A FRAMEWORK FOR CONSTRAINT-BASED MODELING

Gregory Provan
Computer Science Department, University College Cork, Cork, Ireland
{*g.provan@cs.ucc.ie*}

Abstract

We propose a framework that provides a clear process and well-developed semantics for modeling complex systems using a constraint-based language. We describe the aspects of a model that are captured by a constraint-based representation, in terms of function-based and process-based modeling approaches. We propose a formal framework for building constraint-based models, and apply graph grammars to provide verification and analysis tools during the model construction process.
KEYWORDS: constraint satisfaction, graph grammars.

1 Introduction

Modeling using a Constraint-Based (CB) framework is a task of increasing importance. Applications include job-shop scheduling [6], configuring computers [10], and modeling human organs [1]. However, there is no formal framework that describes an appropriate process for modeling complex systems using CB languages.

This article presents a formal framework for CB modeling and CB model verification/analysis. One main focus of this article is to explicitly examine the aspects of a model captured by a CB representation. There are many papers that define efficiently-computable models for specific examples, but little has been written about the underlying semantics of a CB model or the CB model development process. In contrast, in the area of design and simulation, there is a wealth of information about what aspects of a complex system can be captured by a specific representation. For example, you can capture the system's functionality [15], its structure [16], or modeling process [14].

To address the aspects of a model captured by a CB representation, we adopt a functional semantics, and assume that we can use of state-of-the-art GUI tools for describing the relationships of model components. Our main interest is how to model these functional relationships using a CB language, and how to generate a CB model from a set of inter-related sub-system (or component) functional relationships. We also show how we can augment our modeling framework with a formal-methods approach that can be used to guarantee correctness of the models that are generated, throughout the entire modeling process. Note that we show how to represent a complex system using a *generic* CB language, in contrast to domain-specific languages like Bond Graphs [12].

We also focus on the *process* of developing a CB model using a GUI. In contrast to papers such as [3], which examines compositional model refinement, we focus on generating a model using an abstract CB language. In addition, we do not address the process of obtaining efficient model representations, or of issues like model symmetry.

We assume that we will use a (hierarchical) GUI to build a CB model of a physical system, in which we are interested in defining the functional properties of the system. A functional model [17] specifies how a system performs a task in terms of flows (or energy, power, or data), with clearly-defined sources and sinks for the flows. We assume functional models are causal, and capture the flow direction (causality) using directed graphs.[1]

Applying CB formalisms to systems modeling and inference has strengths and weaknesses. The main strengths are its expressive power and large body of inference algorithms. Its main weakness is its lack of appropriate modeling semantics, as compared to languages like Bond Graphs [12] or Petri Nets [19], which are designed specifically for systems modeling. A CB language is general, and can model any type of system. In contrast, model-specific languages have domain-specific primitives for encoding system properties; for example, bond graphs can model the energy and signal flows among system components using a small set of primitives [12]. A second weakness is the difficulty of verifying that the model created using a GUI is identical to the CB model output at the end of modeling: the semantics for the GUI model must be compatible with the semantics of the CB language. We overcome these weaknesses by showing how we can define the entire process of model construction in a formal framework.

The article is organised as follows. Section 2 presents a new formal framework for constraint-based modeling. Section 3 applies graph grammars to verify the CB modeling process. Sections 4 and 5 summarise related work and conclusions, respectively.

2 Functional CB Modeling

Researchers have developed many approaches for systems modeling, ranging from bond graph methods [12], which focus on dynamical systems models for coupled systems,

[1] This assumption translates into having a directed constraint graph for the CB language.

to discrete systems models [7]. We can characterise these modeling approaches as being either *function-based* or *process-based*. A *function-based modeling approach* focuses on the functional aspects of a system, and models the system as a set of functional transformations. Examples include bond graphs [12] and function-converter approaches [15]. A *process-based modeling approach* focuses on the process aspects of a system, defining the system in terms of a set of processes that can be used for model validation, model inference, or limiting the design space that must be searched [17]. Process knowledge may include design rules and/or procedures, process capabilities, or scientific principles. Examples include graph grammars [2, 9] and exemplar methods [16].[2]

Because the CB representation is generic (it can be applied to any domain due to its generality), it rules out a direct integration with domain-specific approaches, such as the bond graph framework, which has a clear domain-specific interpretation and is based on differential equations. Hence, we will adopt a function-based approach and use graph grammars for design-process verification.

In the following, we make the following assumptions: (a) we have a system for which the topology of the system is known, and can be arranged in a hierarchical fashion; (b) we can model our domain using discrete constraints; in particular, we adopt a propositional logic representation for the abstract model representation; (c) we can define the functionality of the components of the system; and (d) we assume a static system, although time can easily be modeled.

We can split the modeling process up into three steps:

1. GUI model development, producing model $\mathcal{M}_{GUI}$;
2. abstract modeling, producing model $\mathcal{M}_A$, and
3. flat CSP modeling, producing model $\mathcal{M}_{CSP}$.

Note that different model information is stored at each modeling step:

GUI model The GUI model captures the hierarchical information in a graphical framework. In particular, it stores block inter-relationships and block GUI-related data, e.g., positioning.

abstract model The abstract model captures both the hierarchical information and the functional descriptions (possibly in a more abstract, and less application-dependent manner).

flat CSP model The CSP model specifies the constraint equations defining the model's functionality, and data for performing CB-inference.

The abstract model can be used to translate to a variety of application-dependent languages, such as a causal network language for diagnostics [7]; it can also be used to identify inference-specific information, such as symmetries in a model's representation, which can speed up constraint-based inference. This model can be defined as a meta-model, due to the role that it plays [18].

[2] See [17] for for a full review of these approaches.

We further assume that we have to (1) transform the GUI model to the abstract model, using transformation $\tau_1 : \mathcal{M}_{GUI} \rightarrow \mathcal{M}_A$, and (2) transform the abstract model to the CB-model, using transformation $\tau_2 : \mathcal{M}_A \rightarrow \mathcal{M}_{CSP}$.

We now describe each model, followed by the model transformations.

2.1 GUI Model

During GUI model development (e.g., using component-based library modeling tools like 2^{nd}-CAD, MATLAB Simulink, or Bayesian network tools like Hugin, Netica or Genie), we assume that a modeler will use a GUI tool to represent a system in terms of a set of connected system components. Hence we need to represent both the system components (as represented by the GUI), their connectivity, and the physical layout of the GUI components.

More formally, the GUI model consists of the tuple $\langle \mathcal{B}, \mathcal{E}(\mathcal{B}), \wp \rangle$, where $\mathcal{B}$ is the block model data together with block interconnections $\mathcal{E}(B)$, and $\wp$ is the GUI presentation data.

Block Model Data: $\mathcal{B}$ consists of the a set of tuples for blocks, each of which is given by $(I, O, f, \mu, \mathcal{E})$, representing inputs I, outputs O, function f, mode μ, and connections $\mathcal{E}$. We denote a function f by $O = f(I, \mu)$. f may be represented as a (flat) set of constraints, or it may be represented as a nested set of sub-blocks and interconnections among them (as we will describe below).

GUI Data: $\wp$ consists of the various types of data needed to display each block, such as shape and colour information, block position, and the wiring information for interconnecting the blocks, e.g., wire type (dashed, solid), colour, routing type (shortest-path, no-overlap, etc.).

The input and output ports I, O will be defined as typed GUI entities. μ_B is the mode of the block B, and represents the set of operating modes that B can be in. We assume that μ_B is a property of B.

We define block interconnections $\mathcal{E}(B)$ in terms of pairs of ports of consistent type. In other words, each $e \in \mathcal{E}(B)$ consists of a pair $e = (o, i \mid o \in O \wedge i \in I \wedge i \cong o)$, where $i \cong o$ means that the type of input port i is consistent with the type of output port o. The types of two ports are deemed to be consistent based on the constraint language being used. The simplest form of consistency is equality, i.e., if the two ports have the same type.

From the GUI's perspective, we can treat the function f in terms of a functional language that is effectively a black box. The GUI is solely responsible for model creation, i.e., facilitating the creation of a model consisting of interconnected blocks, with appropriate functional data present as block attributes.

Example: Throughout this article, we will use discrete electrical circuits for illustrative purposes; the process we describe can be applied to arbitrary constraint languages and domains. Consider a NAND composite block derived from AND and NOT gates, as shown in Figure 1.

We denote our GUI model data, $\langle \mathcal{B}, \mathcal{E}(\mathcal{B}), \wp \rangle$, as shown in Table 1.

Model	I	O	f	μ
NAND	$\{I1, I2\}$	O	f_{NAND}	μ_{AND}
AND	$\{AND.I1, AND.I2\}$	$AND.O$	f_{AND}	μ_{AND}
NOT	$NOT.I$	$NOT.O$	f_{NOT}	μ_{NOT}

Table 1. GUI data for NAND model.

The inter-connectivity data, $\mathcal{E}_{\mathcal{B}}$, are given by {(I1,AND.I1), (I2,AND.I2), (AND.O, NOT.I), (NOT.O, O)}. For clarity of exposition, we omit the block graphics details $\wp$.

2.2 Abstract Model

The abstract model represents the system components, their connectivity, and the functional transformations performed by each component.[3] We represent this information using the tuple $\langle \mathcal{B}, \mathcal{E}(\mathcal{B}), \mathcal{F} \rangle$, where $\mathcal{B}$ is the system components, $\mathcal{E}(\mathcal{B})$ is the set of interconnections, and $\mathcal{F}$ is the set of component functional transformations.[4]

This model may also incorporate the hierarchical information present in the model. We do not assume that this model is in a form for directly performing inference on it, but that it contains the information necessary to transform it into a version on which inference can be performed. In this sense, this model serves as a meta-model for generating models for constraint-based inference.

The abstract model consists of the block tuples $\langle \mathcal{B} \rangle$, where $\mathcal{B}$ is the block model data together with block interconnections $E(B)$, such that the functional transformations are explicitly defined in a formal language.

The primary thing that we need to specify for a block (and its transformation f) is whether it contains any sub-blocks as part of its representation. We define a block (function f) as *primitive* if it contains no sub-blocks (sub-functions).

A block B is given by $(I, O, f, \mu, \mathcal{E})$. For a composite block, the inputs I and outputs O remain the same, but we have to specify the function f and modes μ differently, as described below.

Primitive function: We define a composite function for block B in terms of the sub-functions defined in B for each output port. If we assume that B has k output ports, then for output port i, we have a set of inputs $\mathcal{I}_i$, interconnections E_i, and function f_i, such that, for $i = 1, ..., k$, outputs are denoted $O_i = f_i(\mathcal{I}_i, \mu)$, and connections $E_i = \{(\iota, O_i) \mid \iota \in \mathcal{I}_i\}$.

Composite function: A composite function has an internal set of sub-blocks, so we have to define the function definitions for output ports in a different manner. We define a composite function for block B_i in terms of the sub-functions and their order of composition. This in turn is computed from B_i's sub-blocks, $\mathcal{B}_i$, and their inter-connectivity $\mathcal{E}_{\mathcal{B}_i}$.

Example: Consider the example of the NAND composite block. At the composite level, we can define the model as follows. We have a single composite model, NAND, with inputs (I1,I2), output O, composite function f_{NAND}, mode μ_{NAND}, and inter-connectivity given by $\mathcal{E}_{\mathcal{B}}$= {(I1,O), (I2,O)}.

We can then specify the hierarchical detail in terms of the sub-blocks. The sub-blocks of the NAND block are given by $\mathcal{B}$= {AND, NOT}, and the inter-connectivity by $\mathcal{E}_{\mathcal{B}}$= {(I1,AND.I1), (I2,AND.I2), (AND.O, NOT.I), (NOT.O, O)}.

A composite function f can be represented as a composition of its respective functions. For example, in the case of the NAND model, where the AND and NOT gates have functions f_{AND} and f_{NOT} respectively, the NAND composite function is simply given by $f_{NAND} = f_{NOT} \circ f_{AND}$, where $\circ$ represents function composition. For more complex models, e.g., models in which the sub-models of a block are organised in series-parallel order, the function composition can be defined appropriately.For this example, we have: $AND.O = f_{AND}(\{AND.I1, AND.I2\}, \mu_{AND})$, $NOT.O = f_{NOT}(AND.O, \mu_{NOT})$, which can be rewritten as NOT.O = $f_{NOT}(f_{AND}(\{\text{AND.I1,AND.I2}\}, \mu_{AND}), \mu_{NOT})$.

By abstracting the mode specification and assuming that we have inputs $I1, I2$ and output O, we can then represent function composition for the NAND example as $O = f_{NOT} \circ f_{AND}(\{I1, I2\})$.

2.3 Flat CSP Model

The flat CSP model is a non-hierarchical version of the abstract model in which the constraints are represented explicitly in a standard constraint language, such that a CSP engine can be employed for performing inference on the model. We represent a flat CSP using $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, denoting variables, domains and constraints respectively.

Example: Consider the NAND created from AND and OR gates. We will define a flat CSP with variables $\mathcal{X}$, variable domains $\mathcal{D}$, and constraints $\mathcal{C}$. This model has variables given by $\{I_1, I_2, AND.O, AND, NOT.O, NOT\}$, where mode variables AND, NOT have domains {OK, BAD} and the other variables have domains of $\{t, f\}$. We can define the constraints in a number of ways, such as using allowable tuples or equations like: $[I_1 = t] \wedge [I_2 = t] \wedge [M_{AND} = OK] \wedge [M_{NOT} = OK] \Rightarrow [O = f]$; $\not\{[I_1 = t] \wedge [I_2 = t]\} \wedge [M_{AND} = OK] \wedge [M_{NOT} = OK] \Rightarrow [O = t]$.

2.4 Model Transformations

This section describes the types of transformations necessary to map the models as follows:

[3] Note that we do not have to represent the physical layout of the GUI components in this model.

[4] We assume that the GUI block and the abstract model components are identical; we can generalise this notion such that they are not identical.

- transform the GUI model to the abstract model, using transformation $\tau_1 : \mathcal{M}_{GUI} \rightarrow \mathcal{M}_A$, and
- transform the abstract model to the CB-model, using transformation $\tau_2 : \mathcal{M}_A \rightarrow \mathcal{M}_{CSP}$.

GUI to Abstract-Model Mapping

τ_1 takes the hierarchical model structure (as defined in $\mathcal{B}, \mathcal{E}(\mathcal{B})$) and function composition data, and discards the GUI-specific data $\wp$. We can then use information regarding hierarchical model structure to compute the function composition data. Figure 1 summarises the two transformations, using the NAND example.

We can use the NAND example to see how such a transformation takes place. The GUI model $\langle \mathcal{B}, \mathcal{E}(\mathcal{B}), \wp \rangle$ consists of the block tuples for the AND and the NOT gates. We can assume that we retain the data for $\mathcal{B}, \mathcal{E}(\mathcal{B})$ for both gates, discard the GUI-specific data $\wp$, and use $\mathcal{B}, \mathcal{E}(\mathcal{B})$ to compute all composite function transformations.

Abstract-Model to CSP Mapping

τ_2 takes the hierarchical model structure and function composition data and rewrites it as a flat CSP. To generate a flat model, we must discard all hierarchical information, i.e., rewrite the model solely in terms of primitive abstract model entities.

If we specify primitive block ports (inputs and outputs) using $\pi = I \cup O$, then the abstract model contains a set of primitive entities given by the union of the ports and the modes, $\pi \cup \mu$, together with the functions $\mathcal{F}'$. It is the primitive entities that become the variables in a flat CSP. In other words, the CSP variables are given by $\mathcal{X} = \tau_2(\pi \cup \mu)$. Similarly, if $\mathcal{D}_\alpha$ refers to the domain for α, the CSP domains are given by $\mathcal{D} = \tau_2(\mathcal{D}_\pi \cup \mathcal{D}_\mu)$. We derive the constraints from the primitive, composite functions $\mathcal{F}'$ as defined in the abstract model: $\mathcal{C} = \tau_2(\mathcal{F}')$.

Example: One of the best ways to see the relations between the abstract model and the flat CSP is to represent the CSP in terms of a constraint graph. We can use the NAND example to see how such a transformation takes place. The CSP model of Figure 1 shows the constraint graph, together with some variable domain specifications and constraints, for the NAND example.

- To generate the variables for the CSP, we ignore all ports/modes other than the ports/modes for the primitive blocks. In the NAND example, we ignore the abstract NAND model, and use the ports/modes from the AND and NOT blocks, namely {AND.I1, AND.I2, AND.O, AND} and {NOT.I, NOT.O, NOT}.[5]
- To generate the domains for the CSP, we simply include the domains for the variables just identified.
- Computing the constraints is the most difficult part of the mapping, and depends on the constraint solver that we are targeting. To give an idea of constraint computation, we convert the logic equations of the abstract model into a set of allowable tupes for subsets of variables. We can do this by using a theorem-prover to generate the truth tables for the CSP. Alternatively, if we were targeting a more powerful constraint solver, we could just use the logic equations directly.

[5] We need to merge AND.O and NOT.I into a single variable, since they are actually the same entity.

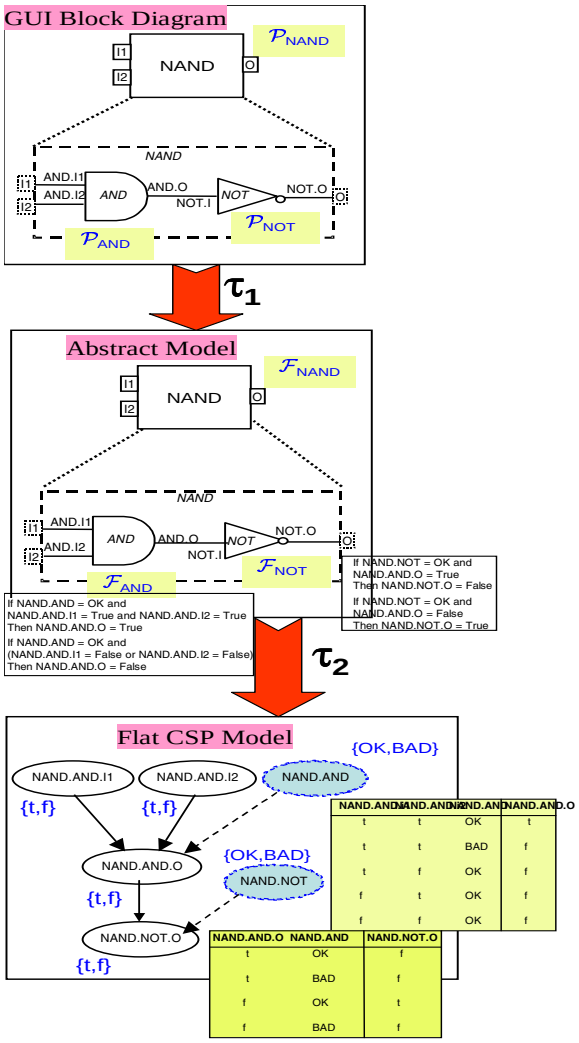

Figure 1. Transformations from GUI to abstract model to flat CSP, using a NAND electrical circuit example. $\mathcal{P}_{NAND}$ and $\mathcal{F}_{NAND}$ denote GUI and functional data for the NAND-gate, respectively.

3 Graph Grammar Framework

One desideratum for our modeling framework is the ability to apply verification and validation (V&V) to the entire modeling process, from GUI-based model construction through to CB-model generation and inference. We use the graph grammars framework [2, 9] for this purpose. This can overcome one large deficiency of model building, namely the inability to relate the process of using a GUI for model construction to the model generated at the end.

We can use graph grammars for each stage of modeling as follows:

GUI-modeling Graph grammar formalisations of visual transformation processes as encountered during GUI-based modeling have been developed by several authors, including [11, 4].

Abstract Model Generation Transforming a graph grammar GUI formalisation into an abstract model can itself be modeled by a graph grammar.

CB-model generation and inference Transforming a graph grammar abstract model formalisation into a CB-model can itself be modeled by a graph grammar, as can the CB-inference on the CB-model.

In the following, we assume that we have a graph grammar that formalises the GUI-based modeling process, e.g., [11, 4]. Hence, we must then formalise that the model transformations τ_1, τ_2 to abstract model and CSP, respectively. We also assume that the graph grammar model we adopt is a hierarchical, attributed model [13], since such a framework is necessary to capture the hierarchical aspects of our models, and the model attributes, such as GUI data, block functions, and CSP constraints.

Transformation τ_1: Given $\tau_1 : \langle \mathcal{B}, \mathcal{E}(\mathcal{B}), \wp \rangle \rightarrow \langle \mathcal{B}, \mathcal{E}(\mathcal{B}), \mathcal{F} \rangle$, the transformation rules that we have to specify are trivial. The first set of rules basically map the GUI model structure directly onto the abstract model structure. The second set of rules map the GUI attributes ($\wp$) onto abstract model attributes ($\mathcal{F}$).

Transformation τ_2: Given $\tau_1 : \langle \mathcal{B}, \mathcal{E}(\mathcal{B}), \mathcal{F} \rangle \rightarrow \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, we have to specify three main types of transformation rules. The first set of rules ($\mathcal{R}_{21}$) basically map the GUI model structure directly onto the abstract model structure. The second set of rules map the GUI attributes ($\wp$) onto abstract model attributes ($\mathcal{F}$). Rules $\mathcal{R}_{22}$ map the variable domains, and rules $\mathcal{R}_{23}$ map the constraints.

$\mathcal{R}_{21}$ To generate the variables for the CSP, we ignore all ports/modes other than the ports/modes for the primitive blocks. In the NAND example, we ignore the abstract NAND model, and use the ports/modes from the AND and NOT blocks, namely {AND.I1, AND.I2, AND.O, AND} and {NOT.I, NOT.O, NOT}.[6]

$\mathcal{R}_{22}$ To generate the domains for the CSP, we simply include the domains for the variables just identified.

$\mathcal{R}_{23}$ Computing the constraints is the most difficult part of the mapping, and depends on the constraint solver that we are targeting. To give an idea of constraint computation, we convert the logic equations of the abstract model into a set of allowable types for subsets of variables. We can do this by using a theorem-prover to generate the truth tables for the CSP. Alternatively, if we were targeting a more powerful constraint solver, we could just use the logic equations directly.

We can perform this transformation on a block-by-block basis, using the transformation schema given below.

[6] We need to merge AND.O and NOT.I into a single variable, since they are actually the same entity.

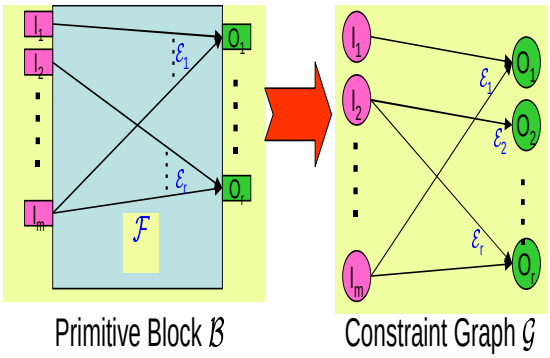

Figure 2. Visual representation of production rule for mapping from block in abstract model representation to constraint graph model.

Figure 2 gives a visual idea of the production rules entailed in generating a constraint graph model from a primitive block B in an abstract model representation.

1. For every port and for the mode μ we generate a constraint graph node.
2. For every variable constructed, we assign a domain generated from the values of the corresponding port/mode.
3. We generate edges in the constraint graph $\mathcal{G}$ as follows. In block B (with m input ports and r outputs), with edges $\mathcal{E}(B)$, call $\mathcal{E}_j$ the edges incident on output O_j. We add edges to $\mathcal{G}$ from the input-port nodes to output-port node j, for $, j = 1, ..., r$, as given by $\mathcal{E}_j$. We then add an edge from the mode node to every output-port node.
4. We add constraints as follows. Given transformation function f for B, we have the following set of output-port transformations: $O_j = f(\mathcal{I}_j)$, for $j = 1, ..., r$. If we define a constraint over variables X as $C(X)$, then we add constraints corresponding to the output-port transformations: $C(\mathcal{I}_j, O_j)$, for $j = 1, ..., r$, based on function f.

This graph grammars approach has many advantages.

V&V Given a formally-verified set of rules, we can validate the CB model generated from a GUI representation using the entailed grammars.

Incremental model generation This approach is fully compatible with incremental model generation.

Code embeddability We can extend this procedure to generate embeddable code using another set of rules, thus providing quick and simple V&V of the embeddable code.

In practice, a user will develop a component library for a particular domain, verifying the correctness of each component. Next, the user will create a complex system by interconnecting components. The graph grammar process that we describe provides a means for justifying the

correctness of the model of the complex system, when only the correctness of the system components, and of the model development process are known. This provides a significant advantage over existing approaches, in which the complex system model has to be formally verified, a task which is arduous and expensive. Further, if the complex system model needs to be altered or the target platform changed (i.e., the embedded language needs to be changed), our approach provides a clear, cost-effective solution; today, the modified system model often needs to be re-verified from scratch.

4 Related Work

The literature contains a large number of papers describing the use of CB languages for model creation, but to our knowledge this is the first paper that provides a formal framework for the entire modeling process. From the constraints perspective, most papers focus on representational efficiency, e.g., [3], or on describing the domain application, e.g., [10].

The graph grammar literature contains several papers describing a graph grammars formalisation of CSPs and CSP inference, but none describing a CSP-based modeling process. For example, [8] describe how a graph grammar can capture a CSP model, and [5] focuses on using graph grammars to formalise CSP inference.

5 Conclusions

This article described a formal framework that provides a clear process and well-developed semantics for modeling complex systems by using a GUI and then generating models using a constraint-based language. To address the aspects of a model that are captured by a CB representation, we showed that two well-known approaches, function-based and process-based approaches, are suited to using a CB representation. We proposed a formal framework for building function-based models, and applied graph grammars to provide verification and analysis tools during all stages of the model construction process.

References

[1] R.B. Altman and J.F. Brinkley. Probabilistic constraint satisfaction with structural models: Application to organ modeling. In *Proc. Annual Symposium on Computer Applications*. 1993.

[2] M. Andries, G. Engels, A. Habel, B. Hoffmann, H. Kreowski, S. Kuske, D. Plump, A. Schorr, and G. Taentzer. Graph transformation for specification and programming. *Sci. Comput. Program.*, 34(1):1–54, 1999.

[3] A. Bakewell, A. M. Frisch, and I. Miguel. Towards automatic modelling of constraint satisfaction problems: A system based on compositional refinement. In *Proc. 2nd Intl. Workshop on Modelling and Reformulating CSPs, Kinsale, Ireland*, pages 2–17, 2003.

[4] L. Baresi and M. Pezzo. Formal interpreters for diagram notations. *ACM Trans. Softw. Eng. Methodol.*, 14(1):42–84, 2005.

[5] C. Castro. Building Constraint Satisfaction Problem Solvers Using Rewrite Rules and Strategies. *Fundamenta Informaticae*, 34(3):263–293, 1998.

[6] C.-C. Cheng and S. F. Smith. Applying constraint satisfaction techniques to job shop scheduling. *Ann. of Operations Research*, 70:327–357, 1997.

[7] A. Darwiche. Model-based diagnosis using structured system descriptions. *Journal of Artificial Intelligence Research*, 8:165–222, 1998.

[8] I. Dotu and J. de Lara. Rapid Prototyping by Means of Meta-Modelling and Graph Grammars. An Example with Constraint Satisfaction. In *JISBD03*, pages 401–410, Alicante, Spain, Nov. 2003.

[9] F. Drewes, B. Hoffmann, and D. Plump. Hierarchical graph transformation. *J. Comput. Syst. Sci.*, 64(2):249–283, 2002.

[10] S. M. Fohn, J. S. Liau, A. R. Greef, R. E. Young, and P. J. O'Grady. Configuring computer systems through constraint-based modeling and interactive constraint satisfaction. *Comput. Ind.*, 27(1):3–21, 1995.

[11] M. Goedicke, P. Tropfner, and B.E. Enders-Sucrow. Hierarchical specification of graphical user interfaces using a graph grammar approach. *Trans. of the SDPS*, 5(1), 2001.

[12] D.C. Karnopp, D.L. Margolis, and R.C. Rosenberg. *Systems Dynamics: A Unified Approach*. John Wiley and Sons, 1990.

[13] G. Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. I: Foundations*. World Scientific, 1997.

[14] L. C. Schmidt, H. Shetty, and S. C. Chase. A graph grammar approach for structure synthesis of mechanisms. *J. Mechanical Design*, 122(4):371–376, 2000.

[15] R. B. Stone and K. L. Wood. Development of a functional basis for design. *J. Mechanical Design*, 122(4):359–370, 2000.

[16] J. Summers, B. Bettig, and J. Shah. The design exemplar: A new data structure for design automation. *J. Mechanical Design*, 126(4):775–787, 2004.

[17] J. D. Summers, N. Vargas-Hernandez, Z. Zhao, J. Shah, and Z. Lacroix. Comparative study of representation structures for modeling function and behavior of mechanical devices. In *DETC2000: Computers in Engineering*, Pittsburgh, PA, Sept. 2001.

[18] H. Vangheluwe and J. de Lara. Foundations of Multi-paradigm Modeling and Simulation. In *WSC '03*, pages 595–603, 2003.

[19] Mengchu Zhou and Frank DiCesare. *Petri Net Synthesis for Discrete Event Control of Manufacturing Systems*. Kluwer Academic Publishers, 1993.