hardware design) to a set of specifications. The dotted arrows in Figure 11 depict an example assignment. This is actually a macro-architectural exploration(area having architectural description languages). Micro-architectural exploration is of camouflaged nature(typically performed by tools), and is closely linked to refinement mapping. An implementation, or structure within a computational model, is termed as architecture, if it is refined into another structure in another MoC that demonstrates more implementation details.

The term architecture exploration has outgrown its initial usage, and is now used in a related problem known as partitioning, in heterogeneous systems. Partitioning is an inverse mapping: portions of specifications are assigned structural components, the collection of which is deemed architecture. Thus, it implies assigning multiple MoC, and may be few architectural templates to portions of specifications.

A partition is optimal if it is found leading to optimization of certain cost functions. Also, it needs to satisfy certain properties, which show eternality, rather than local or global optimality.

Once a first "blueprint architecture" is obtained from prospective architecture exploration, confirmative exploration can help fine-tune characteristics of various components. Usage of cycle-accurate models is an example of this. In the worst case, a complete validation may point unsatisfactory properties, and architecture may need to be redone.

4.4.3.1 Relation to Design Flow

Design flows can never be just top-down, or bottom-up. Loop backs, or iterations are required for various purposes. As an example, already discussed, there can be insertion of IP in a predominantly top-down design, which partially constrains the choice of components. Wherever there are iterations, they are due to something being tried for optimization. These iterations are known as explorations, and can lead to changes in both macro-architecture as well as micro-architecture. Choosing components, and performance analysis to provide feedback, are essential elements of this task.

The position of MoC in the design flow graph, in which the model is changed for next iteration, is sometimes termed as (micro-architectural)exploration level. It is imperative, that different

exploration levels propose a trade-off between the architecture exploration cycle period (defined by the simulation speed and time to make changes in the architectural model) and the level of details obtained.

The macro-architecture is intensional, hence many architectures can satisfy the cost(behaviour) constraints, in the same sense of intension and extension. Such a case leads to the notion of pareto-optimal designs. It is also non-local[39], and hence it can be explored only when a cohesive specification is analyzed for MoC-specific components and their interconnection. That is, architectural exploration should to be tried to be done at the higher levels in the design flow graphs; after components and respective MoCs are fixed, one only needs to bother about the design process in each MoC. If one has to really wait for one set of implementation to be ready, then one may critically fall short of time to investigate and explore all the possible configuration of components.

Design process within a MoC can be either synthesis or compilation, depending on the MoC. The synthesis should not be confused with refinement mapping across MoC; and similarly, analysis should not be confused with abstraction mapping across MoC.

Architecture exploration and choice of design flow are subtly intertwined. The degree of constraints imposed on architecture hints at the possible choice. Current synthesis-based methods almost invariably impose some restrictions on the target architecture in order to make the mapping problem manageable. It may be due to vendor restrictions on components, interfacing constraints.

4.4.3.2 Exploration in Top-down Flow

This is the ideal case, which actually hardly happens(author's experience). In this case, there are no preconceived components, and hence the architect is free to choose his choice of components, analyze and perhaps complete redefine his original choice.

Uniform modelling languages such as Spec-C and System-C make perfect choice for the specification purposes. Such approaches have suffered a bit from the angle of tool support required for high-level cost approximation, and also refinement/synthesis.

4.4.3.3 Exploration in Bottom-up Flow

We take the example of platform-based design here. In bottom-up approach, platform development and application development, are two distinct sub-processes. We assume that the system hardware is already defined: it's either a ready-to-use commercial platform or a reused (in-house or external) design. If that is not the case, then the flow will be a top-down flow(a slight variation in case system hardware is to be derived from an existing design).

These two processes necessarily meet at some point, when the platform is ready to host an application, and the application ready to be hosted on a platform. Performance analysis and architectural exploration takes place at such point. The system architecting activity consists in optimally allocating system parts to the existing programmable and configurable hardware components. Experience, design by analogy and very often non-technical (economical, commercial, etc.) imperatives guide designers in their initial architectural decisions. Also, exploration can be done by trial and error and by executing "what-if" scenarios.

Architectural exploration is possible if the same functions can be simulated on different hardware platforms, which implies defining a separate model(similar to that offered by an ADL)

from the functional model for representing the system hardware(property-based models). The choice of abstraction level for having such models affects the reliability of the exploration result.

4.4.3.4 Exploration in Mix Flow

In this approach, the specifications written in individual MoC are taken. A co-simulation can be done over this collection, an example being the usage of PTOLEMY[17]. Few specifications have the prefixed choice of being mapped to implementation within the same MoC, without further evaluation. This is like saying that GUI is implemented in software, come whatever may. Other specifications are grouped together, and re-partitioned(in precise words, re-targeted) in different set of MoCs. E.g., spec A and spec B are made into spec C(in which A and B are invisible), and repartitioned into specs D, E and F. Further simulations are done, and architecture exploration moves.

4.4.4 Analysis

The refinement process also involves mapping constraints, performance indices and properties to the lower level so that they can be computed for the next level down. Not just that, the synthesis within a model of control(from e.g. a spec to a structure) is mandated to preserve the properties.(1.3.4, [8]).

As pointed out, approximation and evaluation steps are part of analysis of a model. For **any** design flow, if the analysis is done as early on, as possible, good amount of re-work due to iterations can be saved. But, in design flow, early-on models(if there exist such models) tend to be abstract structurally and behaviourally, and hence the analysis is more of approximations. An advantage is that approximations are quick to be evaluated, and hence more exploration can be packed on a level, which is higher in abstraction.

4.4.5 Synthesis

We restrict our definition of synthesis to be specification to behaviour mapping within a model of computation.

Algorithm design falls as a part of synthesis task. Since it is often application-dependent, it can only be done in suitable model(s) of computation(1.3.2, [8]).

Not all constructs of a specification language are synthesizable. Hence, a structure(model) can be synthesis model(outcome of synthesis) or simulation model(). Typically, synthesizable models have constructs which form subset for simulatable models(read: synthesizable subset). Only the implementation, or the design process, is with respect to a composition of objects: which means functions and computation.

4.4.6 Refinement

We have already talked about refinement in section 4.3.2. The refinement can be either *structural* or *behavioural*; we talk about them next.

For reactive systems, refinement becomes even more tricky; how will concurrence be refined, so that the refinement process does not lose its meaning, is still subject to active research. For transformational systems, the refinement mostly becomes functional, or structural,

decomposition. A composition of complex functions, using primitive recursive functions, is an inverse example of this.

4.4.6.1 Behavioural Refinement

Behaviour refinement is a process of fine-tuning the system's performance by supplying/providing more information about its behaviour. Behaviour implies the dynamics, and hence it is either the temporal, or the data-type[25] properties, which get refined. What is preserved here is the set of so-called external signals[27]. In fact, that is the only thing, which seems to be preserved during structural refinement as well.

An example of behaviour refinement is the program transformations used in programming languages domain. Though physically the rules/transformations are changes in structure, loop unrolling introduces refinement of the behaviour(additional behaviour, that the program does not loop, but still does the same computation). It is known, that these transformations are valid refinements, and hence no special verification is required during such steps of the refinement process.

Another example of behavioural refinement can be taken from the observation that the execution time of a CFSM transition(when next a CFSM will transit) is unknown a priori(the semantics of the MoC has globally asynchronous property). But, with refinement involved in synthesis from initial specification, more precise timing information gets added. This happens as more and more design choices are made(e.g., partitioning, processor selection, compilation). Yet another example is the Lustre compiler, which designs(a refinement) FSA to satisfy a Lustre specification(intermediate step), and outputs a FSA structure in the form of an OC-format file.

Such kinds of refinements do not seem to be used often in practice. What is seen is that synthesis is done first, followed by structural refinements. Ideally, the synthesis and (behaviour) refinement steps can be mixed[27], but the authors could not find a concrete example of this.

Across two models of computation of disparate kind, it may not be possible to define any behavioural refinement. In such case, only simulations and bi-simulations can perhaps prove the equivalence.

4.4.6.2 Structural Refinement

Structural refinement is done to bring the system closer to its final form, by supplying more information about its implementation. Also, the whole objective of design process is to do behavioural refinement(to successively have more optimal behaviour). To do so, structural refinement is used.

Structural refinement hence also needs to preserve, or refine interface behaviour. The interface consists of input/output events/signals/channels etc. The processes, or components can themselves be defined in terms of these inputs/outputs[25], and hence any such preservation[27] is imperative. Such refinement also gives some semantics to development of interfacing elements between MoC[26](see section 4.6).

[25] Richer set of data types/operations represent more expressive model for computation.
[26] E.g., a scheduler can be looked as a synchronization process, while a transducer, or a relay network can be looked as communication process.

The so-called "laws", or requirements of structural refinement can be defined easily for models of computation, which are semantically close enough(such as gate and transistor level). [16] provides a set of requirements in terms of constraints over the abstraction and refinement of models. The models themselves are expected to have *related* entity-relationship semantics of their own(still, two different models of computation). If such laws cannot be discovered, then the refinement process cannot be automated, and post-design validation of refinement step might be the only thing possible.

In the software architecture domain, an excellent extension of structural refinement has been noted in [12]. It also provides an example analysis of preservation of behaviour, after the structural refinement(the behavioural equivalence proof). It does so, by using model-checking techniques to analyze the properties of refinement.

Life may not be just behaviour-preserving sequence of structural refinements. Hardware design flow tends to be like that, for it was defined for a functional (sub-)system. At times, the flow may be warped, and hence it will need explicit verification of whether one particular decomposition preserves or refines the behaviour also.

As with the behavioural refinement, structural refinement may not always possible. Say, an analog structure cannot be refined into a digital structure. But given two such structures, their output waveforms can be *observed* to be consistent. Then, we can claim that since the behaviour is consistent, the models are structurally equivalent, or they exhibit abstraction/refinement properties.

4.5 Verification of Design

Formal techniques can be used to do verification of all the steps involved in a design methodology. The following figure shows the relation of verification techniques to a single model of computation.

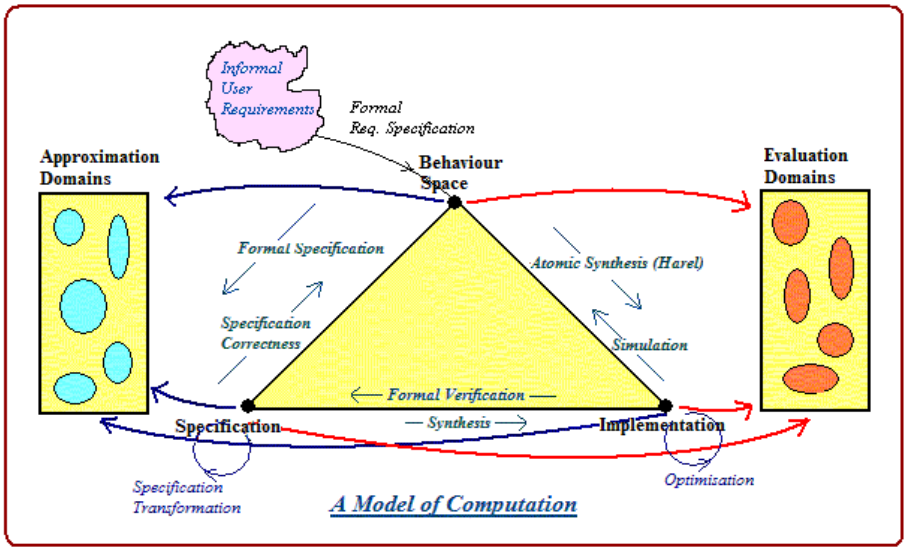

Figure 13: Directed Graph for a Model of Computation

One can exploit the properties of the model of computation to reduce the verification cost. For heterogeneous systems, design for verifiability is pretty important concern; otherwise, the power of using modelling does not help reduce the production cost of the system.

In a design (methodology) flow graph, verification needs to be done at various steps. At times, it helps in deducing properties of refinement. In other cases, it provides with better, or closer to the actual, approximation of various costs, and in certain cases, evaluation. There are a lot of such techniques, which can be fit in our methodology diagram[29].

At the end(when the implementation finishes), a complete **validation** is required to be done. The verification only enhances the probability that the right choice(s) has been made; in the worst case, using verification tool will end up being more duration-costly affair. The results of complete validation might not be directly comparable to result of approximate verification done. This is because cycle-accurate models, and details processor models, etc. are used to do the detailed validation. Hence, to choose what costs/properties to verify at various stages of refinement, has to be carefully decided(so that the disparity is not too much). Such refined costs can be used to perhaps do few more exploration cycles.

Both property- and cost-analysis are *part* of verification process. The data for analysis is obtained by analysis/abstraction mapping.

4.6 Computation Models and Interconnections

At least two partial-order subsets of models of computation exist: the control-dominated models(e.g., CFSM, FSA, StateCharts), and data-dominated(e.g., DFG, Synchronous DFG).

Also, there exist a lot of models of computation(why? See [22]). For our design purposes, we need to figure out the semantics of interconnections for such models(wherever it exists). Physically, interconnect is realized using separate interface component. The behaviour aspect of interface block can be a protocol, and structure can be port /queue model, etc.

We *just* mention a few(limitation of the study of the authors) models of computation relevant to system design. There are many papers, which describe and review each one of these, and hence we do not make any similar attempt. Certain application domains for these are quoted in a table in [19].

A good future exercise will be to elaborate the structural, behavioural and specification view of each of these MoC. For example, a and b are elaborated here. A complete picture will be to figure out design flows possible, and the missing design flows across the planes depicted by MoC. It shall also include a good survey of what are the cost factors involved in each plane, i.e. the evaluation and approximation domains, the relation of these to the views in a plane, and the relationship with their siblings across the planes. May be for a and b, such relationship should be elaborated here.

1. Von Neumann(sometimes compared to RAM model)
2. SDL process network
3. Neural Network
4. Program State Machine

5. Petri Net
6. Kahn Process Networks
7. (Normal) FSM/FSA
8. Behavioural FSM
9. Co-design FSM(CFSM)
10. Timed Automata
11. Hierarchical concurrent FSM(StateCharts)
12. FSM with datapath
13. Synchronous Dataflow
14. Dataflow
15. Synchronous/Reactive
16. Actors
17. Process Algebra
18. Concurrent Sequential Processes
19. Calculus for Communicating Systems
20. Programming language such as 'C'. They have a universal model of computation.
21. Differential Equations
22. RTL, or sequential
23. Logic-level, or combinatorial(almost same as gate-level)
24. Transistor-level

4.7 Trends in Evolution of Methodologies

Though industry still works its way through methodologies evolved by its own experience(at least, the author has been part of two such cases), any particular design flow can never come to a stage of monopoly. Still, there are trends in evolution of these, which are driven both by the industry and the academia. We note (just) a couple of them. The note has been made by looking at academic and industrial documentation. From academia's part, at least five developments deserve mention here: PTOLEMY[6], POLIS[7], SPEC-C[24], SYSTEM-C[5], and MATISSE[33].

Usage of formal models is being pushed[24], and hence, implicitly the top-down design flow. It has got distinct advantages. Such a unified modelling language is developed has the potential

of being far too expressive and generalized. In that case, verification of models and ease of synthesis/refinement becomes very tough. Metropolis[34] claims to be trying to overcome this.

POLIS extends the definition of algorithm development from system design perspective. Algorithm development for POLIS in not just functional elaboration, it is temporal elaboration as well. Which implies, that in a loop, macro-architectural exploration is done, and tested, and hence the algorithm for implementation purposes develops itself. POLIS also shows integration of MoC-specific formal verification technique. Since reliability is a major aspect in avionics/control dominated systems, POLIS integrates a technique for such verification, and also tries to make rest of the flow independent of the MoC.

Easy insertion of pre-developed components(known as IP blocks) is also being targeted. Spec-C and POLIS have both covered this aspect.

MATISSE and METROPOLIS try to attack, what according to us is driving the disparity in industrial practices of methodology: robust support for property and constraint analysis. A good approximation can reduce the iterations during explorations, and a good support for doing so can help bring smiles on faces of few marketing executives!

4.8 Hardware-Software Co-design

Hardware-software Codesign stands for **concurrent** realization of components in hardware as well as software.

4.8.1 Evolution

The major thrust on such design has come from the advent of *embedded systems*. Embedded systems are informally defined as a collection of programmable parts surrounded by ASICs and other standard components that interact continuously with an environment through sensors and actuators. The programmable parts include micro-controller and digital signal processors, to name a few[27].

The original methods of system design were to use hard-wired technology, which could be as compact as an ASIC, or a group of individual components forming a board-level solution. The solution is known as a hardware solution.

With the advent of microprocessors, which separated the algorithm specification from the hardware required to execute it, a new domain of software systems has cropped up. This second method is based on software-programmed microprocessors. Processors execute a set of instructions to perform a computation. By changing the software instructions(to execute a different algorithm), the functionality of the system can be altered without changing the hardware.

What works have contributed to this field, in some kind of chronological order, from various disciplines of research, is excellently presented in [18].

[27] Future will see FPGA-based Reconfigurable components as well

4.8.2 Business Aspect

Software had to be treated as another primary component in chip design; this mainly happened due to Moore's law. Generally, software is used for features and flexibility[28], while hardware is used for performance. Also, certain computations can only be expressed in one way, e.g. user interface part of the system.

In the end, it should result in shorter time-to-market and reduction in the design effort and costs of the designed products(e.g., non-reusable engineering costs). In fact, [17] claims that 70% of the cost of system designing is going to be attributable to software in near future.

4.8.3 Definition

Hardware/software is an adjective applied to systems whose eventual implementations will contain a hardware portion and a software portion; the software portion, of course, must contain the hardware(e.g., microprocessor or micro-controller), on which to run the software.

Hardware and software are two different *implementation domains*. They exhibit different types of execution constraints for software and hardware implementations: concurrency for hardware, mutual exclusion for software. This is because hardware related models are often combined into saying that they are synchronous models of computation, doing parallel computation; this is in the sense that they are governed by a global clock. While, software models are often combined into saying that they are asynchronous models of computation, doing sequential computation.

Codesign is an all-encompassing term that describes the process of creating a mixed(in this case hardware/software) system.

The concurrent development of systems now-a-days demands development of ASICs, standard hardware components, selection of programmable components, and development of application software, which will run on them[17].

The difficulties that feature in Codesign are the result of the two contradictory driving forces described above: increase in complexity and decrease in time and costs. Complete hardware and software multiprocessor systems are integrated into a single silicon circuit called System-on-Chip (SoC), thus increasing complexity. Also, systems integrate an ever-increasing number of external ready-to-use components (or IP blocks), to save time. Thus, most of the design issues with heterogeneous system design apply here as well. As another example, architecture exploration gets refined into partitioning.

The major concerns in co-design are[18]

1. To provide analysis methods to check for the constraints/costs such as performance, power and size, etc. Such methods increase the predictability of the design process. There are methods to do such analysis, the most common one is known as co-simulation. As an example, the POLIS framework uses PTOLEMY as the co-simulation engine to do the cost analyses.

2. Synthesis methods, so far practically restricted to (rapid) prototyping, to let researches and architects/designers to evaluate many potential design methodologies.

[28] Flexibility of software allows late design changes and simplified debugging opportunities.

3. To search modules within the specification for the hardware/software partition(see section 4.8.5).

4.8.4 Modelling

Current methods prevalent in industry, for designing embedded systems, require specifying and designing hardware and software separately. A specification, often incomplete and written in non-formal languages, is developed and sent to the hardware and software engineers. Hardware-software partition is decided *a priori* and is adhered to as much as is possible, because any changes in this partition may necessitate extensive redesign. Designers often strive to make everything fit in software, and off-load only some parts of the design to hardware to meet timing constraints. The problems with these design methods are:

- Lack of a unified hardware-software representation sometimes leads to difficulties in verifying the entire system, and hence to incompatibilities across the HW/SW boundary.

- *A priori* definition of partitions, which leads to sub-optimal designs.

- Lack of a well-defined design flow, which makes specification revision difficult, and directly impacts time-to-market. The "concurrency of development" being the key idea here, the evaluation and optimization of the entire system design gets a bit postponed due to time taken for models to emerge.

It is here that modelling can be used to advantage. Usage of models dominantly comes from hardware design. In software-only systems, the design process is interchangeably used for the implementation(sub-)process, which is a loose usage. This is because usage of modelling and analysis before detailed design is a paradigm coming up now(the OO way, or the usage of UML now). There is also an interesting proposal to make concurrency of development the secondary requirement[17], thus allowing the trade-off and exploration to becoming primary requirement, and hence tended early on.

4.8.5 Partitioning

Lying below the **formal** co-design issue is the problem of efficient allocation of functions, which constitute of system behaviour[29]. It is a specialization of choosing an architecture fitting into the specified constraints(see section 4.1.1). Formally and generically, it is the method of expressing a computation amongst various models of computation.

The suitable architecture evolved here consists of a hardware platform, executing some system and application software (modules). The software part is a set of application-specific software routines, running on a dedicated processor or ASIP, while the hardware part consists usually of one or more ASICs.

Before choosing whether a part of a system should be designed in software or hardware, designers have to represent what the system does (its functions) with no considerations whatsoever on its nature (software or hardware). Functions of a system can be designed and verified independently from any technological considerations by creating a functional model. Functional design helps designers concentrate on the application, with no limitations induced by physical considerations. This leads to identification of modules.

[29] In stricter sense, system behaviour is expressed as a set of relations.

Co-design is not just the problem of hardware-software partition; or choice of macro-architecture. At times, one has to tinker forcibly with a natural partition, due to things as vague as re-use of IP(specifications). Also, if multiple specifications are used, then one also needs to validate, whether the implementations of components, put together, *function* so as to satisfy the specifications(equivalent stage in hardware design flow is sign-off).

4.8.6 Performance Measurement

Performance estimation for software components is very tough, because not only we must consider the structure of the program(control flow, loop etc.), but also the performance of software environment(OS overhead, compiler optimizations, input data patterns/properties) and of the hardware environment(CPU type, caching, pipelining, pre-fetching). Hence at system level, mostly a few performance constraints can be optimized for(such as design/exploration of memory architecture), before the true hardware/software architecture level design issues such as hardware synthesis, software compilation and inter-processor communication synthesis.

4.8.7 Virtual Prototyping

Platform-based Codesign can do a different kind of modelling to alleviate the delay in verification: usage of virtual prototypes of the platform. Virtual prototyping comes from floor-planning requirements(back propagation in top-down design flow) in ASIC as well as FPGA. But it can exist at any level of abstraction[8].

In platform based design methodology, concept of developing a virtual prototype of the platform helps in Codesign. The resulting platforms are capable of executing application software on top of the virtual hardware at speeds of millions of cycles per second. Hence, they enable early software development, in advance of a first silicon prototype, and concurrent with hardware development.

4.9 Issues in Reconfigurable Systems' Design

An introduction to Reconfigurable systems can be found in [38]. Reconfigurable systems have been conceived to overcome the drawbacks, which happen due to having hardware, or software components in the design of the system.

Point to note is that reconfigurability is there in every model of computation. A dynamic change(denoted by path) in the behaviour in a particular model of computation can induce a dynamic change in structure, much in the same way as a static change(at design time) can induce.

As an example, a reduction in battery power level can shut down some of the system components(power-aware scheduling problem), which is equivalent to adjustment in the system structure. In this case, the interconnect was tinkered in order to induce a change in structure. Another adjustment, which can be done is to change the weight and number of coefficients of a high-performance channel equalizer in the base-band segment of the mobile: due to the parametric nature of the components/structure, the function gets tuned to the changed dynamics of behaviour(no need to bother about interconnect).

The worst kind of structural changes can perhaps happen, when one has to change the macro-architecture: for an example, the hardware/software partition, or the binding to a set of models of computation. This might be a very complex activity, and may not be practically realizable/useful at all.

5 References

[1] John Hayes. *Computer Architecture and Organization*. McGraw-Hill Publications.

[2] Michael A. Harrison. *Introduction to Switching and Automata Theory*. McGraw-Hill Publications.

[3] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Publications.

[4] Jiri Adamek, Horst Herrlich and George E Strecker. *Abstract and Concrete Categories*. John Wiley and Sons, Inc.

[5] Open SystemC Initiative. *Functional Specification SystemC Version 2.0.1*. http://www.systemc.org/

[6] The Ptolemy Project Homepage. http://ptolemy.berkeley.edu/

[7] The POLIS Project Homepage. http://www-cad.eecs.berkeley.edu/Respep/Research/hsc/abstract.html

[8] RASSP Taxonomy Working Group. *VHDL Modelling: Terminology and Taxonomy*. http://www.atl.external.lmco.com/rassp/taxon/rassp_taxon.html

[9] CoFluent Design, Inc. CoFluent Studio SDE Specification. www.cofluentdesign.com.

[10] TCS Internal Technology Plan. "*Advanced Multimedia Initiative*". November 2002.

[11] Florentin Eugen Ipate. "*Theory of X-machines with Applications in Specification and Testing*". Ph.D. Thesis, University of Sheffield, UK, 1995.

[12] Andreas Kerschbaumer. "*Behavioural Refinement of Software Architectures*". Ph.D. Thesis, Institute for Information processing and Computer supported new Media, Graz University of Technology.

[13] G. Bosman. "*A Survey of Co-design Ideas and Methodologies*". Master's Thesis, Vrije Universiteit, August 2003.

[14] Gerard Berry. "*The Esterel v5 Language Primer*". INRIA, April 1999.

[15] Antonio Brogi, Alessendra De Pierro. "*Linear Embedding for a Quantitative Comparison of Language Expressiveness*". In Elsevier Science B.V., 2002.

[16] Robert M. Colomb, C.N.G. Dampney, Michael Johnson. "*Category-Theoretic Fibration as an Abstraction Mechanism in Information Systems*". In Acta Informatica, Volume 38, 2001.

[17] Luciano Lavagno, Alberto Sangiovanni-Vincentelli and Ellen Sentovich. "*Models of Computation for Embedded System Design*". In 1998 NATO ASI Proceedings on System Synthesis.

[18] Wayne Wolf. "*A Decade of Hardware/Software Codesign*". In IEEE Computer Society, 2003.

Formatted: Italian (Italy)

Formatted: Italian (Italy)

[19] Luis Alejandro Cortés, Petru Eles and Zebo Peng. "*A Survey on Hardware/Software Codesign Representation Models*". In Report for SAVE Project.

[20] Walid A. Najjar, Edward A. Lee, Guang R. Gao. "*Advances in the Dataflow Computational Model*". In Parallel Computing, 1999.

[21] Rajesh K. Gupta and Giovanni De Micheli. "*Hardware-software Co-synthesis for Digital Systems*". In IEEE Design and Test of Computers, 1993.

[22] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli. "*Design of Embedded Systems – Formal Models, Validation and Synthesis*". In Proceedings of the IEEE, March 1997.

[23] Rabi N Mahapatra. "*Codesign Framework*". Course Presentation, Texas A&M University.

[24] Daniel D. Gajski, Jianwen Zhu and Rainer Domer. "*Essential Issues in Codesign*". In UC Irvine, Technical Report, June 1997.

[25] Edward A. Lee and Alberto Sangiovanni-Vincentelli. "*A Denotational Framework for Comparing Models of Computation*". In IEEE Transactions on CAD, December 1998.

[26] Dag Bjorklund and Johan Lilius. "*A Language for Multiple Models of Computation*". In International Conference on Hardware Software Codesign, 2002.

[27] D. Harel and A. Pnueli. "*On the Development of Reactive Systems*". In Logics and Models of Concurrent Systems, Springer-Verlag, 1985.

[28] P.S. Subramanian. "*The Amorphous and the Crystalline*". TIFR Internal Report, 1996.

[29] P.S. Subramanian. "*Formal Verification Techniques and System Design*". TCS Internal Report, 2004.

[30] Samson Abramsky, Simon Gay and Rajgopal Nagarajan. "*Interaction Categories and the Foundations of Typed Concurrent Programming*". In Proceedings of the 1994 Marktoberdorf Summer School. NATO ASI Series F, Springer-Verlag, 1995.

[31] Luca Benini and Giovanni De Micheli. "*System Level Power Optimization: Techniques and Tools*". In ACM Transactions on Design Automation of Electronic Systems, April 2000.

[32] Daniel D. Gajski, Jianwen Zhu and Rainer Domer. "*IP-centric Methodology and Design with SpecC language*". In Proceedings of the NATO ASI on System Level Synthesis for Electronic Design, Il Ciocco, Lucca, Italy, Aug. 1998.

[33] D. Verkest, J. L. da Silva Jr., C. Ykman, K. Croes, M. Miranda, S. Wuytack, G. de Jong, F. Catthoor, H. De Man, *"Matisse: A system-on-chip design methodology emphasizing dynamic memory management",* In Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology, Kluwer Academic Publishers, July 1999.

[34] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Alberto Sangiovanni-Vincentelli. "*Metropolis: An Integrated Electronic System Design Environment*". In IEEE Computer, April 2003.

[35] J. Morris Cheng and S. Kagan Agun. "*Reusable Component Design in VHDL*". In IEE Computing & Control Engineering Journal, 2002.

Formatted: Italian (Italy)

Formatted: Italian (Italy)

Formatted: Italian (Italy)

Formatted: Italian (Italy)

Formatted: Swedish (Sweden)

Formatted: Italian (Italy)

Formatted: Italian (Italy)

[36] Mohammed Elkoutbi and Rudolf K. Keller. "*Modeling Interactive Systems with Hierarchical Colored Petri Nets*". In Proceedings of the 1998 Advanced Simulation Technologies Conference, 1998.

[37] Patrick Cousot and Radhia Cousot. "*Refining Model Checking by Abstract Interpretation*". In Automated Software Engineering Journal, 1999.

[38] Katherine Compton and Scott Hauck. "*Reconfigurable Computing: A Survey of Systems and Software*". In ACM Computing Surveys, June 2002.

[39] Ammon H. Eden and Rick Kazman. "*Architecture, Design, Implementation*". In International Conference on Software Engineering, 2003.

Formatted: Swedish (Sweden)