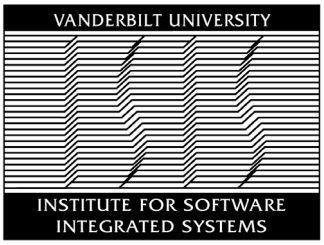
*Institute for Software Integrated Systems*
*Vanderbilt University*
*Nashville Tennessee 37235*




# TECHNICAL REPORT

**TR #:** **ISIS-02-301**

**Title:** **Design-Space Construction and Exploration in Platform-Based Design**

**Authors:** **Sandeep Neema, Janos Sztipanovits, Gabor Karsai**

***Abstract.*** *A fundamental requirement for achieving rapid turn-round and short time-to-market in embedded software and system development is to achieve high level of reuse. Platform-based design offers a systematic way to make tradeoff between the conflicting requirements of flexibility and reuse. This paper describes a model-integrated approach in controlling and exploiting flexibility via the disciplined construction and automated exploration of large design-spaces on hardware/software platforms.*

## 1 Introduction

Embedded software development is inextricably combined with system development. An embedded software component, whose logical behavior is defined in some computer language, is "instantiated into" a physical behavior on a computing device. The instantiation of logical behavior into physical behavior is complicated by the following factors [1]:

1. Physical behavior is directly influenced by the detailed physical characteristics of the devices involved (physical architecture, instruction execution speed, bus bandwidth and others).
2. Modern processor architectures introduce complex interactions between the code and essential physical characteristics of the device (speed, power dissipation, etc.)
3. Lower layers of typical software architectures (RTOS scheduler, memory managers, middleware services) interact with application code in producing the net physical behavior.
4. Properties of physically instantiated software components interfere with each other due to the use of shared resources (processors, buses, physical memory devices, etc.)

Calculation of essential physical properties of designs can be significantly simplified by over design: we use enough resources to minimize or eliminate the need for resource sharing (computation, communication) or to consider hard to compute physical properties unessential (e.g. power). Unfortunately, in most practical cases, efficiency and application circumstances force us to explicitly design for physicality, which requires deep modeling not only the functional structure and behavior of software but also the physical structure and behavior of the embedded system and their interactions.

The cost of modeling on this level of detail is a major concern. The development of deep enough models to compute all interesting physical properties of embedded systems would be cost prohibitive. An emerging concept to mitigate this situation is platform-based design [2]. Platform-based design offers the advantage of large-scale reuse among model development efforts, while preserving controlled flexibility.

The goal of this paper is to describe platform-based design in our model-based design framework called Model-Integrated Computing (MIC). The primary contribution of the described work is a constraint-based model-synthesis technique, which starts with carefully constructed design spaces representing partial designs and synthesizes fully specified models that meet selected design constraints. The paper

first gives a short summary of relevant concepts of MIC and platform-based design. This will be followed by a discussion on constructing, shaping and aggregating design spaces. The next step is the description of our constraint-based model synthesis technique, which is currently based on a symbolic representation of the design space and symbolic pruning of the design alternatives. The paper will conclude with an application example, which shows how to integrate our meta-programmable tools in a domain-specific design environment.

## 2 Platform-Based Design

Since embedded systems are simultaneously computational and physical, it is not surprising that the need for layered abstractions in system design has emerged very strongly in this area. While layered design approaches are known in many engineering field, a clean conceptualization and systematic description of the method in embedded systems is a recent development. The basic tenets of platform-based design (from the point of view of our discussion) are the following [3]:

1. The design proceeds in precisely defined layers of abstraction, such as functional and architectural.
2. Each layer of abstraction is defined by a platform. A platform represents a family of designs (or design space), which satisfy a set of constraints that are imposed on all designs so as to allow the reuse of hardware and software components.
3. A design is obtained by defining platform instances via composing platform components and by mapping one platform to the successive one (e.g. functional to architectural).

In this conceptualization, the following tasks need to be accomplished for establishing an automated platform-based design process:

1. Identification of layers of abstraction, which reflect the characteristics of the domain and the scope of the design process.
2. Systematic construction of the design spaces in each layer of abstraction around some platform concept, which includes the collection of components and the design constraints that must be satisfied.
3. Setting up a synthesis process, which facilitates the mapping among selected platforms.

These tasks represent significant challenge for any realistic application domain. Before describing our solution, we briefly discuss the approach we use for design space representation and manipulation: the meta-modeling technology in Model-Integrated Computing [5].

## 3 Model-Integrated Computing: Meta-modeling

Model-Integrated Computing (MIC) provides a comprehensive methodology and consistent infrastructure for composing domain-specific modeling languages (DSML)

via meta-modeling [5], for automatically generating model-based generators from specifications [6] and for integrating domain-specific design environments [7]. For our later discussion, we briefly summarize our technology for composing domain-specific languages and present a simple example using our meta-programmable modeling tool, the Generic Modeling Environment (GME) [8].

Specification of DSML-s requires the specification of their abstract syntax, concrete syntax, semantic domain and the mapping between the abstract and concrete syntax (syntactic mapping) and the abstract syntax and the semantic domain (semantic mapping) (see e.g. [9]). The formal representations of these specifications are the meta-models and the language we use for describing meta-models is the meta-language. In MIC, the meta-language for representing the abstract syntax of DSML-s and the syntactic mapping is based on UML class diagrams (with stereotypes) and the Object Constraint Language (OCL) [8]. The abstract syntax defines the *concepts, relationships,* and *integrity constraints* available in the DSML. Thus, the abstract syntax determines all the (syntactically) correct "sentences" (domain models) that can be built. (It is important to note, the abstract syntax includes semantic elements as well. The integrity constraints, which define well formed-ness rules for the models, are frequently called "static semantics" [10].) (The formal and manipulable representation of the semantic mapping is harder due to the strongly different formalisms required for representing the abstract syntax and the semantic domain. Therefore, we follow the technique of assigning semantics to a DSML by specifying a mapping between its abstract syntax and the abstract syntax of a language with well-defined semantics. The mapping between two abstract syntax can be defined much easier, e.g. by using graph re-writing [6]. )

Consider a simple example for the abstract syntax of a DSML for Signal Flow (SF) modeling. Figure [1] shows the meta-model of this language (we have omitted the integrity constraints). The core concepts of this language are `Compounds`, `Primitives`, `Ports`, and `Signals`. `Primitives` form the basic signal processing blocks (e.g. Filters, FFT, IFFT, Correlation, etc.). `Ports` define the I/O interfaces of these blocks, and `Signals` represent the signal-flow between the blocks. `Compounds` are processing blocks that can be decomposed into other `Compounds`, and/or `Primitives`. An abstract `Base` concept commonalizes `Compounds` and `Primitives`, and can be said to represent an abstract signal-processing block. With these basic concepts a user can define a signal-processing application. Figure [2] shows a simple hierarchical application model.

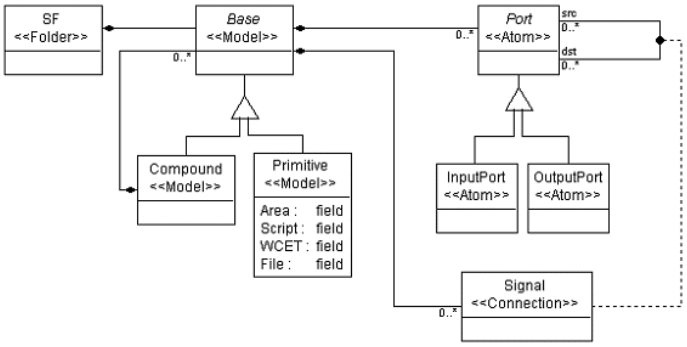



**Fig. 1.** Meta-model for SF

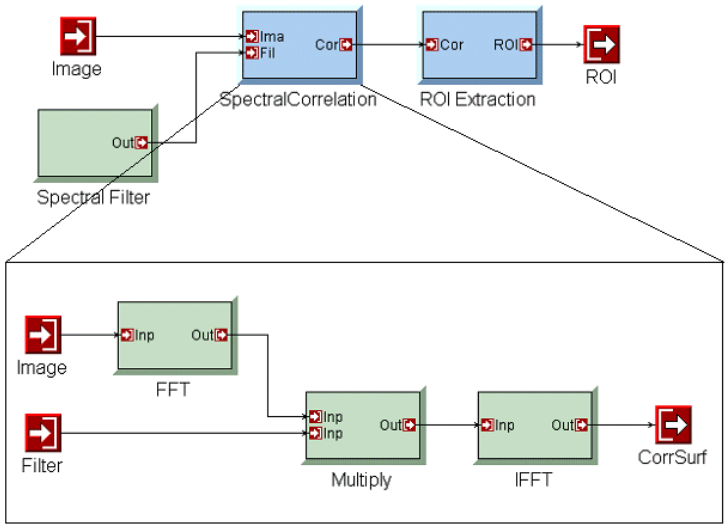



**Fig. 2.** Domain model expressed in SF

The modeling language is sufficient for describing functional models of a broad class of signal processing systems. Each model represents a point design in the overall design space. Operational semantics can be assigned to the modeling language by describing the mapping between the meta-model of SF and the meta-model of a modeling language representing a specific model of computation, such as Synchronous Data Flow (SDF) [12].

## 4. Defining and Shaping Design Spaces

A key requirement for platform-based design is the systematic construction of design spaces. Since we want to develop tools for computer-aided synthesis, we also seek for reusable constructs in DSML-s, which are independent of the actual design language and can be applied over a variety of domains. A second requirement is that the mechanisms should be scalable, in the sense that the effort in constructing the design space should be proportional to the size of the design problem (such as the number of components), and not the size of the design space (typically combinatorial in the number of choices). We use two fundamentally different mechanisms here that satisfy these requirements: 1) Explicit representation of hierarchical alternatives, and 2) Parameterization.

### 4.1 Explicit Representation of Alternatives

Alternative design choices are inherent to an engineering design process. Typically, a design engineer is encountered with several choices for implementing a specification of a system or a sub-system while refining the design. Therefore, to construct a design space using hierarchically layered alternatives is natural to a design process. To enable this, we need to expand DSML-s with the ability to represent design alternatives explicitly.

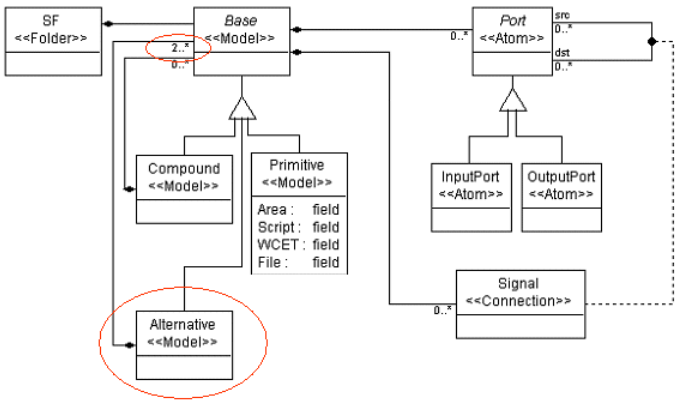



**Fig. 3.** Meta-model of SF extended with the "Alternative" construct

Figure [3] shows the meta-model of SF extended with the concept of `Alternatives`. Note that we use a meta-model composition technique described in [13] to accomplish this extension; figure shows the result of the composition. We selected the abstract `Base` concept for `Alternative` implementation, and introduced a containment relationship to enable hierarchical composition. An

`Alternative`, in the composed SF meta-model context, can now be defined as a processing block with rigorously defined interface, which contains two or more (notice the cardinality of the containment relation highlighted in the figure) alternative implementations. The implementations can be `Compounds`, `Primitives`, or other `Alternatives`, with matching interfaces. Note that matching interfaces is a necessary, but not sufficient condition to ensure composability of the alternative implementations. (We consider the issue of additional conditions for composability later in this paper.)

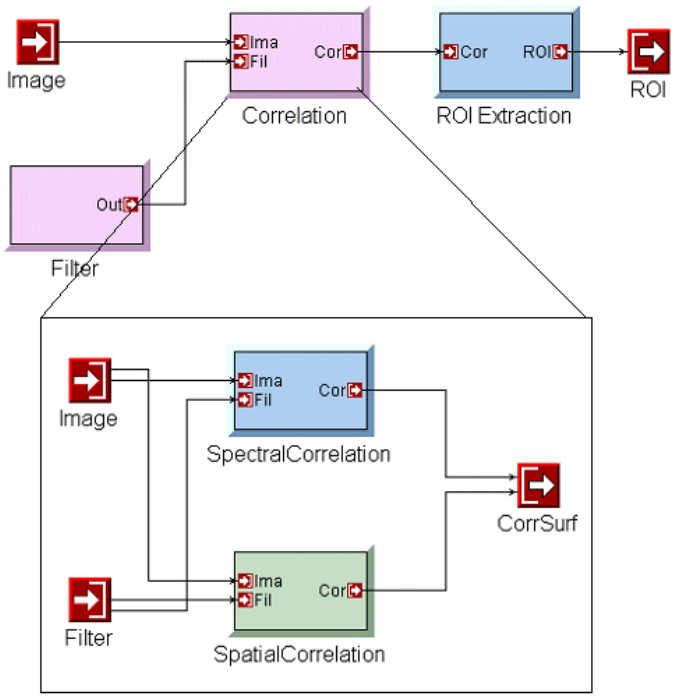



**Fig. 4.** Example for a design space model

With this small extension of the SF design language, a user can now modularly define very large design spaces for a signal-processing application. A domain model, which includes `Alternatives`, now represent a structured design space and not point designs. A small example can be seen in Figure [4], where alternative implementations (Spectral and Spatial Correlation choices for Correlation operator) are utilized to construct a design space for the application.

The scalability of this mechanism in capturing large design space can be judged from the following expressions: With *a* alternative implementations per `Alternative` block, and *n* `Alternative` blocks per `Compound`, composed in

an $m$-level deep hierarchy this representation can define: $a^{k_m}$ design configurations, where $k_m = (k_{m-1}+1)\times n$, and $k_1 = n$, using just $(a \times n)^m$ `Primitives`. As an example, with $n = 4$, $a = 3$, and $m = 3$, a total of 1728 `Primitives` can represent $3^{84}$ design configurations!

We should emphasize here that this construct for representing design spaces is independent of the design language. Extending meta-models with the `Alternative` construct is very simple and can be done according to the nature of the domain and abstraction layer via meta-model composition technique. For example, we can easily define a modeling language for capturing the hardware architecture of a platform, and we can introduce flexibility in the architecture by defining alternative solutions along component hierarchies.

### 4.2 Parameterization

A second approach to define design spaces is based on parameterization, a technique popularized in digital circuit design. A parametric design encapsulates a range of implementations, each of which can be synthesized by binding the parameters appropriately. A simple example to illustrate the point is an N-bit multiplier design in VHDL, using *generics*. The range of the generic parameter N defines the design-space in this case. With multiple parameters the design space is defined by the cross product of the domains of the parameters.

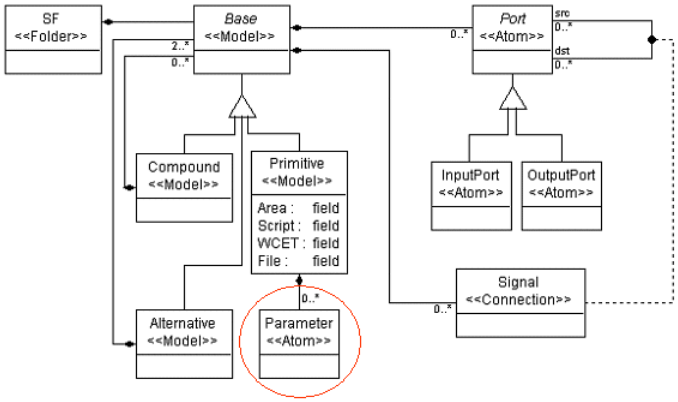



**Fig. 5.** Meta-model of SF extended with the "Parameter" construct

The design spaces defined by this approach can be finer-grained and potentially infinite, compared to that of the explicitly represented alternatives. Figure [5] shows meta-model of the SF language shown earlier extended with `Parameters`, using meta-model composition technique as before. With this extension a `Primitive` can

be parameterized, by containing one or more `Parameters`. It must be noted here that parameterization as a design space definition mechanism, can also be independent of the design language. However, it relies on the implementation language and the support tools (e.g. VHDL compiler) for synthesizing implementations from parametric representations.

## 5. Shaping and Aggregating Design Spaces

In the previous section we identified two basic constructs that can be used to extend DSML-s to represent design spaces instead of point designs. We can use these extensions in composed, multiple-aspect DMSL-s, where individual aspects represent different layers of abstractions in embedded systems and yield a (not necessarily orthogonal) decomposition of the design [13]. However, we face the following problems with the unrestricted composition of large design spaces:

1. While defining design spaces for individual aspects (such as SF), there are many constraints restricting the arbitrary selection of design alternatives.
2. If we construct multiple-aspect design spaces by composing meta-models of individual aspects (e.g. functional and architectural), the resulting aggregate space is the cross product of the design spaces of the individual aspects. However, the product space will represent unrestricted compositions neglecting the fact that aspects may not be orthogonal: design decisions in one sub-space are intricately coupled with design decisions in other sub-space.

There is a need for a mechanism that can help shaping both the individual aspect and the aggregate spaces by restricting them with relations and dependencies. To remain consistent with the selected meta-language, we use OCL-based constraints to "shape" the design space. While the meaning of these constraints is domain-specific, there are typical constraint categories that are suitable to demonstrate the method.

1. *Composability constraints* – We noted earlier that matching interface is not a sufficient condition for composability. In fact, in many situations it may not be possible to compose any arbitrary alternative implementation of a subsystem with any arbitrary implementation of other subsystems. This could be due to the lack of semantic compatibility. A simple example for a semantic composability constraint for a design space defined in SF is shown in Figure [6]. The meaning of the constraint is that Spectral domain correlation composes only with Spectral domain filters and Spatial domain correlation composes only with Spatial domain filters. Semantic composability constraints can express more complex concepts. For example, let us assume that we define a DSML for hierarchical finite state machines with multiple concurrency models [12] – called HFSM. The hierarchically composed components in this language are associated with different models of computations (or concurrency models) [12]. As Lee shows in [14], models of computations that capture the dynamic aspects of component interactions can be considered system-level types. System-level types can be organized in a partial order, which defines composability constraints among the components. By extending the hypothetical HFSM with the construct of

`Alternative` components for defining design spaces, we can restrict the space by imposing Lee's composability constraints.

2. *Inter-aspect constraints* – Inter-aspect constraints express interdependencies across design spaces defined for different layers of abstraction. For example, in platform-based design where we have functional (or application) space on the one hand, and architectural (hardware platform) space on the other hand, a large number of inter-dependencies exist between the functional and architectural components. Composing an end-to-end system requires evaluating crosscutting constraints and making trade-off decisions in the application as well as the architectural space. For example, precision requirements (floating-point vs. fixed-point) in the application may drive the selection of architectural components from one side, while power limitations may drive the selection of architectural components, which in turn drives the selection of application components. Inter-aspect constraints can be used to explicate these dependencies and relations as a constraint network, which can then be subsequently utilized in the design space exploration to systematically synthesize a point-design for the aggregate system.

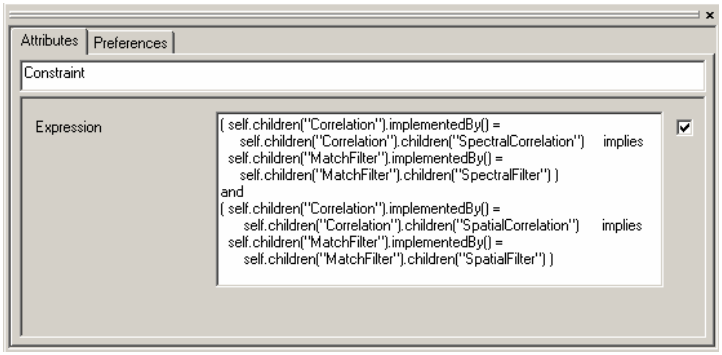



**Fig. 6.** Example for semantic composability constraints

## 6. Design Space Exploration

Up to this point, we identified constructs and methods for defining, aggregating and shaping design spaces. There are two important goals for this exercise:

1. Understand whether or not we have created inconsistency during the design space composition (meaning that the design space is 'empty'), and
2. Synthesizing designs that meet performance constraints.

Given the size of the design spaces we routinely need to deal with, scalable representation, manipulation and exploration of design spaces are very hard problems. In this section we describe a meta-programmable tool, called DEsign Space ExploRation Tool (DESERT) that addresses some of the problems effectively.

### 6.1 Design-Space Abstraction

Design-space representation needs to focus on those aspects of extended DSML-s, which deal with variability in the design. Design-space abstraction means the extraction of this information from domain models defined in extended DSML-s. The abstracted design -space is also represented with a meta-model. Figure [7] shows the meta-model of the design space abstraction within DESERT. `Elements`, `Properties`, `NaturalDomains`, `ElementRelations`, and `Constraints` are the core concepts of this abstraction. An `Element` is an abstraction of a design component (e.g. a `Compound` in SF design language). `Elements` can contain other elements, and the containment relation is characterized by the decomposition attribute that can be either AND-decomposition, or OR-decomposition. AND-decomposition implies inclusive containment (maps to a `Compound` in SF domain), whereas OR-decomposition implies exclusive containment (maps to an `Alternative` in SF domain).

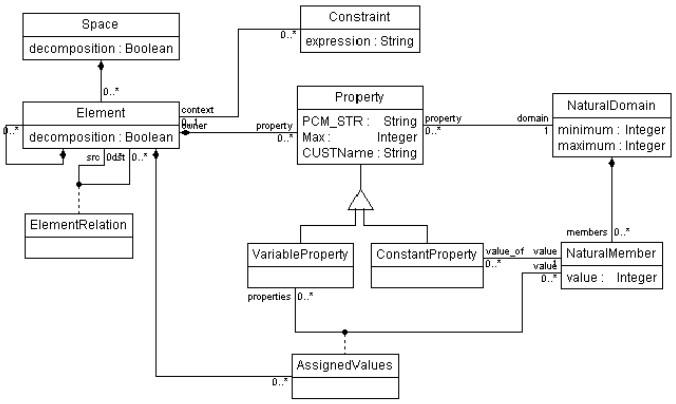



**Fig. 7.** Meta-model for design-space abstraction in DESERT

`Elements` contain `Properties` that capture performance attributes of design components. `Properties` can assume values from `NaturalDomains` that is a bounded range of natural numbers. `Properties` can be `ConstantProperty` to model single-valued performance attributes, or `VariableProperty` to model multi-valued performance attributes. Multi-valued performance attributes commonly occur in platform-based designs, where performance attributes such as timing, power, etc. of functional components depend on the architectural components to which these are assigned.

Introduction of the abstracted design-space has major significance. The design-space manipulation and exploration algorithms of DESERT operate on the abstract design-space, which is isolated from the accidental characteristics of various extended DSML-s. Meta-programmability of DESERT means that after defining the relationship between the meta-model of an extended DSML, and the meta-model of

the abstract design-space, a translator can be generated automatically, which extracts the relevant information from the domain models and builds the abstract design-space. (Currently, the translator generation is only partially automated. A parallel research efforts targets the fully automated generation of model translators [7].)

### 6.2 Symbolic Representation of the Design-Space

The manipulation and exploitation of design-spaces can be reduced to set operations: calculating the product space (composition of design spaces), union and intersection and finding subspaces that satisfy various constraints. Since the size of design-spaces is frequently huge, execution of these operations with enumeration of all elements is hopeless. Therefore, we choose to perform the manipulation and exploitation operations symbolically. Two problems had to be solved: 1) Symbolic representation of design-spaces, and 2) Symbolic representation of constraints.

By introducing a binary encoding of the elements in a finite set, all operations involving the set and its subsets can be represented as Boolean functions [17]. These can then be symbolically manipulated with Ordered Binary Decision Diagrams (OBDD-s) [17], a powerful tool for representing, and performing operations involving Boolean function. The choice of encoding scheme has a strong impact on the scalability of the symbolic manipulation algorithms, as it determines the number of binary variables required for representing the sets. The details of our encoding have been described in [16].

Figure [8] shows the encoding of a set of hierarchically structured alternatives, shown as a tree (branches emerging from a horizontal line below a node denote an AND-decomposition). In this example, S has three alternative implementations: S1 or S2 or S3. S1 also has three alternatives: S11 or S12 or S13. S2's implementation requires three components, S21 and S22 and S23. Out of these components, S21 and S23 have two alternative implementations, S211 or S212, and S231 or S232, respectively. The prefix-based encoding scheme assigns encoding values to each element such that each configuration receives a unique encoding value. A full configuration is defined to be a well-formed path in the tree (e.g. [S {S2 [{S21 S212} S22 {S23 S231}] in the figure). Figure [9] shows the symbolic Boolean representation of this set of hierarchically structured alternatives, given the encoding ($v_i$-s are Boolean variables).

In addition to encoding the structure of design-space, the encoding scheme has to encode the properties of elements also. This requires discretizing the domains of the property variables. The domain size heavily influences the total number of binary variables required to encode the design-space [16]. Subsequent to encoding, and deciding the variable ordering, the symbolic Boolean representation is mapped to an OBDD representation in a straightforward manner [16].

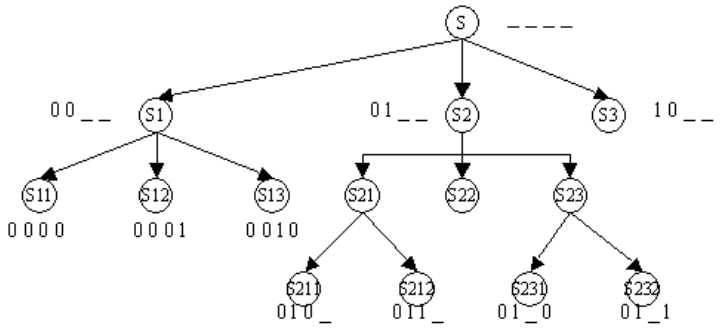



**Fig. 8.** Encoding abstracted design-spaces

$S=S_1 \vee S_2 \vee S_3$
$S_1=S_{11} \vee S_{12} \vee S_{13}$
$S_2=S_{21} \wedge S_{22} \wedge S_{23}$
$S_{21}=S_{211} \vee S_{212}$
$S_{23}=S_{231} \vee S_{232}$

$S_{11}=\neg v_0 \neg v_1 \neg v_2 \neg v_3$
$S_{12}=\neg v_0 \neg v_1 \neg v_2 v_3$
$S_{13}=\neg v_0 \neg v_1 v_2 \neg v_3$
$S_{22}=\neg v_0 v_1$
$S_3=v_0 \neg v_1$

$S_{211}=\neg v_0 v_1 \neg v_2$
$S_{212}=\neg v_0 v_1 v_2$
$S_{231}=\neg v_0 v_1 \neg v_3$
$S_{232}=\neg v_0 v_1 v_3$

**Fig. 9.** Symbolic Boolean representation of abstracted design-spaces

### 6.3 Symbolic Representation of Constraints

Earlier we listed some basic categories of constraints. Symbolic representation of each of these categories of constraints is summarized below.

1. *Composability and Inter-aspect constraints* – These constraints specify relations between elements of the space. Symbolically, the constraints can be represented as a Boolean expression over the Boolean representation of the design-space. Figure [10] shows an example of a composability constraint, and its symbolic Boolean representation.
2. *Performance constraints* – Performance constraints specify bounds on the performance attributes of an aggregate or composed system. These may be in the form of size, weight, energy, latency, throughput, frequency, jitter, noise, response-time, real-time deadlines, etc. Following are some simple examples for the SF language:
   - Timing – end-to-end latency constraints, specified over a signal-flow system, or subsystem e.g. (latency < 20).
   - Power – expresses bound over the maximum power consumption of a system or a subsystem e.g. (power < 100).

Given that various design alternatives may have different values for different performance attributes, performance constraints indirectly imply composability of design alternatives. This makes performance constraints more challenging to represent symbolically, than composability or inter-aspect constraints. Different performance attributes compose differently, e.g. cost can be composed additively, reliability can be composed multiplicatively, latency can be composed as additively for pipelined components, and as maximum for parallel components, etc. The `PCM_STR` attribute of the `Properties` in the meta-model of the abstract design-space (Figure [7]) specifies the composition function to use. DESERT provides some built-in composition functions (addition, maximum, minimum, etc.), and has a well-defined interface for creating custom composition functions. The containment relation between elements is generally not sufficient for composing properties. The `ElementRelations` concept abstracts these other relations such as dataflow, execution order, and others.

In addition to these basic categories of constraints, complex constraints may be expressed by combining one or more of these constraints with first order logic connectives. The symbolic representation of the complex constraints can be accomplished simply by composing the symbolic representation of the basic constraints.

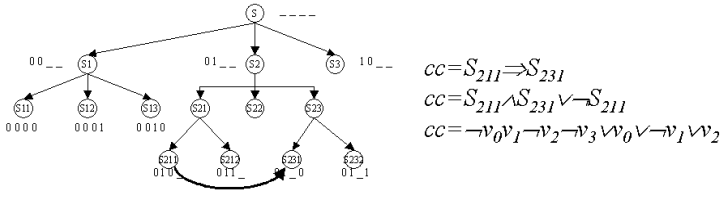



**Fig. 10.** Symbolic representation of a composability constraint

### 6.4 Symbolic Pruning of the Design-Space

The symbolic pruning of the design-space, as observed earlier, in essence is a set manipulation problem. The aggregate design-space is the cross product of design-spaces, each of which is a finite set of designs. Constraints specify relations within the aggregate space. Constraint-based pruning is a restriction of the aggregate space with the constraints. Symbolic pruning is simply the logical conjunction of the symbolic representation of the space with the symbolic representation of the constraints. It is worth reemphasizing that during the pruning process all of the (potentially very large) design spaces are evaluated simultaneously. Figure [11] illustrates the process of symbolic design-space pruning.

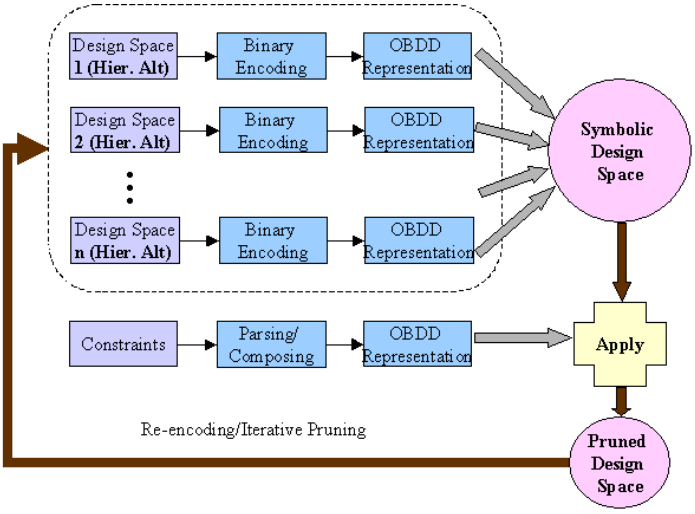



**Fig. 11.** Symbolic design space pruning

## 7. Example: Automated Model Compiler (AMC)

One of the challenge problems presented at EMSOFT 2001 by Butts et al from Ford Research [15] was an automated model compiler for composing controller models for automotive applications. In this section we briefly summarize their problem definition, and show our solution based on DESERT.

### 7.1 Usage scenarios

AMC is an automated model composition tool, which takes component models, target architectures, and synthesizes models automatically, which meet design requirements. The key elements of the usage scenarios for our discussion are the following:

1. Model components are Matlab®, Simulink®, Stateflow® models.
2. Model components have a set of key component attributes (I/O definitions, essential parameters, etc.) that influence composability and capture performance characteristics.
3. Target model architectures are described by an abstracted, hierarchical, high-level modeling language, whose leaf nodes refer to model components defined above.
4. There is a rich set of compatibility relations for components. Structural constraints focus on I/O signal types and simulation properties. Component compatibility relates components via high-level design goals, such as “fun-to-drive” or “green”.

The challenge is to synthesize models that meet set design goals and performance targets using the available model components. To characterize problem sizes, authors

refer to a powertrain control example, where 218 model components, each with 3-30 alternatives, are used in 130 vehicle applications. A typical vehicle application includes 75-105 components.

### 7.2 AMC Architecture

AMC is a rich enough very interesting application problem, therefore we use a preliminary implementation of AMC for demonstrating an important category of DESERT applications. Since in the defined use scenario AMC needs to work together with other modeling tools, the presented solution needs to deal with broader tool integration issues.

The architecture of AMC is shown in Figure [12]. Components of the architecture are the following:

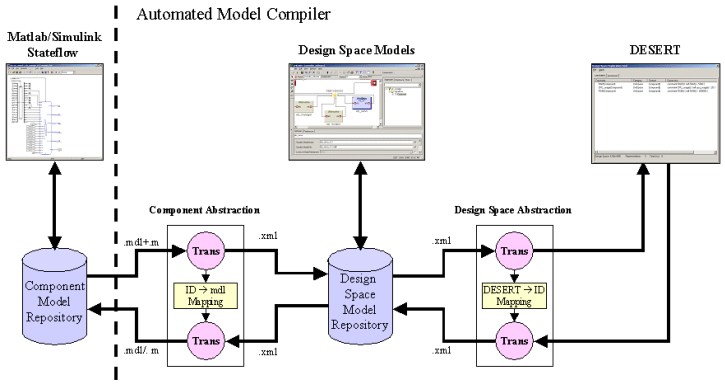



**Fig. 12.** Architecture of an Automated Model Compiler

1. *Matlab/Simulink and Component Repository*: The repository contains simulation model for various automotive subsystems. The models are stored in .mdl (Model Definition Language) files, and there is an associated parameter definition file, that defines a number of performance parameter (e.g. CPU usage, RAM/ROM usage) and characterization information (green vs. fun-to-drive vehicle, etc.) for the Simulink model. (The experimental system does not include StateFlow models.)
2. *Component Abstraction*: AMC does not need all of the detailed information in the components. Component abstraction means that components are modeled, and the component models include only the relevant information for model synthesis. For example, the detailed I/O interface specifications of Matlab/Simulink models are abstracted into basic I/O types. Parameters which are not essential for model composition are suppressed. The component abstraction separates the Matlab/Simulink world and the AMC world. The actual abstraction is supported by a two-way model translator (generator). The content of the Component Repository

is translated into partial component models and sent to the GME-based Design-Space Modeling environment in .xml format. The unique association between the components in the Repository and their models in GME are maintained. In the other direction, abstracted models, synthesized by AMC are translated into fully specified Matlab/Simulink models by using the unique model component id-s and auto-generating all of the connections based on the detailed I/O specifications.

3. *Design-Space Modeling Environment*: The challenge problem definition [15] suggested the introduction of a high-level modeling language for the specification of target architectures. This language uses the abstracted components at its leaf nodes so as to allow modelers focusing on the appropriate level of abstraction. Therefore, we defined an SF-like design language with hierarchy and alternatives and instantiated it in our meta-programmable Graphical Model Editor (GME). Design space models capture the hierarchical composition of vehicle systems and capture design alternatives for subsystems. A primitive (leaf node) in this language represents a simulation model in the Matlab/Simulink Component Repository, and is linked to the simulation model through attributes that store model name, version number, and file name. There are additional placeholder attributes for performance, and characterization parameters that need not be filled by the user. The translator in the Component Abstraction tool parses the parameter and I/O definitions and populates the models with this information. The user can model design specifications (e.g. CPU_usage < 70%, RAM < 20Kbytes) as constraints in the design space models. Consistency constraints (e.g. connected I/O should have matching data types) are automatically introduced in the models. The Design-Space Modeling Environment supports the specification of structural, and component compatibility constraints in OCL.
4. *Design-Space Abstraction*: As we described it in Section 6, DESERT uses a domain-independent meta-model, which separates its internal algorithms from domain-specific constructs. The Design-Space Abstraction component of the architecture provides two-way model translation between the Design-Space Models and the DESERT's abstract design-space models. The two-way translation enables that acceptable point designs selected from the pruned design space by DESERT can be presented in the Design-Space Modeling Environment and can be translated further automatically for the Matlab/Simulink environment for detailed simulation studies.

We have a working prototype with the feed-forward path from the Matlab/Simulink component repository to design-space models and DESERT, fully implemented. The prototype has been demonstrated and tested. Design-spaces were subjected primarily to I/O compatibility and performance constraints (CPU usage, RAM/ROM requirements). The largest design spaces constructed during "stress tests" included 130 binary variables.

## 8. Conclusions and Future Work

In this paper we presented an approach and a related tool for constructing and exploring large design-spaces as part of a platform-based design process. Symbolic

pruning of large design-spaces seems to be a useful tool component in the overall tool chain. Our preliminary results in the example described above and in other examples (e.g. reported in [16]) have shown that the selected binary representation method for design spaces scales well. The critical challenge in scalability is during the design-space pruning phase. Application of complex constraints to large spaces may result in explosion of the OBDD-s, therefore DESERT has an interactive user interface. Users can control the importance of constraints and select the sequence order of their application. We are experimenting with re-encoding the design-space after each pruning steps, which usually results in a drastic decrease in the number of binary variables.

Since OBDD-s are not effective as SAT solvers, we separate design-space pruning from finding a single architectural alternative, which satisfies complex performance constraints. To facilitate this step, we are in the process of developing an interface in DESERT toward high performance SAT solvers.

## References


[1] J. Sztipanovits and G. Karsai: "Embedded Software: Challenges and Opportunities," EMSOFT 2001, LNCS 2211, Springer. (2001) 403-415

[2] Ferrari, A., Sangiovanni-Vincentelli, A.: "System Design: Traditional Concepts and New Paradigms,"

[3] Sangiovanni-Vincentelli, A.: "Defining Platform-based Design," EEDesign

[4] J. Sztipanovits and G. Karsai: "Model-Integrated Computing," *IEEE Computer,* April, 1997 (1997) 110-112

[5] Nordstrom G., Sztipanovits J., Karsai G., Ledeczi, A.: "Metamodeling - Rapid Design and Evolution of Domain-Specific Modeling Environments", Proceedings of the IEEE ECBS'99, Nashville, TN, April, 1999. (1999) 68-75

[6] Levendovszky, T., Karsai G.: "Model reuse with metamodel based-transformations," ICSR, LNCS, Austin, TX, April 18, 2002.

[7] Karsai, G., Gray,J.: "Design Tool Integration: An Exercise in Semantic Interoperability," Proceedings of the Engineering of Computer Based Systems (ECBS) Conference, Edinburgh, UK, March, 2000. (2000) 272-278

[8] Generic Modeling Environment documents, http://www.isis.vanderbilt.edu/projects/gme/Doc.html

[9] T. Clark, A. Evans, S. Kent, P. Sammut: "The MMF Approach to Engineering Object-Oriented Design Languages," Workshop on Language Descriptions, Tools and Applications (LDTA2001), April, 2001

[10] UML Semantics, Ver. 1.1., Rational Software Corporation, 1997.

[11] F. Puntigam, "Types for Active Objects Based on Trace Semantics," Proc. of the Workshop on Formal Methods for Open Object-Oriented Distributed Systems (FMOODS'96), Paris, France, March, 1996.

[12] E. A. Lee and A. Sangiovanni-Vincentelli: "A Framework for Comparing Models of Computations," IEEE Trans. CAD Integrated Circuits and Systems, (1998) 1217-1229

[13] Ledeczi A., Nordstrom G., Karsai G., Volgyesi P., Maroti M.: "On Metamodel Composition," IEEE CCA 2001, CD-Rom, Mexico City, Mexico, September 5, 2001.

[14] Lee, E.A., Xiong, Y.: "System-Level Types for Component-Based Design," EMSOFT 2001, LNCS 2211, Springer. (2001) 237-253

[15] Butts, K., Bostic, D., Chutinan, A., Cook, J., Milam, B., Wand, Y.: "Usage Scenarios for an Automated Model Compiler," EMSOFT 2001, LNCS 2211, Springer. (2001) 66-79

[16] Neema, S., "Design Space Representation and Management for Embedded Systems Synthesis," Technical Report, ISIS-01-203, February 2001. http://www.isis.vanderbilt.edu/publications/archive/Neema_S_2_0_2003_Design_Spa.pdf
[17] Bryant R., "Symbolic Manipulation with Ordered Binary Decision Diagrams," School of Computer Science, Carnegie Mellon University, Technical Report CMU-CS-92-160, July 1992.