The Design Space Layer: Supporting Early Design Space Exploration for Core-Based Designs*

Helvio P. Peixoto and Margarida F. Jacome
ECE Dept., The University of Texas at Austin
Austin, TX 78712
{peixoto|jacome}@ece.utexas.edu

Ander Royo and Juan C. Lopez
ECE Dept., Technical University of Madrid
Madrid, Spain
{ander|lopez}@die.upm.es

ABSTRACT

A novel library layer, called the "design space layer," is proposed, aimed at supporting both, IP-based and traditional "in-house" design methodologies, during early design space exploration. Strategies for effectively pruning the large design spaces characteristic of system-on-chip designs, and for transparently retrieving information on cores adequate for implementing the system components, are supported by the proposed layer. The layer is self-documented and highly compartmentalized into hierarchies of classes of design objects, and is thus easily scalable. A design space layer developed for encryption applications is presented and discussed in some detail.

1 Introduction

The trend towards developing *core-based*, *system-on-chip* solutions for complex application specific systems is clearly irreversible. Increasing the level of design integration is quite attractive from a reliability, power consumption, and unit-cost perspective. The use of cores, i.e., macro-cells developed by third party IP providers, can lead to significant decreases in design cost and in time-to-market. Reflecting the tremendous opportunities created by this emerging trend in the design of application specific systems, the number of IP providers is booming -- more than seventy two are listed in [1], offering over 1000 reusable blocks.

This paper proposes a new library *layer*, to be implemented on top of conventional reuse libraries (see Fig. 1), aimed at supporting both, *IP-based* and traditional "*in-house*" design methodologies, during early design space exploration. Specifically, the objective is to assist designers in systematically considering relevant alternative implementations for the various components of a system-on-chip architecture. Accordingly, it will be shown that the proposed library layer, called the *design space layer*, allows for cores, that are good candidates for implementing specific system components, to be quickly and transparently selected from reuse libraries. Furthermore, when no adequate cores are available, it will be shown that the proposed layer can still assist the *conceptual design* of the corresponding system components, i.e., the specification of the fundamental design options that should be adopted for such components, in order to meet the required performance, silicon area, power consumption, etc.

As its name suggests, the proposed layer creates a representation of the *design space*, i.e., the space of all feasible (alternative) implementations for the object under design. Note that such a representation is not *explicit*, since the actual alternative designs may or may not exist in the reuse libraries that underlie the layer. The design space is thus *implicitly* represented by *discriminating* the areas of design decision, or *design issues*, that are on the basis of the creation of such alternatives. An example of one such design issue is "implementation style," discriminating between hardware and software designs. Note that the cores available in the reuse library correspond to "points" in the design space represented in the layer. Accordingly, they are logically indexed (i.e., referenced) via these same areas of design decision. Some regions of the design space may be thus populated by a large number of cores, while others may not.

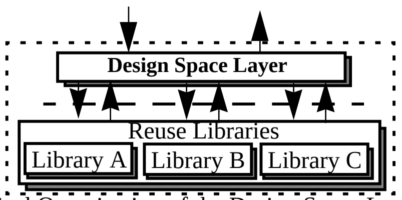


Fig. 1 Logical Organization of the Design Space Layer and the Reuse Libraries

* This research project is sponsored by NSF, under CAREER Grant No. MIP-9624321.

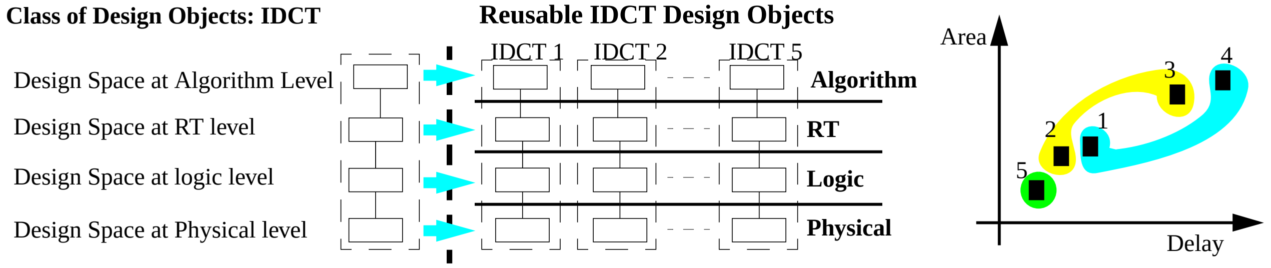
The proposed layer is aimed at supporting strategies for systematically pruning large design spaces and for retrieving information on available cores (i.e., reusable designs) complying with the system's requirements and with the design decisions made so far. Indeed, each design decision made with respect to a specific architectural component, during conceptual design, corresponds to a pruning of the component's design space. The reusable designs that fall outside the selected region, i.e., those cores which do not comply with such a decision, are immediately eliminated from consideration. Critical information on the *set* of reusable designs that do comply with the decision, including ranges of performance and power consumption, can be then directly provided to the designer. In some cases, directly reusable designs may not be available in the reuse libraries, i.e., the set of "readily available" design points (within the selected design space region) is empty. In such cases, the proposed design space layer still assists the designer in undertaking conceptual design, adequately supported by early estimation tools, whenever such tools are available.

As shown in Fig. 1, the proposed design space layer can be logically connected to any number of reuse libraries, i.e., transparently index/reference designs residing in different libraries. Ideally, each design environment should thus develop its own design space layer, tailored to the application domains of interest, and then use such a layer to reference available cores, stored in reuse libraries maintained by the IP-providers themselves.

The remainder of the paper is organized as follows. Section 2 gives an overview of the fundamental support mechanisms provided by the proposed layer for assisting conceptual design and core selection during the definition of system-on chip architectures. Previous work is discussed in Section 3. The proposed modeling framework is described in more detail in Section 4. Section 5 presents a detailed case study on the development of a design space layer for *encryption applications*. Some conclusions are given in Section 6.

2 An Overview of the Design Space Layer

As alluded to in the previous section, the proposed layer creates an implicit representation of the design space, i.e., the space of all possible alternatives for the design at hand. Such a design space representation is based on the design formalism described in [2]. The cornerstone of this formalism is the concept of a *class of design objects*. Examples of classes of design objects are "Adders," "Inverse Discrete Cosine Transform" (IDCT) blocks [3], "MPEG II encoders/decoders" [4], and "embedded RISC processors."



Class of Design Objects: IDCT
Design Space at Algorithm Level
Design Space at RT level
Design Space at logic level
Design Space at Physical level
Reusable IDCT Design Objects
IDCT 1
IDCT 2
IDCT 5
Algorithm
RT
Logic
Physical
Area
Delay
1
2
3
4
5

(a) Design Space Layer Based on level of Abstraction (b) Library of Reusable Designs (c) Evaluation Space

Fig. 2 Design Space Layer Strictly Based on Abstraction

A class of design objects is an abstraction used to implicitly define the design space encompassing all feasible implementations of a specific behavior or functionality. For example, the "Adder" class of design objects abstracts all possible adder implementations, each of which is a point in the design space, called an adder *design object*. Alternative designs are differentiated within the "Adder" *class* by the various areas of design decision, or *design issues*, that are relevant to the design of an adder. An example of one such design issue is "logic style", with the corresponding design options given by: "ripple-carry," "carry-look-ahead," "carry-save," etc. Another example is "layout style," with the corresponding options given by: "standard-cell," "gate-array," etc.

Since the design space for complex application specific systems tends to be large, it is important to provide basic mechanisms to enable an effective and systematic pruning of the space. In what follows, we argue that the "traditional" concept of *design abstraction*, while adequate for driving the top-down design process, is *inadequate* to support the early design space exploration performed during system-on-chip architectural definition. We then discuss the generalization/abstraction features provided in our layer, aimed at supporting an effective pruning of the design space.

2.1 On the Inadequacy of Traditional Design Abstraction

In top-down design methodologies, the design process is decomposed into a number of design steps, each of which increasingly refines/details the structural and behavioral descriptions of the object under design. Specifically, in each step, the design data characterizing the design object at a given level of abstraction is refined into an equivalent representation at a lower level of abstraction. Examples of levels of abstraction traditionally adopted in *digital design* are algorithm, register-transfer (RT), logic, and physical. At the end of the design process, the design data is naturally partitioned into such levels of abstraction, creating the various views of the design object.

However, such a compartmentalized notion of abstraction is *not* the most convenient way to organize data in order to support early design space exploration. Fig. 2 will be used to illustrate this point. Consider for example that five IDCT hard-cores are available in the reuse library. The boxes shown in Fig. 2(b) symbolically denote how the *detailed design data* for each of the IDCT cores would be *partitioned* into the four levels of abstraction: algorithm, RT, logic, and physical.

For the sake of the discussion, let us assume that the design space layer would be defined strictly with respect to this hierarchical organization of data. As shown in Fig. 2(a), we would in this case define four "sub-classes" for the IDCT class -- the first sub-class would represent the *design space of the IDCT* at the *algorithm level*, and the second, third, and fourth sub-classes would represent it at the RT, the logic and the physical levels, respectively. So, while each box in Fig. 2(b) denotes the *detailed design data* of a particular view of an IDCT core, the boxes in Fig. 2(a) enumerate the *areas of design decision* (and corresponding design options) that have to be considered at each level of abstraction, when designing an IDCT. Accordingly, each of the IDCT cores would thus be discriminated first at the algorithm level, then at the RT level, etc. Naturally, one would still need to decide on the specific *sub-set* of design issues "relevant enough" to be considered during conceptual design (from among those defined at each level). This important point is addressed next, while discussing the inherent limitations of this organization.

By exploring the design space in such a way, there would be no guarantee that the designer would be quickly and coherently guided to the best candidate IDCT core(s). For example, Designs 1 and 4, with quite distinct area/performance characteristics (see Fig. 2(c)), could very well be different implementations of the *exact same IDCT algorithm* (say, one using a 0.35μ standard cell library, and the other using a 0.7μ standard cell library). So, the design space regions selected by initially exploring only the algorithm design space could map into quite uninformative regions in the evaluation space, as symbolically shown in Fig. 2(c).

The important observation to be made is that, some design issues may only explicitly appear in the design data at the logic or *lower levels of abstraction* (e.g., "layout style," "fabrication technology," etc.), yet they may still have a *major impact* on performance, silicon area, and power consumption. Recall that, the ultimate objective of the design space exploration performed during conceptual design is to specify the *fundamental design options* to be adopted for a design object (during its detailed top-down design), so that its required performance, power consumption, etc., can be achieved. Thus lower level decisions such as these may actually need to be considered very early in design space exploration process.

2.2 A Design Space Layer Based on Specialization/ Generalization

The decision on which design issues need to be considered during conceptual design (and used to discriminate the designs) should be based on their impact on the *figures of merit* of interest -- this will allow for a coherent organization of designs, reflecting their actual proximity in the *evaluation space*. Consider again the five IDCT cores. (Assume, for simplicity, that all such cores can support the specific "word size," "precision," and other requirements posed by the application.) Intuitively, one would expect to see Designs 1 through 5 discriminated into the clusters/ groups as shown in Fig. 3(b). This means that the designer should first be presented with the design issues that result in the significantly different area/performance ranges exhibited by the cluster {1,2,5} and the cluster {3,4}. Only after the designer commits to one of these ranges, by adopting the design options that are *common* to the designs within the corresponding cluster, should options that further discriminate between the designs within a single cluster be considered. This is symbolically shown in Fig. 3(a).

We now discuss how this is achieved in our proposed design space layer. Let us revisit the notion of a class of design objects, and introduce the notion of a *generalized* class of design objects. A generalized class of design objects is one that considers only a *subset* of the design issues that need to be addressed in order to complete conceptual design.[1] A hierarchy of such generalized design classes can be thus constructed, starting with more general classes and increasingly specializing them, thus defining increasingly "specific" design space regions, and corresponding families of design objects, see Fig. 3.

This generalization/specialization hierarchy is to be constructed based on common functionality at a desired level of abstraction/detail, and also on similarities of alternative designs with respect to achievable ranges of performance, power consumption, etc. So, as shown in Fig. 3(a), all IDCT designs would first be *generalized* into a single family of design objects, since they implement the same basic function. The IDCT family would then be further specialized, by identifying the design issues and corresponding options *on the basis* of the similar ranges of performance and silicon area exhibited by the two clusters of designs, as shown in Fig. 3(a), and so forth.

In this paper we argue that the ability to create such generalization/specialization hierarchies is fundamental to supporting/enabling a systematic exploration of large design spaces. Modeling mechanisms are proposed in order to systematically define such hierarchies (see Section 4).

A simplified design space layer for the IDCT class is given in the Fig. 4. Recall that such a design space layer is supposed to define a strategy for systematically pruning the design space for IDCT designs, and simultaneously accessing available IDCT cores. The "top" IDCT generalized class contains the definition of the transform -- all available IDCT cores are thus indexed through this node.

Many different IDCT algorithms can be found in the literature.[3] Such algorithms, obviously all derived from the same basic mathematical definition of the transform, have however different critical paths, different numbers of operations, precisions, etc. The speed, power consumption, and other application-specific requirements, should thus ultimately dictate which algorithmic implementation of the transform is most suitable for each design.

The proposed layer should thus support such an *algorithm-level* design space exploration, whenever required. However, as shown in Fig. 4, the design issue "implementation style" precedes the design issue "algorithm," suggesting that: (1) its overall impact on performance is more significant than that of the "algorithm" design issue; (2) it may impact the definition of the options for the "algorithm" design issue, e.g., same of the algorithms may only make sense when implemented in hardware; and/or (3) it may impact the comparative efficiency of such algorithms.

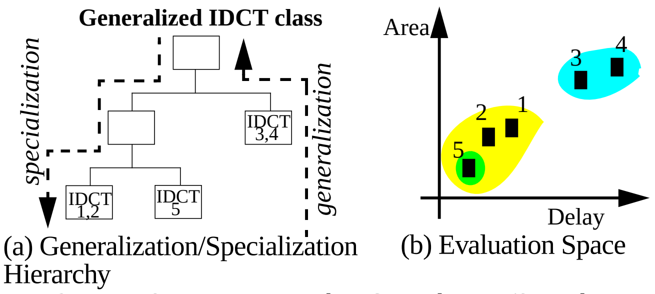


Fig. 3 Design Space Layer Based on Generalization/Specialization

1. The definition was simplified, for clarity. In section 4 we provide the precise definition of a generalized class of design objects.

So far we have mostly focused on hardware designs but software implementations pose no significant challenge to our modeling framework. For example, consider again Fig. 4. The design issue to be used for further discriminating the "software" generalized class would be "programmable platform," with options such as "embedded RISC processor" and "embedded digital signal processor." These platforms would be then further discriminated. The software routines and the processor cores themselves, would be the "reusable designs".

This concludes our overview, which, for clarity, has been mostly qualitative. Still, it should be clear by now that the above design space layer should not be defined in an ad hoc way. Systematic mechanisms are needed for creating self documented, scalable design space layers, tailored to the target application domains and the needs of each individual design environment. This is the topic of Sections 4 and 5.

3 Previous Work

The *Virtual Socket Interface* (VSI) alliance [5] was recently created to address the challenges posed by core-based system-on-a-chip designs. Even if the most well known effort of the VSI alliance has so far been the definition of a *standard on-chip bus*, working groups have also been formed in the areas of system level design, manufacturing related test, intellectual property protection, mixed signal design, and implementation/verification. The VSI alliance efforts towards defining required and recommended design practices for IPs, covering logic design, physical design, test, and communication protocols (bus interfaces), will facilitate the task of developing the design space layer proposed in this paper. Specifically, the produced standards and recommendations will be instrumental in defining the complete set of requirements and areas of design decision that must be specified for each reusable core.

A number of reuse methodologies have been reported in the literature, geared towards defining parametrized HDL components so as to increase the opportunities for reuse (e.g, [7]). Note that such *design data* should reside in a *reuse library* (see Fig. 1), and be then properly indexed/referenced, via the design space layer proposed in this paper. In [8], a methodology for *reuse by adaptation* was proposed which relies on a feature-based model. The selection of design objects is performed using heuristic similarity functions. The aim and scope of our work is different from the above in two fundamental ways. First, in order to create an "open" design space layer, capable of referencing populations of cores which are constantly increasing, or changing, our focus is placed *not* on the design objects themselves, but on *design space* that contains such objects. Moreover, our emphasis is on *supporting trade-off exploration*, as opposed to automating the selection of reusable designs. In [9], an environment is proposed to support *early estimation* of performance, power consumption, etc. The attractiveness of this approach relies on its "light weight," i.e., on the small effort required to create the infrastructure required by the self-contained estimation environment. As discussed in Section 2, our proposed design space layer has an *aim* and *scope* different from those in [9].

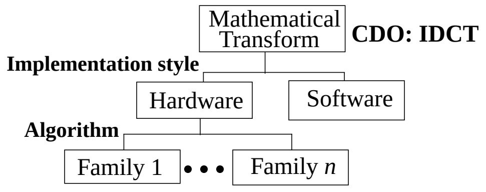


Fig. 4 Organization of Design Space for an IDCT

4 A Modeling Framework for Creating Design Space Layers

For simplicity, classes of design objects have so far been presented as "atomic black-boxes." They are not, however, the finest grain modeling construct provided in our design space representation. At its finest level of granularity, the *design space* is actually abstractly characterized (i.e., discretized) by a set of behavioral and structural *properties* or *features*. Such properties are meta-data used to organize the myriad of design data characterizing each given design object. They can be classified into:[1]

- behavioral and structural *descriptions*, used to define the structure or intended behavior of design objects at various *levels of design abstraction* (for example, an RTL behavioral description, written in VHDL or Verilog.);
- design *requirements*, specifying the target performance, area, etc., for the object under design, as well as other "problem givens," such, for example, the required "word size" for an "adder" or "multiplier," and the required "precision" for an "IDCT" block;
- design decisions or *restrictions* made during conceptual design with respect to critical *design issues*, such as the "implementation style" for the various system components.

The properties alluded to above are then organized into a hierarchy of increasingly *specialized classes* of *design objects*. So, groups/families of alternative algorithms and/or implementation styles, for example, can be *merged/collapsed* (i.e., "generalized") to facilitate the *initial exploration phases*. By doing so, they become selectable as a "family of design alternatives," and can then be incrementally discriminated, as conceptual design progresses.

It is important to note that the specialization alluded to above is actually realized at the design issue (i.e., property) level. *Generalized design issues*, collapsing families of alternatives, will thus precede more *detailed design issues*, whenever that makes sense from the point of view of aggressively pruning the design space. For example, the option "hardware" shown in Fig. 4 for the IDCT class, actually results from collapsing all "layout style" and "implementation technology" alternatives, enumerated at lower levels of the specialization hierarchy. The generalized design issue "implementation style" is thus created in order to differentiate, up-front, the entire family of hardware alternatives, from the software family.

This organization of properties into a hierarchy of classes of design objects defines the design space for the target set of application domains. Note that a class of design objects (CDO) may contain at most one generalized design issue. Each *option* of a generalized design issue defines a new CDO descending from the original CDO, i.e., defines a *specialization* of the previous class. CDOs with no generalized design issue are thus the leafs of the hierarchy.

As mentioned above, this hierarchy of CDOs provides also a basic schema for classifying and indexing families of cores residing in the *reuse libraries*. Reusable designs (i.e., design objects of a certain class) are thus no longer monolithically represented and accessed. Instead, as conceptual design progresses, it is possible to access families (i.e., groups) of such objects, associated with desired *ranges* of performance, power consumption, etc., down to single properties of individual design objects (i.e., down to a single property "value," say, an HDL model). Thus reuse becomes naturally *merged* with design, in that it is not only a mechanism to minimize the need for re-design, but also a mechanism to assist *conceptual design* and *estimation*, throughout the entire design process.

Note that the properties discussed above may not be (and *do not need to be*) entirely independent from each other. For example, the designer may make a design decision that, later on, proves to be *inconsistent* with the performance requirements, suggesting that a consistency relationship should be established between such properties. Moreover, some design issues may have a more significant impact on the figures of merit (i.e., requirements) of interest than others, suggesting that such design issues should be *partially ordered* in order to allow for a systematic exploration of the design space. A single modeling construct, called *consistency constraint* (CC), is used to express such ordering and consistency relationships among properties.

CCs are defined by an *independent* set of properties, a *dependent* set of properties, and a *relation*. The dependent set can only be addressed by the designer after the independent set has been addressed. Moreover, when the independent set is modified, the dependent set needs to be re-assessed. A relation defines the type of dependency between both sets of properties. The relations defined within a CC can be quite different in nature. Namely, they may be stated exactly, using first principles, or be heuristic. They may be quantitative or qualitative. Moreover, they may directly state inconsistencies between specific design options, or identify inferior (i.e., dominated) combinations of such options. Detailed examples are given in Section 5.

As will be shown, CCs are one of the key mechanisms provided in our modeling framework to support a systematic exploration of large design spaces. CCs allow for establishing general consistency relationships among design options across design issues, and/or among design issues and requirements. CCs also allow for establishing a partial ordering among design issues, considering the degree to which they impact key requirements, such as performance, area, etc. Finally, CCs establish also the precise utilization context for early estimation tools, as illustrated in Section 5.2.

Before concluding this section, we will further clarify the difference between the coarse-grained "partitioning" of the design space implemented by generalized design issues, and the finer-grained design space exploration strategies, implemented by "regular" design issues and consistency constraints. On one hand, the options of a generalized design issue typically collapse a number of options from a number of design issues into single "artificial" (generalized) alternatives -- this is thus a coarse partitioning of the design space, discriminating among broad alternatives, created with respect to *important* "commonalties/ similarities" in: (1) functionality; and (2) achievable figures of merit. CCs, on the other hand, establish a finer-grained, *trade-off oriented* organization of the design issues defined within each class of design objects. This important difference will be illustrated in the following section, in the context of our case study.

5 A Detailed Case Study on Cryptography

In what follows we present a case-study on the application of our modeling framework to cryptography applications, e.g., digital signature and public key encryption and decrypting. [10] These applications use, as a basic operation, the *modular exponentiation,* M^E mod N, typically performed on integers with values up to 2^{1000}. Modular exponentiation, in turn, uses *modular multiplication*, *A*x*Bmod M*, as a basic operation. Due to space limitations, most of our discussion focuses on modular multiplication.

This case study aims at demonstrating two fundamental points: (1) the proposed *generalization hierarchies* are instrumental to effectively *pruning the design space* so as to quickly identify adequate design space regions; and (2) a *trade-off oriented*

1. For a more detailed discussion on this classification, see [2].

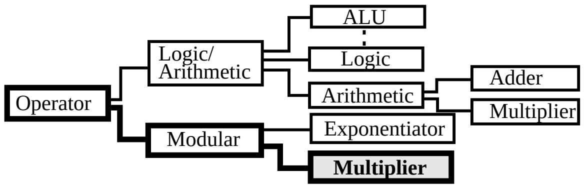


Fig. 5 Organization of Classes of Design Objects

design space exploration can be adequately performed at each *stage* of the traversal of the generalization hierarchy, i.e., within the increasingly narrower/smaller design spaces defined by each specialized class in the hierarchy.[1] In order to demonstrate points (1) and (2) above, the specifics of the design space layer developed for modular multiplication will be presented assuming that a core for implementing modular multiplication is to be selected, so as to meet the specifications given in [11] for a modular exponentiation coprocessor to be used in cryptography applications.

A number of alternative *hardware* modular multiplier designs was developed for this experiment, using the Synopsis Design Compiler and the LSI Physical Design Tools and Toolkit for the 0.35μ G10 library -- these designs are summarized in Table 1 (page 8). The *software* modular multipliers considered in our case study comprise a set of *C* routines and a set of highly optimized *assembly* routines, both executing on a Pentium 60, as reported in [12]. These alternative implementations are thus the cores (i.e., reusable designs) in our experiment.

We start by discussing the properties defined in the design space layer that we have developed, and their organization into a generalization hierarchy. In Section 5.2 we discuss the consistency constraints established among such properties.

5.1 Properties and Generalization Hierarchy

5.1.1 Algorithms for Modular Multiplication

Modular multiplication can be performed using the "Paper and Pencil" multiplication algorithm, followed by a *mod M* reduction. This algorithm, although very intuitive, is usually not used because of the size of the partial products and the carry ripple length of the parallel additions. Brickell proposed in [13] a more efficient algorithm. It is based on the paper and pencil method but starts with the most significant digit of *A* and performs a *mod M* reduction at every partial product. Finally, for applications in which the module *M* is known to be *odd,* another algorithm, due to Montgomery [14], can potentially achieve even better efficiency. Fig. 10 shows the behavioral description of the Montgomery algorithm. Although it requires a pre-computation of the Multiplicative Inverse for the computation of the quotient *Q* (line 4) and a post processing (lines 5 and 6), the basic iteration step has great potential for speedup optimizations.

5.1.2 Hierarchy of Classes of Design Objects

Fig. 5 shows part of the hierarchical organization of classes of design objects (CDOs) used to define the design space layer for cryptography applications. The first three levels of the specialization hierarchy are thus defined with respect to commonalties in functionality. Specifically, a generalized class "operator" is first defined, then it is specialized into "logic/arithmetic" and "modular" operators and, finally, the various operators defined for each class are discriminated.

In what follows we discuss in detail the *Operator - Modular - Multiplier* (OMM) CDO, shown in gray in the figure. Because of the *inheritance hierarchy* (see path in bold), the properties to be discussed in the following sections may be part of the CDO in

1. Note that, since generalized design issues *partition* the design space, each specialized class in the generalization hierarchy defines a design space region contained within that defined by its predecessor class.

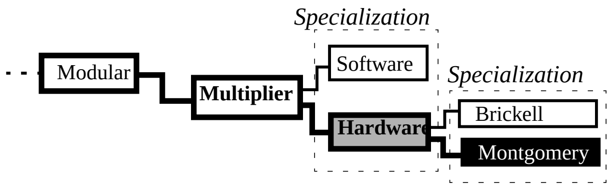


Fig. 7 Generalization Hierarchy for Modular Multiplication

question or of any of its ancestor classes. (Due to space limitations, only a sub-set of representative properties is presented.)

5.1.3 Requirements for the OMM CDO

Fig. 8 lists some of the requirements specified for the "Operator - Modular - Multiplier" CDO, with the corresponding values taken from the specification in [11].[2] Req1, Req2, and Req3 are self explanatory. Req4 asks the designer to specify if the modulo is known to be odd. For cryptography applications the modulo is known to be prime, and thus odd, so the option "Guaranteed" is selected. The target performance is loosely specified in terms of significant performance points. For example, a modular multiplication with a 768-bit operand/modulo should take no more than 8 μs, as indicated in Req1 and Req5.

5.1.4 Design Issues for the OMM CDO

The only Design Issue defined for the "Operator - Modular - Multiplier" CDO is "Implementation Style," with options "Hardware" and "Software." (see DI1 in Fig. 8). As in the IDCT example, "Implementation Style" is defined as a generalized Design Issue, i.e., it partitions the design space into two sub-spaces which, from then on, will be explored independently. Observe that such partitioning is justified by the fact that hardware and software designs offer radically different ranges of performance for this application, and thus *fine-grained trade-offs* cannot be explored based on this Design Issue. Just to give an idea of such ranges, Fig. 6 shows the execution time (in μseconds) of a single modular multiplication with an operand length of 1024 bits for several hardware and software designs.[3] The hardware designs are labeled according to Table 1 -- for example, the label #2_64 denotes a modular multiplier constructed using a number of 64-bit slices (see fourth column in Table 1), each of which based on Design #2 (see second row in Table 1).[4] The software alternatives comprise the Assembler (ASM) and C routines reported in [12].

Given the value in Req5 (stating that a modular multiplication with a 768-bit operand should take no more than 8 μs), the option "Hardware" is selected, i.e., the design space can be immediately pruned so as to encompass only hardware designs.

5.1.5 Specialization of the OMM CDO

Fig. 7 shows the next specialization level for the "Operator - Modular - Multiplier" CDO, created by the generalized "imple-

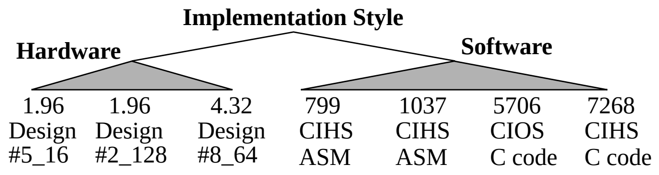


Fig. 6 Execution delay (in μs) of a modular multiplication with 1024 bit operands.

2. All such requirements are *defined* in the layer; during conceptual design, the designer enters their corresponding *values*, based on the specification of the system under design.
3. The delays shown in the figure for the Montgomery designs correspond to the execution delay of the loop (lines 3-4 in Fig. 10). Note that this is the relevant delay in the context of modular exponentiation.
4. Design #2 denotes a modular multiplier implementing the Montgomery algorithm with radix 2, and using Carry-Save adders.

mentation style" Design Issue discussed in the previous section -- accordingly, two new CDOs, "Hardware" and "Software", appear as specializations of that previous (generalized) class. (Ignore for now the second level of specialization also shown in the same figure.) Given that the "Hardware" option was previously selected, we will focus on the characterization of the design space for the sub-hierarchy defined by the "Operator-Modular-Multiplier-**Hardware**" (OMM-H) CDO, shown in gray in Fig. 7, i.e., we consider in some detail the properties defined for this (more specialized) class of design objects.

Six Design Issues are defined for the specialized "Operator-Modular-Multiplier-Hardware" CDO, namely: "Layout Style," "Implementation Technology," "Algorithm," "Number of Slices," "Slice Width," and "Radix," some of which are shown in Fig. 11.

The first two Design Issues, "Layout Style" (DI5 Fig. 11) and "Fabrication Technology" (DI6 Fig. 11), basically define the "meaning" of the generalized "Hardware" option, i.e., discriminate the "real" design options collapsed/lumped into the "hardware" category. The designer can now explore combinations of these two Design Issues, and consider cost-performance and other trade-offs during such an exploration, so as to identify the combination of options that is more convenient for the design at hand. Note that each combination of options selected by the designer (for example, "standard cell" for "layout style," and "0.35 μm" for "implementation technology") *filters* the set of cores indexed under the "Operator-Modular-Multiplier-Hardware" CDO accordingly, thus allowing the designer to consider the performance ranges and other figures of merit, for each such alternatives.[1]

The selection of the number and the width of the multiplier *slices* to be used to build the Modular Multiplier are also important Design Issues -- note that given the characteristically huge *Effective Operand Lengths* (EOL) in encryption applications (see Req1 in Fig. 8), the multiplier ought to be decomposed into a number of slices with widths compatible with the target *clock rate*. The designer can thus filter cores with respect to these two key parameters, in order to explore the design space with respect to sustainable clock rates vs. overall execution delay (in number of clock cycles) for a given EOL.

The "Algorithm" Design Issue (DI2 Fig. 11) states that the modular multiplier can be implemented using two different algorithms: "Montgomery" and "Brickell."[2] Finally, the "Radix" Design Issue, allows for the selection of a radix other than 2 (the default value -- see DI3 in Fig. 11), for any of the algorithms. This last Design Issue thus allows designers to explore the area-

Operator Modular Multiplier	**Req1**	Effective Operand Length (*EOL*) = *768*	
		SetOfValues=$\{2^i \mid i \in Z^+\}$	**Unit**: bits
	Req2	Operand Coding = *2's Complement*	
		SetOfValues={2's compl., Signed ...}	
	Req3	Result Coding = *Redundant*	
		SetOfValues={2's compl., Signed ...}	
	Req4	Modulo is Odd= *Guaranteed*	
		SetOfValues={Guaranteed, notGuaranteed}	
	Req5	LatencySingleOperation ≤ 8	
		SetOfValues=R^+	**Unit**: μsec
	DI1	Implementation Style	**Type**: Generalized
		SetOfValues={Hardware,Software}	

Fig. 8 Requirements and Design Issues for OOM CDO

1. Note that, in practice, the designer may be restricted to a sub-set of such options, but the principle still holds.
2. Since the "Paper and Pencil" algorithm is an inferior solution, it was eliminated.

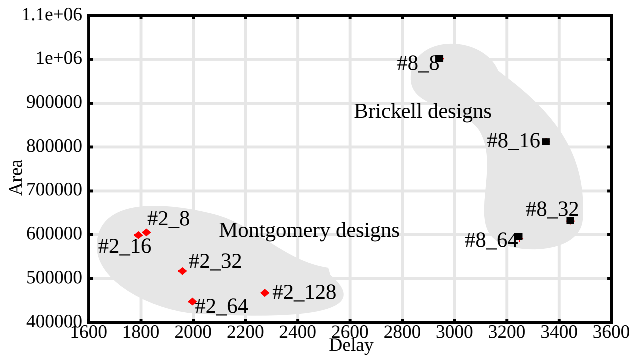


Fig. 9 Evaluation space for Brickell and Montgomery modular multipliers, assuming 768 bit operands.

performance trade-offs that can be implemented via this algorithmic parameter.

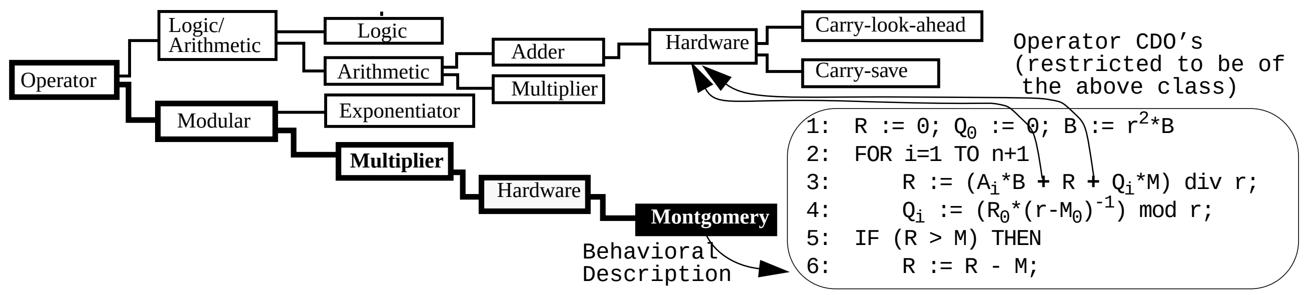
In what follows, we illustrate again the use of generalization, using the "Operator-Modular-Multiplier-Hardware" class of design objects. Consider again the "Algorithm" Design Issue. Fig. 9 shows the evaluation space for a number of reusable designs (available in our reuse library) implementing the Montgomery and the Brickell algorithmic alternatives. All such designs share a common "layout style" ("standard cell"), a common fabrication technology (" 0.35 μm"), and a common radix (2), but have different "slice widths" and thus "number of slices." Note that, in spite of the different performances exhibited by the various designs, resulting from the different "slicing" strategies, the relative *superiority* (in *area* and *performance*) of the Montgomery algorithm with respect to the Brickell algorithm is *consistent*, and is significant. This could suggest that solutions based on the Brickell algorithm are inferior, and thus should not be considered. This is not the case, since the Montgomery algorithm cannot always be used. (Recall that, if the Modulo is not guaranteed to be odd in a given application, the designer has no other choice but use Brickell's algorithm.) Given the above, it should be clear that the selection between both algorithms is *not a "fine-grained," trade-off oriented decision*. This clearly justifies the up-front partitioning of the design space based on these two alternatives, i.e., the use of a *generalized* Design Issue for the "Algorithm" Design Issue.

Accordingly, Fig. 7 shows the *leaf* CDOs, defined by the generalized Design Issue "Algorithm." Since our application can use the "superior" Montgomery algorithm, we will focus on the characterization of the design space for the sub-hierarchy defined by the "Operator - Modular - Multiplier - Hardware-**Montgomery**" (OMM-HM) CDO, shown in black in Fig. 7.

5.1.6 Behavioral Description and Behavioral Decomposition for the OMM-HM CDO

A *Behavioral Description* (at the algorithmic level of design abstraction) is provided for the "Operator - Modular - Multiplier - Hardware - Montgomery" CDO (shown in Fig. 10). The behavioral description also characterizes the *coding type* assumed for the operands, modulo and result (not shown in Fig. 10). This last information is important, since it establishes the possible need for conversions, given the application's requirements (see Req2 and Req3 in Fig. 8).

We now address *design object decomposition*, a key strategy used to control design complexity in large designs. Note that the behavioral description of any complex CDO (at a given level of abstraction) can always be seen as a *behavioral decomposition*.[2] This is so because the behavior of the complex CDO is expressed (in its *behavioral description*) in terms of the behavior of *other, less complex CDOs*. For example, the behavioral description of the modular multiplier shown in Fig. 10 utilizes a



```
1:  R := 0; $Q_0$ := 0; B := $r^2$*B
2:  FOR i=1 TO n+1
3:      R := ($A_i$*B + R + $Q_i$*M) div r;
4:      $Q_i$ := ($R_0$*(r-$M_0$)$^{-1}$) mod r;
5:  IF (R > M) THEN
6:      R := R - M;
```

Fig. 10 Supporting Behavioral Decomposition of Complex Design Objects

number of *arithmetic* operators, such as adders and multipliers. Some of these operators are critical to the performance of the modular multiplier, in particular the additions and multiplications shown in lines 3 and 4 of Fig. 10.[14]

The conceptual design of such *critical operators* is realized by addressing Design Issue DI7 shown in Fig. 11, which requires the selection of the "Behavioral Description" (BD) for each of the operators. (The character "*" represents a "wild card." Note that the expression forces the consideration of "Hardware" realizations for those operators.) This design space exploration step is thus performed using *other CDOs in the hierarchy* (i.e., the "Arithmetic" "Adders" and "Multipliers"), as symbolically shown in Fig. 10.

The "Operator - Modular - Multiplier - Hardware - Montgomery" CDO is a leaf node in the specialization hierarchy, i.e., is the last level of specialization defined in the design space layer. While exploring trade-offs on this leaf CDO, the designer is allowed to revisit the *non-generalized* Design Issues defined for all of its ancestor CDOs, including "Implementation Technology," "Radix," "Number of Slices," etc. Fig. 12, for example, illustrates some of the trade-offs implemented by the Hardware Montgomery Multiplier cores available in our reuse library, with respect to "Radix," "Number of Sections," and also with respect to Adder and Multiplier Implementations. Specifically, "Carry-Look-Ahead" and "Carry-Save" adders are alternatively considered, as well as "Multiplexer-Based" multipliers (implementing multiplications by constants), as alternatives to regular multipliers -- see the corresponding description of each labeled design in Table 1.

OMM-HM	**DI7**	Behavioral Decomposition

		FOR ALL
		Oper:=OPERATORS(BD@*.Hardware)
		SetOfValues={SELECT(BD@Oper),USE(Default)};

OMM-H	**DI2**	Algorithm **Type**: Generalized
		SetOfValues={Montgomery,Brickell}
		Default: Montgomery
	DI3	Radix
		SetOfValues=$\{2^i \mid i \in Z^+, 2^i \le val(EOL)\}$
		Default: 2
	DI4	Number of Slices
		SetOfValues=$\{i \in Z^+ \mid EOL/i = 0\}$
		Default: 1
	DI5	Layout Style
		SetOfValues={Standard-cell, Gate Array, ...}
	DI6	Fabrication Technology
		SetOfValues={0.7μ, 0.35μ, ...}

Fig. 11 Design Issues for OMM-H and OMM-HMCDOs

5.2 Consistency Constraints

In this section we illustrate the different roles played by consistency constraints in the design space layer. CC1 in Fig. 13 express a consistency relationship between properties, i.e., it indicates that the modulo must be odd for the Montgomery Algorithm. Trade-offs and general heuristics may also be represented. CC2, for example, states that the greater the radix, the smaller the latency (in number of cycles).As indicated in Fig. 13, the relation in CC2 is defined for Montgomery multipliers implemented using "Carry-Save" adders. A similar consistency constraint is defined for Montgomery multipliers using "Carry-Look-Ahead" adders, expressing the impact of the unbounded propagation of carries in the multiplier's latency.

CC3 defines the context of utilization of an early estimation tool, denoted *BehaviorDelayEstimator*, used to assign a *rank* to alternative algorithmic-level behavioral descriptions with respect to "MaxCombinationalDelay." Estimation tools are useful when no suitable hard cores are found in the reuse library. For example, the values generated for the (algorithm level) estimation tool in CC3 should be used to compare alternative solutions with respect to combinational delay when no additional design information (at physical, logic and RT levels) is available. Hence, the proposed design space representation defines also the context for which specific *metrics* and *early estimation tools* are to be used. Observe that CC3 is applicable to any "Hardware" CDO.

Finally, constraints may also eliminate inferior solutions. A consistency constraint could, for example, indicate that the guarantee that the modulo is odd is inconsistent with the selection of the (inferior) "Brickell" algorithm. CC4, shown in Fig. 13, eliminates inferior solutions, by stating that when the selected algorithm is the "Montgomery" for EOL 128, only "CarrySaveAdders" should be used for implementing the additions in the loop.The solutions eliminated by CC4 are clearly inferior, due to unbounded propagation of carries (i.e., low performance) and large areas. A similar constraint is also defined to enforce the use of multiplexer-based multipliers for the same loop (in this case for any EOL). This concludes our illustrative example of a design space layer.

As alluded to above, due to space limitations the design space

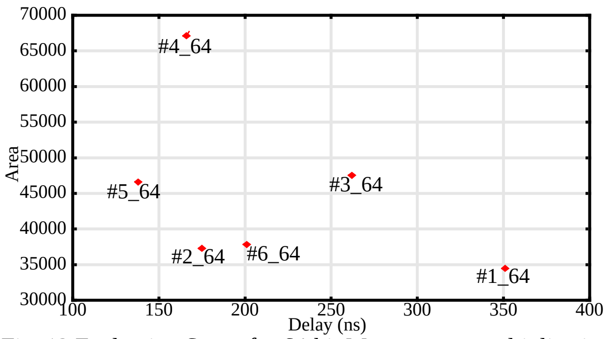


Fig. 12 Evaluation Space for 64-bit Montgomery multiplications using 64-bit slices

Table 1: Operator - Modular - Multiplier - Hardware: Alternative Designs

Design #	Radix	Algorithm[a]	Adder	Multiplier	Slice Width														
					8			16			32			64			128		
					Area	Latency[b]	Clk[b]	Area	Latency[b]	Clk[b]	Area	Latency[b]	Clk[b]	Area	Latency[b]	Clk[b]	Area	Latency[b]	Clk[b]
1	2	M	CLA	N/A	5436	25	2.73	8872	62	3.64	17420	138	4.17	34491	351	5.40	63897	844	6.54
2	2	M	CSA	N/A	6307	27	2.37	12477	45	2.33	21554	92	2.55	37299	175	2.60	77905	388	2.96
3	4	M	CLA	MUL	7433	38	4.21	12265	45	4.93	23987	106	6.18	47533	262	7.91	96106	661	10.16
4	4	M	CSA	MUL	9912	37	3.33	16969	41	3.72	34142	78	4.10	67106	166	4.60	122439	372	5.63
5	4	M	CSA	MUX	9075	38	3.39	14359	38	3.39	24398	67	3.52	46604	138	3.81	85735	295	4.53
6	4	M	CLA	MUX	8013	35	3.84	11939	40	4.43	18983	86	5.07	37829	201	6.08	69751	499	7.67
7	2	B	CLA	N/A	7326	71	3.93	12300	113	4.33	23370	217	5.16	34391	472	6.37	73268	1031	7.47
8	2	B	CSA	N/A	10433	72	3.78	16927	120	4.30	26303	195	4.42	49296	313	4.17			

a. M=Montgomery, B=Brickell

b. Latency and Clk in ns. Latency computed for EOL=Slice width.

0.35µ standard cell library

exploration discussed in this section only addressed modular multiplication, one of the critical components of the modular exponentiation coprocessor of interest [10]. Note, however, that this exploration could have been part of the design space exploration performed for the main architectural component, i.e., the modular exponentiation coprocessor. The exact same behavioral/structural decomposition mechanisms discussed in Section 5.1.6 would have supported the transition between the conceptual design of the main architectural component (i.e., the coprocessor) and the conceptual design of its critical blocks (including the modular multiplier).

Note finally that BUS interface requirements must be specified for each main architectural component of a system-on-a-chip. Such requirements were not part of our discussion since they would have to be specified for the modular exponentiation architectural component (not for its modular multiplication block).

6 Concluding Remarks and Work in Progress

A novel library layer, called the "design space layer," is proposed, aimed at effectively supporting both, *IP-based* design methodologies and traditional "*in-house*" design methodologies, during *early design space exploration*. The systematic exploration of large design spaces is supported by defining a hierarchy of generalized classes of design objects -- such a hierarchy defines increasingly specific families of alternative designs (i.e., design space regions) with respect to the figures of merit of interest. When traversed during conceptual design, this generalization hierarchy supports an effective design space pruning and tradeoff exploration. The design space representation implemented in the layer is self-documented and highly compartmentalized (into hierarchies of CDOs), and is thus easily scalable. Moreover, it can be tailored to the needs and resources of each design environment.

So far we have mostly concentrated on performance vs. area trade-offs. We are currently incorporating power consumption in our case studies, and investigating the need for supporting the co-existence of different specialization hierarchies, so as to effectively guide designers based on the specific trade-offs they may be interested in locally or globally exploring.

CC1	//Montgomery Algorithm requires odd modulo
	Indep_Set={*O*=ModuloIsOdd@OMM}
	Dep_Set={*A*=Algorithm@OMM}
	Relation: InconsistentOptions (*O*=NotGuaranteed & *A*=Montgomery)
CC2	//The > the Radix, the < the latency in #cycles
	Indep_Set={*R*=Radix@*.Hardware.Montgomery, *EOL*@Operator, CSA==oper(+,line:2)@BD@*.Hardware.Montgomery}
	Dep_Set={*L*=LatencySingleOpe@OMM}
	Relation: $L = \frac{2 \times EOL\text{@Operator}}{R} + 1$
CC3	//Behavioral Decomposition impacts delay
	Indep_Set={*B*=BehavioralDecomposition@*.Hardware}
	Dep_Set={MaxCombDelay_R@Operator}
	Relation: MaxComDelay_R=BehaviorDelayEstimator(*B*)
CC4	//Inferior solutions eliminated
	Indep_Set={*EOL*@Operator, *A*=Algorithm@*.Modular.Multiplier.Hardware}
	Dep_Set={BD=BehavioralDescription@OMM-HM}
	Shorts={*Adders*=oper(+,line:2)@BD}
	Relation: InconsistentOptions (*A*=Montgomery & $EOL \geq 32$ &Algorithm@$Adders \neq CSA$)

Fig. 13 Consistency Constraints

7 References

[1] http://www.design-reuse.com

[2] M.Jacome and S.Director, "A Formal Basis for Design Process Planning and Management," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 15, No. 10, Oct1996.

[3] K.Rao and P.Yip, Discrete Cosine Transform, Academic Press, 1990.

[4] D.Gall, "MPEG: A Video Compression Standard for Multimedia Applications," *Comm. ACM* 34, 1994.

[5] http://www.vsi.org

[6] V.Preis *et all*, "Reuse scenario for the VHDL-based hardware design flow," in Proceedings of the *European Design Automation Conference* with *EURO-VHDL*, Sep 1995.

[7] J.Altmeyer, S.Ohnsorge and B.Schuermann, "Reuse of design objects in CAD frameworks," *IEEE/ACM International Conference on Computer-Aided Design*, Nov 1994.

[8] O.Bentz, J.Rabaey, and D.Lidsky "A dynamic design estimation and exploration environment," In proceedings of *34th Design Automation Conference*, ACM Press, June 1997.

[9] R.Rivest, A.Shamir, and L.Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Comm. ACM*, 21, 1978

[10] A.Royo, J.Moron and J.Lopez "Design and Implementation of a Coprocessor for Cryptography Applications," In proceedings of *European Design & Tst Conference*, 1997.

[11] C.Koc, T.Acar, and B.Kaliski, "Analyzing and Comparing Montgomery Multiplication Algorithms," *IEEE Micro*, 16(3):26-33, June 1996.

[12] E.Brickell, "A fast modular multiplication algorithm with

application to two key cryptography," in *Advances in Cryptology - CRYPTO'82*, Chaum *et al.*, Eds. New York: Plenum, 1983.

[13] C.Walter and S.Eldridge, "Hardware Implementation of Montgomery's modular multiplication algorithm," *IEEE Transaction on Computers*, 42(2), 1993.