Proceedings of EMSOFT 2003, LNCS 2855, pp. 290-305

Constraint-Based Design-Space Exploration and Model Synthesis

Sandeep Neema, Janos Sztipanovits and Gabor Karsai

Institute for Software Integrated Systems, Vanderbilt University, P.O. Box 1829 Sta. B.
Nashville, TN 37235, USA
{sandeep.neema,janos.sztipanovits,gabor.karsai@vanderbilt.edu}

Abstract. An important bottleneck in model-based design of embedded systems is the cost of constructing models. This cost can be significantly decreased by increasing the reuse of existing model components in the design process. This paper describes a tool suite, which has been developed for component-based model synthesis. The DESERT tool suite can be interfaced to existing modeling and analysis environments and can be inserted in various, domain specific design flows. The modeling component of DESERT supports the modeling of design spaces and the automated search for designs that meet structural requirements. DESERT has been introduced in automotive applications and proved to be useful in increasing design productivity.

1 Introduction

Extensive modeling and model analysis is a crucial element of design flows for embedded systems. The cost of the design process is largely influenced by the cost of constructing models for detailed analysis and code generation. There is a significant pressure toward tool manufacturers to decrease this cost by increasing the automation of the modeling process. A promising approach to achieve this goal is to take advantage of reusable modeling components and automated model composition. An exposition of this problem was detailed by Ken Butts and others [8] describing the need for a model compiler in automotive applications.

Automotive controller modeling is typically done by means of model-based tool suites, such as MathWorks Simulink® and Stateflow® [2]. According to the authors [1], building models for powertrain controller assemblies requires selection of 75-100 component models from thousands for each of the 130 vehicle types. The model components include 1 to 3 sampling contexts (crankshaft position or time) and on the average 20 inputs and 14 outputs. This means that the complete model has over 100 scheduling connections and 2000 data connections. Components cannot be selected independently from each other. There are complex compatibility relationships among model components. Selection of components may require selection of others and the overall component structure influences or determines basic performance characteristics of the design.

The goal of our research has been the design of a component-based model synthesis tool, which given a set of models for subcomponents, composes a system model *m* automatically such that a set of design constraints are satisfied. The design requirements for the tools have been inspired by the practical needs of automotive control engineers:

1. Model components are Matlab®, Simulink®, Stateflow® models.
2. Model components are characterized by a set of component attributes (component type like "continuous" and "discrete", I/O definitions, essential parameters, solver used in simulation, sampling time, etc.) that influence composability and capture performance characteristics.
3. Model architectures are described by an abstract, hierarchical, high-level modeling language, whose leaf nodes refer to model components defined above.
4. There is a rich set of compatibility relations among model 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 integrated models that meet set design goals and performance targets using the available model components. While the original model compiler challenge [8] is defined in the context of Simulink® models, we generalized the method and tools developed and made them independent from the actual domain specific modeling languages and modeling tools used in a particular design flow.

The primary contribution of the described work is the DEsign Space ExploRation Tool (DESERT), which starts with carefully constructed design spaces representing design templates and synthesizes fully specified models that meet selected design constraints. DESERT is a domain independent tool chain for defining design spaces and executing constraint-based design space exploration. The DESERT tool chain can be linked to different domain specific modeling environment via model transformations. These domain specific modeling environments may include hardware architecture or software architecture models.

The paper first gives a short summary of relevant concepts. This will be followed by a discussion on constructing, shaping and aggregating design spaces. The paper concludes with 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.

2 Background and Terminology

In model-based design, systems are described by models expressed in domain specific modeling languages (DSML). Formally, a DSML is a five-tuple of concrete syntax (*C*), abstract syntax *(A)*, semantic domain *(S)* and semantic and syntactic mappings (M_S, and M_C) [9]:

$$L = < C, A, S, M_S, M_C >$$

The *C concrete syntax* defines the specific (textual or graphical) notation used to express models, which may be graphical, textual or mixed. The *A abstract syntax* defines the *concepts, relationships,* and *integrity constraints* available in the language. Thus, the abstract syntax determines all the (syntactically) correct "sentences" (in our case: models) that can be built. (It is important to note that the abstract syntax includes semantic elements as well. The integrity constraints, which define well-formedness rules for the models, are frequently called “static semantics”.) The *S semantic domain* is usually defined by means of some mathematical formalism in terms of which the meaning of the models is explained. The M_C : A®C mapping assigns syntactic constructs (graphical, textual or both) to the elements of the abstract syntax. The M_S: A®S semantic mapping relates syntactic concepts to those of the semantic domain.

Any DSML, which is to be used in the development process of embedded systems, requires the precise specification (or modeling) of all five components of the language definition. The languages, which are used for defining components of DSMLs are called *meta-languages* and the concrete, formal specifications of DSMLs are called *metamodels* [1].

The specification of the abstract syntax of DSMLs requires a meta-language that can express concepts, relationships, and integrity constraints. In our work in Model-Integrated Computing (MIC) [4], we adopted UML class diagrams and the Object Constraint Language (OCL) as meta-language. This selection is consistent with UML’s four layer meta-modeling architecture [16], which uses UML class diagrams and OCL as meta-language for the abstract syntax specification of UML.

The semantic domain and semantic mapping defines semantics for a DSML. The role of semantics is to give a precise interpretation for the meaning of models that we can create using the modeling language. Naturally, models might have different interesting properties; therefore a DSML might have a multitude of semantic domains and semantic mappings associated with it. For example, *structural* and *behavioral* semantics are frequently associated with DSMLs. The *structural semantics* of a modeling language describes the meaning of the models in terms of the structure of model instances: all of the possible sets of components and their relationships, which are consistent with the well-formedness rules in defined by the abstract syntax (structural semantics is frequently called *instance semantics* [16]). Accordingly, the semantic domain for structural semantics is defined by some form of set-relational mathematics. The *behavioral semantics* describes the evolution of the state of the modeled artifact along some time model. Hence, behavioral semantics is formally modeled by mathematical structures representing some form of dynamics, such as discrete transition system [2] or Hybrid Systems [3].

Although specification of semantics is commonly done informally using English text (see e.g. the specification of UML 1.3 [16]), the desirable solution is explicit, formal specification. There are two frequently used methods for specifying semantics: the *metamodeling approach* and the *translational approach* (see Figure 1).

- In the *meta-modeling approach* (see e.g. [5]), the semantics is defined by a meta-language that already has a well-defined semantics. For example, the UML/OCL meta-language that we use for defining the abstract syntax of a DSML has a structural semantics: it describes the possible components and structure of valid, syntactically correct domain models. The semantics of this meta-language can be represented by using a formal language, which supports the precise definition of sets and relations on sets. By providing the formal semantics for UML class diagrams and OCL - say, in Z [6]- the UML/OCL meta-model of the abstracts syntax of a DSMLs specifies not only its abstract syntax, but its structural semantics as well.

- The *translational* approach defines semantics via specifying the mapping between a DSML and another modeling language will well-defined semantics. As it can be seen on the right side of Figure 1, a model translator is a function $T: A_1 ?\ \ A_2$ whose domain and co-domain are the abstract syntax specifications. By choosing UML class diagrams and OCL as meta-language for the specification of abstract syntax, the specification of the model translator can be facilitated using graph transformations between (instance) graphs generatively specified by the class diagram [7].

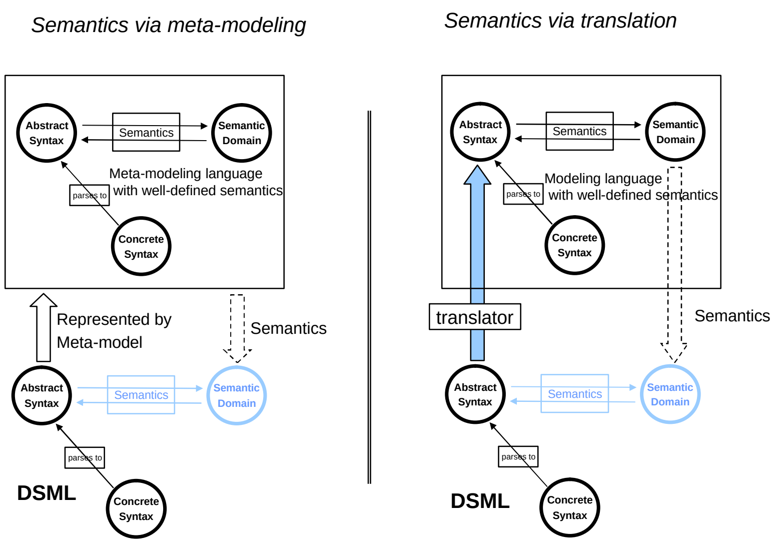


Figure 1: Specification of semantics

1

As it will be shown later, formally specified structural semantics, meta-modeling and model transformations play essential role in solving the model synthesis problem in a domain independent manner. Although in this paper we focus on meta-models representing the abstract syntax of DSMLs (because of their significance in defining structural semantics and because of the central role of abstract syntax in specifying model translations), we should mention that meta-models are also used for defining behavioral semantics. For example, Lee and Sangiovanni-Vincentelli [19] developed the Tagged Signal Model to compare models of computations. Burch, Passerone and Sangiovanni-Vincentelly introduced trace algebras as semantic domain for modeling behaviors in the Metropolis project [20]. These, and other meta-modeling approaches have important roles in defining different behavioral semantics for DSMLs.

3 Construction of the Design Space

Model structures for automotive control are specified as refinements of a generic architecture [8] shown in Figure 2. Each top-level model component, such as transmission, transmission control T/M, etc., has many alternative realizations with different parameters and internal structure - arranged in a refinement hierarchy. Accordingly, components at any level in the refinement hierarchy may have alternative implementations. We call components with alternative implementations *templates.* At the leaf nodes of the refinement hierarchy are the *primitive* model components, which may also be templates with alternative (primitive) implementations. In our specific application context, the primitive model components are Simulink® models. (The “Bus” concepts on the diagram represent a set of variables shared among the model components.)

The *template* concept has a significant impact on the *structural semantics* of models. Without templates, each m model is unique, i.e. represents a point design. By introducing *templates* in the specification of the modeling language, each model including templates defines a set of models, M_D, which we call *design space*. An m_j ***Î*** M_D model instance in the design space is defined by binding the component templates to one of the alternative implementations and by the binding the parameter values of the parameterized components to a specific value.

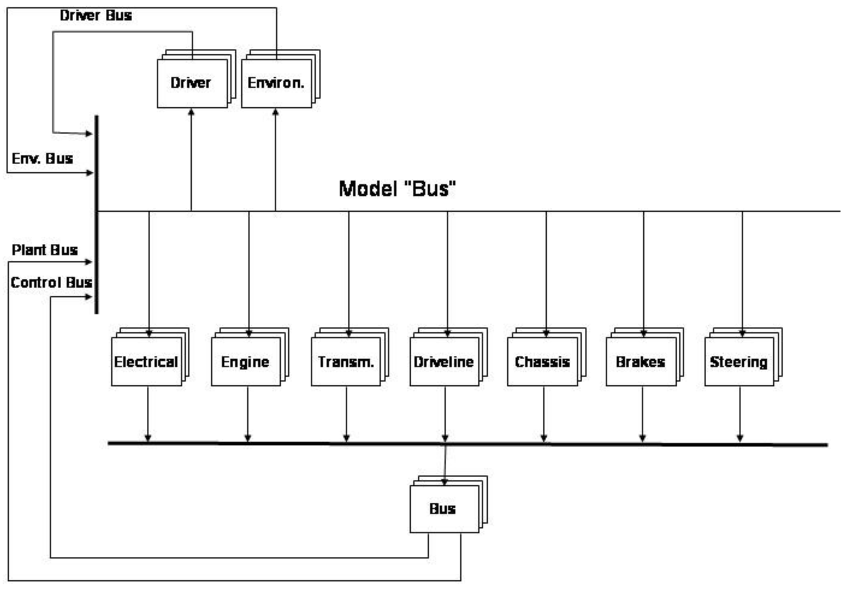


Fig. 2. Top-level Model Architecture in Automotive Systems

Remark 1: The scalability of hierarchically structured alternatives in capturing large design space can be judged from the following example: With *a* alternative implementations per Template, and n Template blocks on each level of an m-level deep refinement 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!

Remark 2: The design space captures all possible *m_j Î M_D* model instances defined by the instantiation of templates and model parameters. Therefore, the essential semantics of a DSML, which is extended with the template and parameter concepts is *structural semantics.*

The goal of model synthesis is to find model instances in the design space, which satisfy design constraints regarding the feasible composition of model components (see Compatibility Constraints in section 5.3) and satisfy selected structural characteristics. *Structural characteristics* can be directly derived from structural features of the model. For example, let *C* be the set of all primitive control components and let *C_j={c|c Î C,c Î m_j}* is the subset of primitive controller components, which are elements of model *m_j Î M_D*. Furthermore, let *cost(c_i)* is a function, which assigns and estimated implementation cost for each primitive controller components. Since the overall cost of the controller implementation for *m_j* is (approximately) the sum of the implementation cost of all controller components: *cost(C_j)=å$_i$ cost(c_i|c_i Î C,c_i Î m_j)*, we can further restrict the design space with a cost constraint *L*:

$M_{D,<L}$={m_j Î M_D|cost(C_j)<L}

where *$M_{D,<L}$* is the set of all models in the design space, whose controller cost is less then *L*. The significance of *structural constraints* is that constraint-based design space exploration tools can efficiently restrict the design space using computationally inexpensive techniques.

Remark 3: As opposed structural constraints, *behavioral constraints*, such as controller dynamic performance, require behavioral analysis of designs using simulation or other analysis tools. Since simulation for large models is expensive, it is a good strategy to restrict the design space first by means of *structural constraints* and explore only the structurally correct designs using simulation.

It is interesting to relate our concept of *design space* with the concept of *platform* in Sangiovanni-Vincentelli's platform-based design [21]. Applying the definition of platforms in [22] we can conclude that the set of all designs that can arise from composing the *C* set of primitive model components form a *platform* (we will refer to this platform as *Model Platform, MP*). Clearly, the set of all designs, which can be generated by combining the primitive model components (i.e. the Model Platform) is not practical for automated model synthesis. The size of this set is unbounded and includes practically only meaningless configurations, therefore any automated search in that space would be hopeless. The top-down refinement strategy followed in our design space construction strategy starts with an application (or a set of applications, such as automotive controllers) and incrementally refines it to *MP*. Using the terminology of platform-based design [21][22], we start by selecting a point or a set of points in an *Application Platform* (formed by the interesting set of automotive controllers) and capture the feasible refinement paths to *MP* via hierarchically layered alternative refinements and parametrization. The resulting set of designs M_D *Ì* *MP* restricts tremendously the number of meaningless configurations. The remaining meaningless configurations in the top-down refinement strategy are the result of compatibility violations and neglected interdependences among design decisions. However, we can express these formally by means of compatibility and interdependency *constraints*, which than can be used for further restricting the design space.

Creation of extensive, well structured design spaces for different category of applications requires significant effort. However, this effort is not required in the form of an initial investment, which is to be done before design space exploration can start. We envision design space construction as byproduct of product development process; the creation of a structured repository where design experience is accumulated.

4 Overview of the DESERT design flow

Our goal with DESERT has been to provide a model synthesis tool suite, which can represent large design spaces and can manipulate them by means of structural constraints. The place of these steps in the overall design flow is shown in Figure 3. The input to the DESERT tool suite are models and model components used for behavioral modeling and analysis (in our case Simulink® models). DESERT does not require all details of Simulink models, only those, which are required to specify the basic structure of designs.

1. The *Component Abstraction* tool maps Simulink® model components into Abstract Components by executing the T_A model transformation. The Abstract Components preserve the structure of I/O interfaces (using interface types) and configurable parameters.
2. The *Design Space Modeling* tool supports the construction of M_D design spaces using Abstract Components. Templates and parameters are the added language constructs that enable modelers define design spaces instead of point designs.
3. The *Design Space Encoding* tool maps the M_D design space into a representation form, which is suitable for manipulating/changing the space by restricting it with various design constraints.
4. The *Design Space Pruning* tool performs the design space manipulation and enables the user to select a design, which meets all structural design constraints.
5. The *Design Decoding* tool reconstructs the selected design from its encoded version.
6. The *Component Reconstruction* tool takes the design with abstracted components and reconstructs the detailed design suitable for behavioral analysis (i.e. it synthesizes the detailed Simulink model in our example).

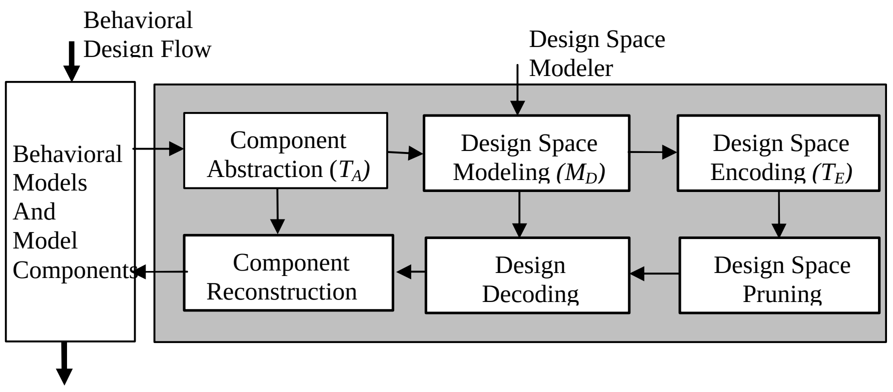

Behavioral Design Flow
Design Space Modeler
Behavioral Models And Model Components
Component Abstraction (T_A)
Design Space Modeling (M_D)
Design Space Encoding (T_E)
Component Reconstruction
Design Decoding
Design Space Pruning

Fig. 3. DESERT Design flow

In the following sections we describe the key principles we used in implementing the system.

5 Specification of DSMLs in the DESERT Design Flow

As we discussed earlier, the domain and codomain of model transformations in the design flow are represented through meta-models specifying the abstract syntax of the modeling languages. The DESERT design flow includes several transformations on the models. These transformations decouple the complex design space exploration tools (Design Space Encoding and Design Space Pruning) from the domain (and tool) specific detailed Simulink component models and from the Design Space Modeling tool, which still uses a modeling language with domain specific flavor (although uses abstracted components). In this section we show some of the important components of the design flow focusing more on the approach rather then the technical details.

5.1 Component Abstraction

The T_A model transformation receives Simulink® model components and generates abstract components for Design Space Modeling. The domain of T_A is represented by the meta-model of the Simulink models, which is shown in Figure 4. (Detailed description of the Simulink® meta-model can be found in [10].) In short, the *System* class encapsulates a system model in Simulink. It serves as a container for the block diagram that models a dynamical system. It contains *Block* objects and connections (*Line* objects) between the Block objects. Some of these Block objects may be *Subsystem* objects. A Subsystem can be composed as a block diagram. The composition is indirect through a System object i.e. every Subsystem object contains a single System object. This enables hierarchical representation of a complex system.

Every Model/Subsystem object contains a single System object. A System object may also contain *Annotation* objects that are used to add documentary information/user comments to the system models. The System class has a small set of attributes that capture various visual preferences, printing preferences, and system information.

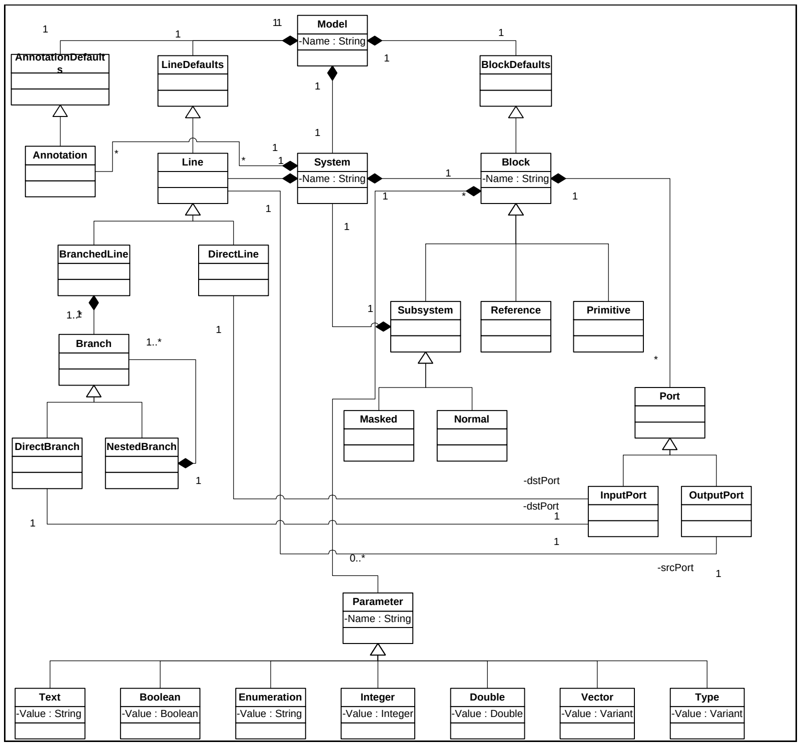


Fig.4. Meta-model of Simulink®

The codomain of the T_A model transformation is represented by the meta-model of the Design Space Modeling tool. The meta-model is shown in Figure 5.

Similarly to the Simulink meta-model, the Design Space meta-model specifies also a hierarchical block diagram language with object types *Compound*, *Template* and *SimulinkSystem.* The *SimulinkSystem* objects refer to the abstracted components imported from the Simulink environment. The Simulink and Design Space meta-models in Figures 4 and 5 clearly show role of the T_A (Component Abstraction) transformation: internal details of the Simulink model objects are suppressed, only the name (*Name* attribute in the *System* object) I/O interfaces (*InputPort* and *OutputPort*) and parameters (*Parameter* object) are preserved in the Design Space Modeling environment. The specific new constructs added to the meta-model for modeling Design Spaces are the *Template*-s (see our earlier discussion on templates) and *Constraint*-s. In DESERT's Design Space Modeling environment the constraints are described as OCL expressions [13].

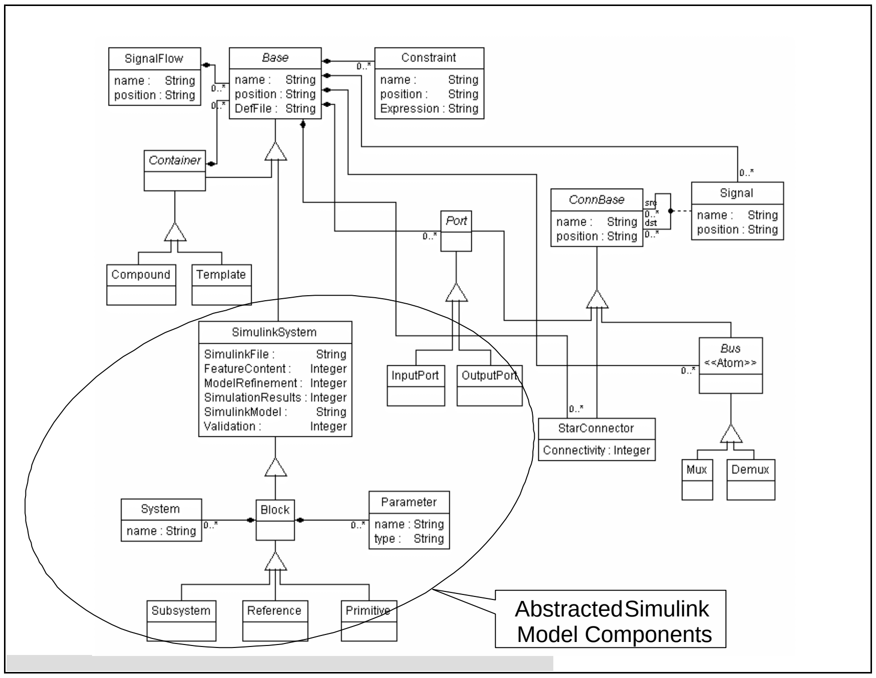


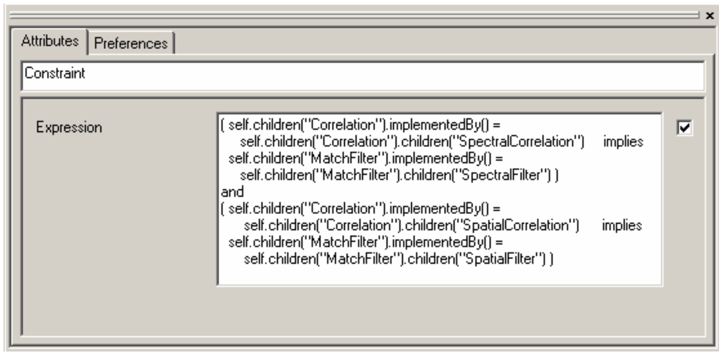
Fig.5. Design Space meta-model

5.2 Design Space Modeling

As we described earlier, the M_D design space is constructed manually by Design Space Modelers (see Figure 3). The DESERT tool suit uses Vanderbilt/ISIS Graphical Modeling Environment (GME) as Design Space Modeling tool. GME is a metaprogrammable graphical model builder [11], which can be customized to a DSML via abstract syntax and concrete syntax specification in terms of meta-models. The latest public release of GME3 is downloadable from the ISIS web site [12].

Design space modeling is essentially creating product line model architectures: modelers incrementally build, expand the M_D space by adding new primitive model components (abstracted from Simulink models), and compose them into new versions of plant and controller models. Note that this process is *not* the enumeration of all possible designs; the designer merely specifies alternatives on various levels of the hierarchy. While adding new implementation alternatives to plant and controller models on various levels of the refinement hierarchy, the rapidly expanding design space can be restricted by new compatibility and other structural constraints. To remain consistent with the selected meta-modeling language (UML class diagrams and OCL), we use OCL-based constraints [13] 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. *Compatibility constraints* – Matching interfaces are not sufficient conditions for composability. In fact, in many situations selection of implementation alternatives are not orthogonal due to the lack of semantic compatibility. A simple example for a semantic compatibility constraint for a design space defined in a signal processing domain 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”.



Constraint

Expression ☑

```
( self.children("Correlation").implementedBy() =
    self.children("Correlation").children("SpectralCorrelation")     implies
  self.children("MatchFilter").implementedBy() =
    self.children("MatchFilter").children("SpectralFilter") )
and
( self.children("Correlation").implementedBy() =
     self.children("Correlation").children("SpatialCorrelation")     implies
  self.children("MatchFilter").implementedBy() =
     self.children("MatchFilter").children("SpatialFilter") )
```

Fig.6. Example for semantic composability constraints

2. *Inter-aspect constraints* – Inter-aspect constraints express interdependencies among design spaces defined for capturing different aspects of designs. For example, plant models and controller models can be represented in two separate design spaces. Naturally, a large number of inter-dependencies exist between the plant model configurations and controller configurations. Composing an end-to-end system requires evaluating these inter-aspect constraints. 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.

5.3 Design Space Encoding and Pruning

Up to this point, we identified constructs and methods for defining, aggregating and constraining design spaces. The roles of the next two steps in the DESERT design flow (see Figure 3) Design Space Encoding and Design Space Pruning are the followings:

1. understand whether or not we have created inconsistency during design space modeling (meaning that the design space is 'empty'), and
2. find models that meet the required structural constraints.

Since we are focusing on structural semantics of the design space and intend to compute with structural constraints, manipulation of design-spaces can be reduced to set operations: calculating product spaces (composition of design spaces) and finding subspaces that satisfy various (structural) constraints. Since the size of design-spaces is frequently huge, execution of these set manipulation operations with enumerating all elements is hopeless. Therefore, we choose to perform the manipulation operations *symbolically*. Two problems had to be solved: 1) symbolic representation of design-spaces, and 2) symbolic representation of constraints.

If we restrict the parameters of model objects to finite domains, the design space will be also finite. 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 [15]. These can then be symbolically manipulated with Ordered Binary Decision Diagrams (OBDD-s), 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 scheme has been described in [16], here we only demonstrate the approach.

Figure (7) shows the encoding of a simple design space formed by hierarchically structured alternatives. In this example, *s*, the top-level node in the design space is a template, which has three alternative implementations: *s1 or s2 or s3*. *s1* is also a template with three alternative implementations: *s11 or s12 or s13*. *s2*'s implementation requires three components, *s21 and s22 and s23*. Out of these components, *s21* and *s23* are templates with two-two alternative implementations, *s211 or s212* and *s231 or s232*, respectively. The prefix-based binary encoding scheme [14] assigns binary code to each element such that each configuration receives a unique encoding value. In the example, four Boolean variables, $[\nu_0, \nu_1, \nu_2, \nu_3]$ are required for the encoding of the structure. Figure (8) shows the symbolic Boolean representation of this design space, given the encoding. For example, the binary code of node *s2* in the design space is $S2 = \neg\nu_0\nu_1$. As it can be seen, *s2* represents a subspace in the design space with four alternative configurations.

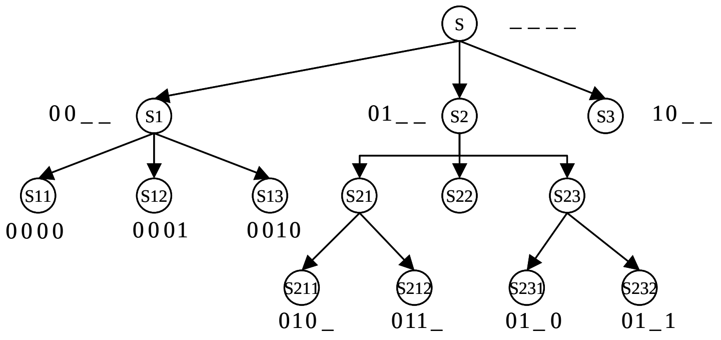


Fig. 7. Encoding abstracted design-spaces

$S=S_1ÚS_2ÚS_3$	$S_{11}=Øv_0Øv_1Øv_2Øv_3$	$S_{211}=Øv_0v_1Øv_2$
$S_1=S_{11}ÚS_{12}ÚS_{13}$	$S_{12}=Øv_0Øv_1Øv_2v_3$	$S_{212}=Øv_0v_1v_2$
$S_2=S_{21}ÙS_{22}ÙS_{23}$	$S_{13}=Øv_0Øv_1v_2Øv_3$	$S_{231}=Øv_0v_1Øv_3$
$S_{21}=S_{211}ÚS_{212}$	$S_{22}=Øv_0v_1$	$S_{232}=Øv_0v_1v_3$
$S_{23}=S_{231}ÚS_{232}$	$S_3=v_0Ø\,v_1$	

Fig. 8. Symbolic Boolean representation of abstracted design-spaces

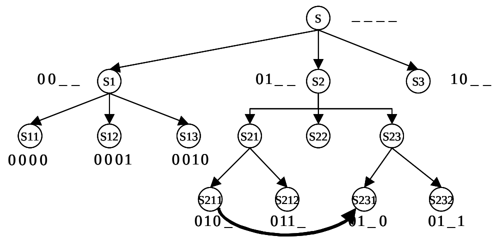
In addition to encoding the structure of the design-space, the encoding scheme has to encode the parameters of the parameterized model components. Subsequent to encoding, and deciding the variable ordering, the symbolic Boolean representation is mapped to an OBDD representation in a straightforward manner [14].

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

1. *Compatibility and Inter-aspect constraints* – These constraints specify relations among subspaces in the overall design space. Symbolically, the constraints can be represented as a Boolean expression over the Boolean representation of the design-space. Figure (9) shows an example of a compatibility constraint, and its symbolic Boolean representation. The compatibility constraint in the example expresses interdependency between implementation decisions. Specifically, selection of implementation *s211* for template *s21* implies that *s231* must be the selected implementation for template *s23*. In the Boolean space, this constraint is expressed by the following Boolean expression:

   $$cc= S211\ Þ\ \ S231=S211Ù\ S231Ú\ \neg S211$$
2. *Resource constraints* – Resource constraints specify bounds on the resource needs of a composed system. These may be in the form of size, weight, energy, cost, etc. As we described earlier, the important limitations for the resource constraints, which DESERT is able to manage is that they are

derived from structural characteristics of designs. In general, resource constraints are more challenging to represent symbolically, than composability or inter-aspect constraints. Different resource 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. DESERT provides some built-in composition functions (addition, maximum, minimum, etc.), and has a well-defined interface for creating custom composition functions.



$cc=S_{211}\ \text{Þ}\ S_{231}$

$cc=S_{211}\text{Ù}S_{231}\text{ÚØ}S_{211}$

$cc=\text{Ø}v_0v_1\text{Ø}v_2\text{Ø}v_3\text{Ú}v_0\text{ÚØ}v_1\text{Ú}v_2$

Fig. 9. Symbolic representation of a compatibility constraint

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.

The symbolic pruning of the design-space, as observed earlier, in essence, is a set manipulation problem. 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 emphasizing that during the pruning process all of the (potentially very large) design spaces are evaluated simultaneously. Figure 9 illustrates the process of symbolic design-space pruning. As the figure shows, the Design Space Pruning tool is prepared for applying constraints on the Design Space. According to our experience, OBDD based representations scale well for representing the structure of the design space (nested AND/OR expressions). The critical challenge in scalability occurs during the design-space pruning step. Automatic application of complex constraints to large spaces may result in explosion of the OBDD-s therefore DESERT has an interactive user interface to influence this process. 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 (see Figure 10).

The primary advantage of the symbolic design space pruning approach is that it is exhaustive: the pruned space includes all of the designs, which meet the applied design constraints. As experience with scalability in real-life industrial applications at Ford Research accumulate, we will be able to see the limitations of the method.

A significantly simpler, but still useful alternative approach to design space pruning is to find a single design configuration (not all), which satisfies the selected design constraints. We currently experiment with various constraint solvers and languages, such as Oz [17] to develop solution for this approach.

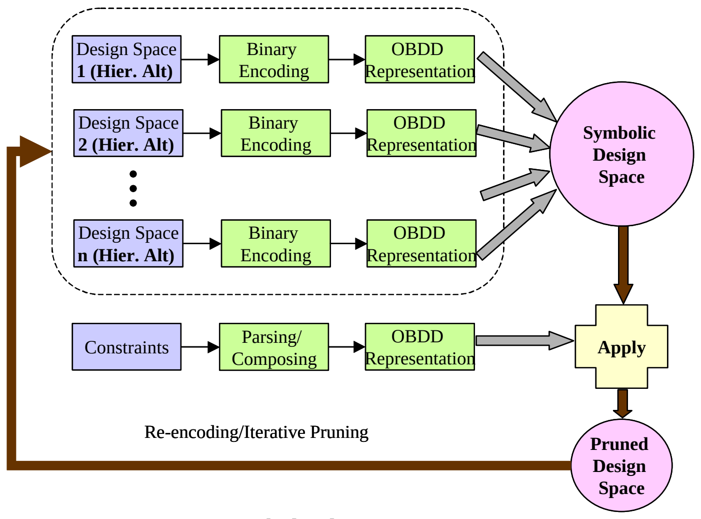


Fig. 10. Symbolic design space pruning

5.4 Reconstructing Detailed Design

The concluding steps of the DESERT design flow (see Figure 3) are the decoding of the selected abstract design from its binary code and reconstructing the detailed models from the abstract design. The decoding process involves reconstructing a data structure from the encoded form, where the data structure is an abstracted representation of the actual design. The design space encoder and decoder components operate on a shared “dictionary”, which captures what specific design space model elements correspond to what codes. The resulting data structure is a specific, point design (i.e. a single model *m*), which can be looked at in the modeling tool. Naturally, it does not have any templates (i.e. no alternatives).

The final step is the creation of the actual design model in Simulink. The design model created in the design decoding step is follows the Design Space metamodel (from Figure 5), and from it a model is constructed that follows the Simulink metamodel (from Figure 4). This construction is mostly straightforward, the only complication arises because of the connections: the blocks in the final design have to be “wired-up” as required. However, this wiring can be algorithmically generated from the (higher-level) buses and wires captured in the design space models.

6. Conclusions and Future Work

Design space modeling and model synthesis are important part of the design flow for embedded systems. Automated tools can offer major productivity advantages for practicing engineers. During the development of the DESERT tool suite the Vanderbilt/ISIS team has been in close interaction with Ford Research, which was essential in identifying the crucial challenges engineers face. At this point, the prototype tool is field tested in real-life vehicle development programs.

The method described has been based on the structural semantics of models. The role of model synthesis based on structural constraints is to prune the design space before the computationally more expensive behavioral analysis methods are used for finding optimal design solutions. Since structural semantics is defined for all domain specific modeling languages, the tool suite can be interfaced to many design flows (independently from the behavioral semantics of the languages). We have been using this opportunity extensively in other domains, such as modeling environment for power-aware computing [18].

References

[1] Karsai, G., Sztipanovits, J., Ledeczi, A., Bapty, T.: "Model-Integrated Development of Embedded Software", *Proceedings of the IEEE*, Vol. 91, No.1., pp.145-164, January, 2003

[2] D. Harel and M. Politi, Modeling Reactive Systems with Statecharts: The STATEMATE Approach, McGraw-Hill, 1998

[3] Thomas A. Henzinger. The theory of hybrid automata. Proceedings of the 11th Annual Symposium on Logic in Computer Science (LICS), IEEE Computer Society Press, 1996, pp. 278-292

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

[5] Clark, T., Evans, A., Kent, S.: "Engineering Modeling Languages: A Precise Metamodeling Approach," R.-D. Kutsche and H. Weber (Eds.): FASE 2000, LNCS 2306, pp. 159-173, 2002

[6] Evans, A., France, R., Lano, K., Rumpe, B. : « Developing UML as a Formal Modeling Notation,"LNCS 1357, pp. 145-150, Springer Verlag Berlin, 1997

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

[8] 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

[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] Neema, S.: "Simulink and Stateflow Data Model", see www.isis.vanderbilt.edu.

[11] 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

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

[13]Object Constraint Language Specification, ver. 1.1, Rational Software Corporation, et al., Sept. 1997. (1997)

[14] 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

[15] Bryant R., "Symbolic Manipulation with Ordered Binary Decision Diagrams," School of Computer Science, Carnegie Mellon University, Technical Report CMU-CS-92-160, July 1992

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

[17] http://www.mozart-oz.org/

[18] Ledeczi A., Davis J., Neema S., Eames B., Nordstrom G., Prasanna V., Raghavendra, C., Bakshi A., Mohanty S., Mathur V., Singh M.: Overview of the Model-based Integrated Simulation Framework, Tech. Report, ISIS-01-201, January 30, 2001.

[19] Lee, E.A., Sangiobanni-Vincentelli, A.L.: "A framework for comparing models of computations," *IEEE Transactions on Computer Aided Design Integrated Circuits*, 17(12):1217-1229, Dec. 1998.

[20] Burch, J.R., Passerone, R., Sangiovanni-Vincentelli, A.L.: "Modeling Techniques in Design-by-Refinement Methodologies, Proc. of IDPT-2002, June, 2002

[21] Sangiovanni-Vincentelli, A.L.: " Defining platform-based design," *EEDesign*, February, 2002

[22] Lee, E.A., Neuendorffer, S., Wirthlin, M.J.: "Actor-Oriented Design of Hardware and Software Systems," (to be published in *Journal of Circuits, Systems and Computers*) *Technical Memorandum UCB/ERL M02/13*, University of California, Berkeley, CA 94720, May 1, 2002