A Methodology for Example-based Specification and Design

C. Alippi, S. Ferrari and V. Piuri
Department of Electronics and Information
Politecnico di Milano
Piazza L. da Vinci 32, 20133 Milano, Italy
{alippi,sferrari,piuri}@elet.polimi.it

Abstract

In this decade we are experiencing an ever-increasing use of embedded systems; fast prototyping, time to market and severe implementation constraints must be faced to provide an effective — low cost — solution for a given application.

To this end, several algorithmic formalisms are available to describe and validate complex systems at a behavioural level in order to minimise development costs and facilitate the integration of design and implementation constraints.

Unfortunately, a soft-computing paradigm cannot be directly manipulated by conventional development environments for embedded systems unless an algorithmic description is available. In general, such a description is the result of a training procedure which, by following the selection of the most suitable soft-computing paradigm, configures it.

The paper addresses the issues related to the integration of soft-computing paradigms within conventional development environments for embedded systems. The analysis is carried out at a behavioural abstraction level.

1 Introduction

Embedded systems are becoming a very pervasive presence in everyday life where they are used to perform more and more complex tasks. In this challenge towards more complex application several tools are available to help the designer to configure the system directly at a high level [1][2]. In general a top-down approach is considered, which allows the designer for coping with the complexity of the problem by decomposing it in simpler ones. Furthermore, a top-down description for a system eases upgrade interventions and components reuse.

A very high level description of the system is usually composed of informal information, properties and constraints. As the specification activity proceeds, the description of the system is translated into more precise statements by using several specification formalisms [1]. A formal verification is therefore necessary to check for possible inconsistencies coming from non-univocal specification translation and refining. Behavioural correctness has to be taken into account at the earliest stages of the development process: a prompt discovery of pathological functionality errors reduces the possibility of complex and expensive redesign phases. Testing the correctness at behavioural for a system is however more difficult (and often more critical, e.g., in life support systems) than at the implementation level (e.g., silicon area occupation) [2]. Since formal validation is often impractical to achieve an effective alternative is the validation through simulations: an executable model for the system is developed first and its behaviour is verified then on a limited number of relevant situations.

Four main phases can be devised in the conventional specification/design methodology for embedded systems: Formal specification, Algorithmic description, Hw/Sw codesign, and Implementation [2]. The aim of formal specification is to describe the system functionalities. This task is usually accomplished by decomposing the system in a pool of disjoint co-operating subsystem (or components), whose functionalities have the same effects of the original system. Although hypotheses on the final implementation of the component are no required, any physical constraint has to be expressed and checked as soon as possible in the design process to avoid that late detection of unsatisfiability imposes roll back and time-consuming iterations. This practice is more important in the design of embedded systems, where physical features are often critical. Therefore, the development of embedded systems requires both a rapid prototyping and an estimate of the physical requirements satisfaction. The second phase provides an algorithmic description satisfying the functionalities of each component. Hw/Sw codesign techniques are able, by exploiting physical constraints, to map the algorithmic description of the system onto an optimal hardware/software architecture. Once a behaviour description and a target architecture are given a detailed description of the system, suitable for the implementation is issued.

The methodology outlined above is supported by semi-automatic tools, often bundled in development environments (see, e.g., [1][2]). A common feature provided by such tools is the ability to deal with different description languages [3][4], hence leaving the designer to choose the more suitable description language for each component.

The increasing use of soft-computing paradigms is pushing the designers to look for flexible tools able to address the development of hybrid systems design composed of soft-computing and traditional paradigms. Unfortunately, the behavioural description of soft-computing paradigms can be difficulty tackled within an algorithmic formal framework. In fact, soft-computing components are easily described by means of (potentially) incomplete or uncertain specifications, such as historical data (or examples) or linguistic relations, and constraints on vague mathematical properties, such as smoothness. We will refer to such a component as *non-algorithmic* in contrast with *algorithmic* components, whose behaviour can be immediately described as a sequence of operations. The name must not mislead the reader since non-algorithmic must be intended at a behavioural level; only when a specific soft-computing paradigm has been configured its description becomes algorithmic.

0-7695-0619-4/00 $10.00 © 2000 IEEE

Soft-computing design follows a different and complex design chain, which encompasses model selection, parameters optimisation and model validation (e.g., as happens in neural networks). After configuration, a soft-computing paradigm becomes an algorithm and can be synthesised as any other; only at this stage the paradigm can be integrated within traditional embedded systems development tools [5].

The aim of this paper is to introduce a methodology for component specification and design based on examples.

The behavioural description of a component is example-based, where each example refers to I/O relation to be realised. The motivation derives naturally from the nature soft-computing paradigms [6] [7] [8]. An example-based framework supports an abstraction level description of soft-computing paradigms for their subsequent integration in a development environment for embedded systems. Nevertheless, the flexibility of available development environments to integrate different formalisms allows the achievement of a general framework where both algorithmic and non-algorithmic components can be described at a behavioural abstraction level. After behavioural specification, algorithmic components are directly expressed in formalisms suitable for Hw-Sw codesign processing, while for non algorithmic components a configuration process has first to be performed. After this last step, however, both algorithmic and non-algorithmic components behaviour is expressed in algorithmic formalisms and the design activity can proceed dealing with the system description as a whole to allow a global optimisation.

The main entities, which can be used to express system features, are related to behavioural characteristics, performance indices and structural peculiarities; we will name them as *variables* since their final values — usually constrained by specification — are often subjected to design choices. Our methodology groups variables in three main categories — Design, Behavioural and Physical specification variables — and confines their reciprocal influence in few defined evaluation moments. This practice notably simplifies the number of factors to be dealt with.

The structure of the paper is as follows. Section 2 introduces the variables used for specification and design. Specification variables are intended to code constraints (both physical and behavioural) given by the use the embedded system is conceived for, while design variables refer to the paradigm which will be used to implement the desired behaviour.

Section 3 describes the methodology to select, configure, and validate the soft-computinf paradigm. Since the design of an optimal model is an extremely complex problem, a heuristic-based trial-and-error procedure is considered.

2 Specification and design variables

A system can be described at different abstraction levels. For instance, a digital embedded system can be characterised at a behavioural level by means of the function it performs, at the architectural level by specifying the supporting digital architecture, at a physical level by describing the structure of its circuitry. In addition, some abstraction levels can be differently described according to the envisaged level of detail: a description for a device can be carried out, in fact, at processor or transistor level.

We are mainly interested in the behavioural abstraction level augmented with a set of low abstraction level constraints directly related the system implementation. In a wider sense, the suggested methodology deals with physical properties, the input/output behavioural description of the system and a set of high level a priori requirements.

Constraints related to physical properties are envisaged to model and constrain the hardware resources necessary to the final system. These specifications are usually stated early in the system specification phase, they address the system as a whole and only seldom are assigned to the components composing the system. Nevertheless, this information has to be preciously considered, as the impossibility to fit the designed architecture into physical constraints involves at least the redesign of a part of the system.

Specifications characterising the component behaviour are a collection of all information about the I/O relationship to be implemented. In a soft-computing training by example perspective, we have a set of examples and a set of mathematical features associated with the desired relation (e.g., smoothness, differentiability), as well as some environmental properties (e.g., data are affected by noise). The first step is related to the choice of the most suitable soft-computing paradigm for the application. If Neural Network paradigms are taken into account we could choose among several different networks such as feedforward NNs, Radial Basis Function, recurrent NNs, etc. Unfortunately, both paradigm and model selections are not trivial and straightforward: given the same set of examples and specification constraints, different designers will probably choose different solutions to satisfy the given requirements. Usually, the choice of the model's family is carried out on the basis of some a priori knowledge about the typology of the problem to be addressed and application-level requirements. Exiting tools aims at facilitate the designer's task by providing a collection of model families and several training algorithm (e.g, for NN [9][10]).

Specifications rarely impose a precise value for the entities defined onto the above domains; rather often they are expressed by means of constraints on a validity interval. Such flexibility allows the designers to maximise the system performances still satisfying the constraints. The real instance of the system, synthesised at the end of the design, depends on the values assumed by the entities. In this perspective, the entities are (independent) *variables* of the system realisation (which assumes the role of the dependent one). Physical properties will be coded using *physical specification variables*, presented in section 2.3, while component behavioural related entities will be called *behavioural specification variables* and will be presented in section 2.2. Entities strictly related to the paradigm are used in the model selection and configuration phases and can be considered the model design activity. Therefore, those entities, introduced in section 2.1, will be called *design variables*.

To be useful in model design, every constraint has to be mapped onto design variables. In this sense, there is an interaction between constraints defined on variables belonging to different abstraction levels; such interaction we is discussed in section 2.4.

2.1 Design variables

In this paper we focus the attention on computational paradigms which can be specified as:

- a model hierarchy, $\mathcal{M} = \{M_i\}$, which is a sequence of model families such that $M_i \subset M_{i+1}$. Each model family, M_i, represents the set of functions $M_i(\cdot, \theta)$ which can be obtained by allowing the vector of the free parameters of the model to span its domain. A particular model of the family is obtained by specifying the parameter vector θ, task carried out in the model configuration phase as discussed in the section 3.
- a complexity measures, p, is a measure for the number of free parameters of the model associated with the family M_p (e.g., the number of hidden units of a feedforward NN, the VC dimension of a family, etc...). This value can be used to characterise the complexity of the model.
- a set of input/output examples, $Z_c = \{(x_i, y_i)\}$.
- a figure of merit, J, to be used to configure/validate the model. For instance, it can be the mean squared error augmented with some penalty term such as the weight decay [6].
- a configuration algorithms, $Algo$, which provides the best parameter vector by optimising J. The choice of the configuration algorithm is related the structure of the model hierarchy. In fact, if the function to be configured is non continuous or the model is not differentiable, a gradient descent based optimisation algorithm cannot be directly used.

This paradigm class is large enough to enclose the most relevant soft-computing paradigms: feedforward NN and recurrent NN can easily fit into the above description. Soft-computing paradigms which generally present irregular internal structure (e.g.,bayesian networks, fuzzy system) can be enclosed provided some limitation on their structure in order to be described as a hierarchy $\mathcal{M}$.

The use of the above computational paradigm specification is motivated by the availability of theoretical results which allows different (feasible) realisation of the methodology. The methodology is however flexible enough to be extended to a large class of paradigms as well as to different behavioural specification (e.g., linguistic and or cause-effect relations).

2.2 Behavioural specification variables

Rising the abstraction level of the specification is the main goal of the present work. This is accomplished by providing the designer with concepts useful to describe model properties at behavioural level. We consider therefore the following entities:

- I/O data set, $Z_N = \{(x_i, y_i) | i = 1, \ldots, N\}$, which is the complete collection of the examples (possibly affected by uncertainty) pairs of the I/O relationship to be realised. In general, it contains the data set used to configure the model, Z_c. Depending on the actual configuration technique used, Z_N could be partitioned in order to devote some I/O examples for model selection and validation.
- accuracy, V, is a measure of the model $M_p(\cdot, \theta_p)$ performance; in a training from examples perspective it can be intended as the generalisation property. Mean square error or noise-to-signal ratio can be used to specify the accuracy constraints.
- robustness, R, is a measure of the model sensitivity to perturbations which affect the computations [11].
- mathematical features of the I/O relation, $\mathcal{C}$, contains all the analytical information related to the I/O relation which the system is supposed to realise, e.g., domain and codomain constraints, continuity and smoothness requirements for the model.
- noise, σ, addresses all the uncertainty "injected" structural with the application or induced by the designer choices (e.g., errors due to finite precision representation). We do not require a priori information about the structure of the noise.
- computational cost, $Cost$, is the computational complexity allowed to the system. For instance, this complexity will constrain the number of instructions in software realisation, or the number of functional units in hardware implementation.

2.3 Physical specification variables

Physical constraints are limitations due to the physical implementation of the above mentioned relation onto dedicated hardware. Examples are power consumption, latency time, number of pins and silicon area. The physical variables considered in this work can be grouped as:

- number of bits, $\#bits$, is the number of bits used for the representation and the processing of numerical quantities.
- number of pins, $\#pins$, is the number of I/O pins of the physical device.
- area, $\mathcal{A}$, is the maximum silicon area available to the component.
- power, P, is the power disposable to the component.
- latency time, L, takes into account the maximum time granted to the component to compute the output.
- economical cost, Υ, of the component realisation.

2.4 Interactions among constraints

A design phase is driven by two sources of knowledge: the specification and the experience of the designer. In the latter we include both the human experience and the knowledge contained in a database accessible to the designer. At this stage of the design process, the specification gives mainly behavioural hints, which are formalised as constraints on behavioural variables. In order to achieve a configured model, those constraints have to be mapped on design variables. Even if physical variables are at a lower abstraction level with respect to the others, the constraints on them are often defined in the earliest phases of the specification activity. Since the knowledge of constraints on physical variables has the effect of limiting the search space, it has to be carefully taken into account. Unfortunately, although constraints on physical variables can be evaluated only in latest phase of the design activity, they have to be considered as early as possible: the impossibility to satisfy a physical constraint can involve redesign of parts of the system with a cost increasing in time.

Then the next step is to face two constraints transformation: physical to behavioural constraints and behavioural to design constraints. The first transformation is necessary to not anticipate choices that would be better done in later design phases (Hw/Sw codesign, in particular). A premature implementation satisfying (locally considered) physical requirements could prevent a better global solution.

Tables 1 explain the relationships among variables. The presence of the symbols "$\nearrow$", means that the column variable is influenced by the row one. Similarly, "$\swarrow$" indicates that the column variable influences the row one. "x" means that row and column variables are affected by a mutual dependency relation.

We consider only direct relations. For example, the number of bits impacts onto the achievable accuracy; this effect is not taken into account because the influence of $\#bits$ on V is mediated by the noise inflated by quantisation. In other words, $\#bits$ will influence σ, which will limit V, and only these two direct influence relations will be considered.

We excluded the Υ from the physical variables considered in the tables 1 because — even if it can be critical to the design choices — it is not directly related to any of the other variables. It relates only with the implementation techniques that will be chosen. Its influence on the design activity will be discussed later in 3.4.

	R	V	σ	$Cost$	$\mathcal{C}$	Z_N
$\#bits$			$\nearrow$			
$\#pins$			$\nearrow$			
$\mathcal{A}$				x		
L				x		
P				x		

(a)

	R	V	σ	$Cost$	$\mathcal{C}$	Z_N
$\mathcal{M}$	x	x		x	x	
$Algo$					$\swarrow$	$\swarrow$
J		x	x	$\swarrow$	$\swarrow$	$\swarrow$
p		x		x		$\swarrow$
Z_c		$\nearrow$				$\swarrow$

(b)

Table 1: Variables dependencies: (a) physical vs. behavioural, (b) design vs. behavioural.

3 Design Cycle

The configuration phase — as given in figure 1a — is divided in four main activities: Model Selection, Model Configuration, Model Validation, and Component Validation.

Two kinds of information constitute the input and the output of this phase: specification constraints, $\{C_i\}$, and validation judgments, $\{(E_i, T_i)\}$. An *intransigence* judgement, I_i, is associated to each specification constraint, C_i and indicates the strength of constraints. This avoids the risk to give the same importance — i.e., to profuse the same amount of resources in the attempt of assure their satisfability — to behavioural and physical constraints. Unless a previous choice of the designer about the architecture and the implementation technique, the physical constraints are generally less predictable at the actual level of abstraction. The intransigence value can be assigned by the designer, or can be assigned and revised according to the constraint history. In fact, some kind of constraints derived from system constraints can be initially considered weak, and reinforced as the design activity proceeds. For instance, a constraint on latency for a system can be distributed on its components if they are connected serially: the real constraint is not the estimated latency time available to each component, but the sum of them. Its value cannot be accurately estimated before the configuration of all its components. Hence, even if a target component latency time can be devised, there is no gain in being very intransigent about slower solutions. Once the other components are configured, the total time can be redistributed more reasonably, and the intransigence value associated to every component latency time can be raised.

Similarly, validation of the configured model produces a set of triples $\{(C_i, T_i, E_i)\}$, where C_i is a constraint, T_i is an implementation technique, and E_i is a (qualitative) judgement about its *easiness* to satisfy the specification constraint.

At a high abstraction level, behavioural constraint validation can be sufficiently accurate, while it is difficult to verify physical constraints precisely. However it is possible to give some confidence measure on the possibility to implement the configured model by satisfying the physical constraints. Because of the problem's nature, the judgement cannot be anything but heuristic-based.

This information (intransigence and easiness) have to be appropriately combined in order to formulate a definitive judgement of the model acceptability.

The linguistic nature of the intransigence and the easiness of judgements seem to push for a fuzzy implementation of the decisional device, even if other techniques can be efficiently used. However, a proposal for the implementation of an automated tool based on the discussing methodology is out of the scope of this work.

In the following, the four above mentioned activities will be discussed more deeply.

3.1 Model Selection

The Model Selection activity summarises the information derived from the specification constraints $\{(C^i, I^i)\}$ (figure 1a) as well as previous configuration attempts (which can be stored in the previous attempts database, PA-DB) in order to formulate a proposal regarding the model hierarchy, the complexity of the considered model, the use of the available examples, the configuration algorithm, and the figure of merit to be optimised. This activity makes use of knowledge about the relations which bind the variables (as discussed in section 2.4) and the properties of the known hierarchies (stored in a database of a priori knowledge, APK-DB).

Behavioural constraints can guide the model selection. For instance, constraints onto the smoothness property — which are coded in C — can be integrated in J [12].

Moreover, variables design can influence each others: hence, similarly to constraints translation between variables of different kind, new (temporary) constraints can be generated by design choices. For instance, the choice of p is influenced by the number of examples available for the configuration: generally, the ratio between number of model's parameters and the data set cardinality — related to the uncertainty on the examples — provides a clue about the (un)reliability of the model's configuration. Thus, if a constraint on V moves the model proposal towards a certain value of p a temporary constraint on Z_c has to be set up.

This activity collects the output of other activities and manages the whole configuration process. In particular, it provides a model proposal to be configured to the Model Configuration activity. The configured model $M_k(\cdot, \theta_k)$ is passed to the Model Validation activity, to which possible validation information has to be provided (e.g., a validation data set, Z_v, for Cross Validation purposes). Model Validation computes possibly performance indices P_i and provides a collection of judgements E^b about the willingness of the configured model $M_k(\cdot, \theta_k)$ to satisfy behavioural constraints C^b. Whenever a reliable model $M_A(\cdot, \theta_A)$ has been found, Model Selection has to submit it to the Component Validation activity, in order to verify the feasibility of the implementation. Physical constraints C^p are here evaluated in order to produce, a collection of judgements E^p about the easiness in realising the component reliably for every possible implementation technique T^p.

Model Selection has to combine the intransigence and the easiness judgement to formulate a global evaluation for every configured model. Eventually the more promising models $M_S(\cdot, \theta_S)$, associated with correspondent constraints satisfaction ability and performance indices.

3.2 Model Configuration

The Model configuration activity is mainly an optimisation engine. It receives an instance of the design variables and produces an instance of the model parameters vector which minimises the given figure of merit J. We are not interested in deeply investigate how this task is performed since a wide literature is available on the argument [6] [13]. Particular operations to customise the optimisation procedure are coded (by the Model Proposal activity) in $Algo$.

3.3 Model Validation

Model Validation is a necessary step to verify the configured model's with respect to the behavioural specification constraints. It consists of two main steps: validation of generalisation and behavioural constraints.

Model Validation has to verify that the function realised by $M_k(\cdot, \theta_k)$ would be a faithful generalisation of Z_N. Several techniques (e.g. Cross Validation) can be applied. Whenever possible, a test on the statistical distribution of the error resulting as difference between the $M_k(\cdot, \theta_k)$ and the target function (if available, or the data set) can give interesting information about the reconstruction quality.

Not all behavioural constraints can be translated in model design constraints. For instance a non differentiable performance index cannot be included in J if $Algo$ is gradient-based. Therefore, this activity will verify exactly that the performance indices that were imperfectly coded in J are actually satisfied.

It outputs, for every configured $M_k(\cdot, \theta_k)$, a vector of performances indices, P_k, which can be useful for comparison between different models and can be used as tie-break discriminant. For instance, they could be measures of $M_k(\cdot, \theta_k)$ robustness [11].

A minimal output of this activity has to be a set of judgements about the easiness (which could mean confidence) with which the behavioural constraints will be satisfied $\{(C^b, E^b)\}$.

3.4 Component Validation

Once a faithful model $M_A(\cdot, \theta_A)$ is selected and configured, the effectiveness of its realizability has to be tested. It is performed in two steps. First, some perturbations are applied to its parameters to take into account the approximation caused by the physical implementation — e.g. I/O bounds, binary representation, finite precision representation. These perturbations are based on a subset the physical constraints — namely the ones which refer to $\#bits$ and $\#pins$ — and are hypothesised to be small. Then, providing some hypothesis regarding the subsequent implementation of the component, the physical constraint satisfability can be tested. Even if an accurate and precise judgement is impossible to achieve, a qualitative judgement can be useful to drive the design activity. The constraints on Υ can effect in the choice of the implementation technique. This activity is schematised in figure 1b.

4 Conclusions

In this paper we presented a general methodology to specify and design soft-computing components in a way suitable to be included in a general framework for embedded systems. This methodology allows to describe the components at a behavioural abstraction level by taking into consideration known physical constraints, and to generate an executable description of the system.

The behaviour of the component is described mainly through a set of examples of the I/O relation which is intended to synthesise. The uncertainty degree of the description is controlled by means of other factors, such as accuracy or robustness of the desired model. Hints on the feasibility of the component can be derived by the constraints on the physical resources allocable for it. A low value of intransigence on the physical constraints allows an initial rapid prototyping to test the correctness of the behavioural specification. Besides, explicit formalisation of the interactions among specification and design variables allows also to easily document the design choices.

Future efforts will be directed to the implementation of a CAD support to manage the integrated framework for embedded systems development.

References

[1] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli, "Design of embedded systems: Formal models, validation, and synthesis," *Proceedings of the IEEE*, 1997.

[2] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*. P T R Prentice Hall, 1994.

[3] J. Davis, R. Galicia, M. Goel, C. Hylands, E. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, and Y. Xiong, "Ptolemy II: Heterogeneous concurrent modeling and design in java," tech. rep., University of California, 1999.

[4] Mentor Graphics, *Renoir '99 Presentation*, 1999. http://www.mentor.com/renoir/presentation/renoir99.zip.

[5] B. Lu, B. L. Evans, and D. V. Tošić, "Simulation and synthesis of artificial neural networks using dataflow models in Ptolemy," in *Proc. Seminar on Neural Network Applications in Electrical Engineering*, pp. 84–89, 1997. Belgrade, Yugoslavia.

[6] J. Hertz, R. Palmer, and A. Krogh, *Neural Computation*, vol. 1. Reading, MA, USA: Addison-Wesley, 1991.

[7] W. Buntine, "A guide to the literature on learning probabilistic networks from data," *IEEE Transactions On Knowledge And Data Engineering*, vol. 8, pp. 195–210, 1996.

[8] M. Russo, "FuGeNeSys: A fuzzy genetic neural system for fuzzy modeling," *IEEE Transactions on Fuzzy Systems*, vol. 6, pp. 373–388, August 1998.

[9] H. Demuth and M. Beale, *Neural Networks Toolbox User's Guide*. The MathWorks Inc., January 1998.

[10] *SNNS Stuttgart Neural Network Simulator — User Manual, Version 4.1*, 1995. http://www.informatik.uni-stuttgart.de/ipvr/bv/projekte/snns/UserManual/UserManual.html.

[11] C. Alippi, "Feedforward neural networks with improved insensitivity abilities," in *IEEE-ISCAS99*, 1999.

[12] F. Girosi, M. Jones, and T. Poggio, "Regularization theory and neural networks architectures," *Neural Computation*, vol. 7, 1995.

[13] R. A. H. G. Kan and G. T. Timmer, *Global Optimization: A Survey*, vol. 87 of *International Series of Numerical Mathematics*. Basel, Switzerland: Birkhauser Verlag, 1989.

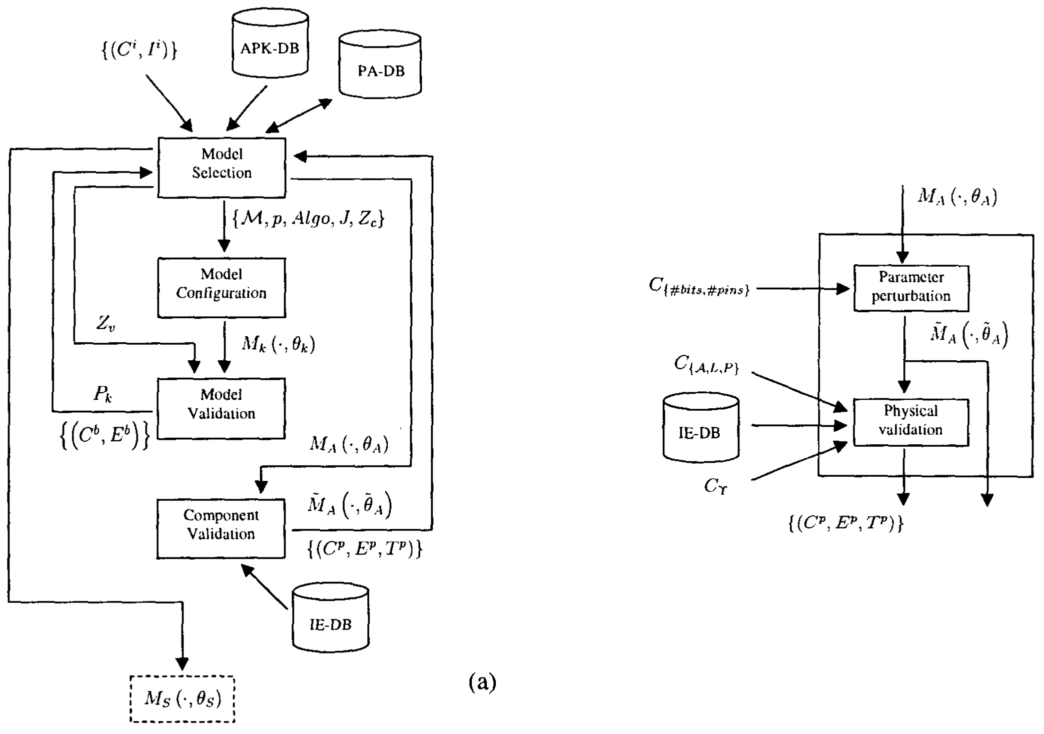

Figure 1: Design cycle (a), and Component Validation (b).